Singleton
Factory
Method

Builder
Abstract
Factory

Adapter
Composite
Bridge

Proxy
Command
Observer

Template
Method
Strategy

Design Patterns (GoF) in .NET

Aniruddha Chakrabarti
Senior Architect

Agenda

- What is Design Pattern
- History of Design Pattern
- Background
  - Principles of OOP and OOAD
  - What is Good Design
  - Other Design Patterns and Principles (SOLID, GRASP)
  - Anti Patterns
- Classification of different types of design patterns
- Creational Design Patterns
- Structural Design Patterns
- Behavioral Design Patterns
- Architecture Patterns (Brief)
- Takeaway

OO Basic Tenants

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance
- Loose/Low Coupling
  - low dependency between classes;
  - low impact in a class of changes in other classes
  - high reuse potential
- High Cohesion
  - measure of how strongly-related or focused the responsibilities of a single module are
  - If methods of a class tend to be similar in many aspects, then the class have high cohesion
  - In highly-cohesive system, code readability and the likelihood of reuse is increased, while complexity is kept manageable

OO Principles

- Encapsulate what varies
- Favor Composition over Inheritance
- Program to interface not implementation
- Strive for loosely coupled design between objects that interact

Interface

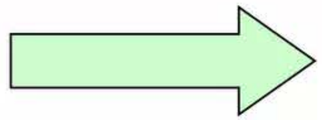
- Like an abstract base class: any non-abstract type inheriting the interface must implement all its members.
- Can not be instantiated directly.
- Can contain events, indexers, methods and properties.
- Contain no implementation of methods.
- Classes and structs can inherit from more than one interface.
- An interface can itself inherit from multiple interfaces.

Difference between Interface and abstract base class

- Abstract class can have non abstract methods, methods of an interface are effectively abstract since they contain only the signature
- A class can implement any no of interfaces, but can subclass at most one abstract class.
- An abstract class can declare and use variables, an interface can not.
- An abstract class can have methods whose access is public, internal, protected, protected internal and private. Interface members implicitly have public access.
- An abstract class can declare constructor, an interface can not.
- An abstract class can have delegate, an interface can not.

Interface Example – Implicit Implementation

```
public class Calc : ICalc
{
    public int Add(int i, int j) ...

    public int Subtract(int i, int j)...

    public int Multiply(int i, int j)...

    public int Divide(int i, int j)...
}

public interface ICalc
{
    int Add(int i, int j);
    int Subtract(int i, int j);
    int Multiply(int i, int j);
    int Divide(int i, int j);
}
```



✘

```
// Should be avoided
Calc calc = new Calc();
int res = calc.Add(20, 30);

var calc2 = new Calc();
res = calc2.Add(30, 40);
```

Does compile and work, but should be avoided

✓

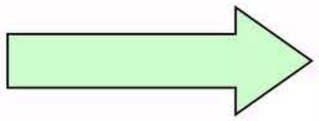
```
ICalc calc = new Calc();
int res = calc.Add(20, 30);
```

Object of type Interface should be used

Interface Example – Explicit Implementation

```
public class Calc : ICalc
{
    // Using public is not valid - causes compile error
    //public int ICalc.Add(int i, int j) { }

    int ICalc.Add(int i, int j) ...

    int ICalc.Subtract(int i, int j)...

    int ICalc.Multiply(int i, int j)...

    int ICalc.Divide(int i, int j)...
}
```

```
public interface ICalc
{
    int Add(int i, int j);
    int Subtract(int i, int j);
    int Multiply(int i, int j);
    int Divide(int i, int j);
}
```



```
Calc calc = new Calc();
int res = calc.Add(20, 30);
```

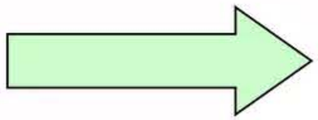
Since the Interface is implemented explicitly, it does NOT even compile

```
ICalc calc = new Calc();
int res = calc.Add(20, 30);
```

Object of type Interface should be used – compiles and works perfectly.

Interface Example – Business Logic Layer

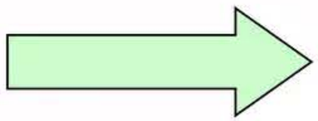
```
// DTO / Entity
public abstract class EntityBase
{
    public Guid Id { get; set; }
    public string CreatedBy { get; set; }
    public DateTime CreatedDate { get; set; }
    public string UpdatedBy { get; set; }
    public DateTime? UpdatedDate { get; set; }
    public bool IsChanged { get; set; }
}

public class Student : EntityBase
{
    public string RoleNo { get; set; }
}

public class Teacher : EntityBase
{
    public string EmployeeId { get; set; }
    public string[] SubjectsTaught { get; set; }
}
```



```
// Business Logic Layer
public interface IEntityManager
{
    // CRUD Operations
    List<EntityBase> GetAll();
    EntityBase Get(Guid id);
    void Save(EntityBase entity);
    void Delete(Guid id);
}

public class StudentManager : IEntityManager
{
    public List<EntityBase> GetAll()
    {
        // Following line does not compile
        // return new List<Student>();
        return new List<EntityBase>();
    }
    public EntityBase Get(Guid id)
    {
        return new Student();
    }
    public void Save(EntityBase entity)...
    public void Delete(Guid id)...
}

public class TeacherManager : IEntityManager
{
    public List<EntityBase> GetAll()...
    public EntityBase Get(Guid id)...
    public void Save(EntityBase entity)...
    public void Delete(Guid id)...
}
```

Generic Interface Example

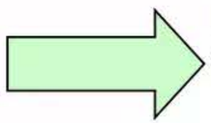
```
// DTO / Entity
public abstract class EntityBase
{
    public Guid Id { get; set; }
    public string CreatedBy { get; set; }
    public DateTime CreatedDate { get; set; }
    public string UpdatedBy { get; set; }
    public DateTime? UpdatedDate { get; set; }
    public bool IsChanged { get; set; }
}

public class Student : EntityBase
{
    public string RoleNo { get; set; }
}

public class Teacher : EntityBase
{
    public string EmployeeId { get; set; }
    public string[] SubjectsTaught { get; set; }
}
```



```
// Business Logic Layer
public interface IEntityManager<TEntity>
{
    // CRUD Operations
    List<TEntity> GetAll();
    TEntity Get(Guid id);
    void Save(TEntity entity);
    void Delete(Guid id);
}

public class StudentManager : IEntityManager<Student>
{
    public List<Student> GetAll()
    {
        // Now the following line does compile
        return new List<Student>();
        //return new List<EntityBase>();
    }
    public Student Get(Guid id)
    {
        return new Student();
    }
    public void Save(Student student)...
    public void Delete(Guid id)...
}

public class TeacherManager : IEntityManager<Teacher>
{
    public List<Teacher> GetAll()...
    public Teacher Get(Guid id)...
    public void Save(Teacher teacher)...
    public void Delete(Guid id)...
}
```

Generic Interface Example – making it perfect

```
// DTO / Entity
public abstract class EntityBase<TIdentity>
{
    public TIdentity Id { get; set; }
    public string CreatedBy { get; set; }
    public DateTime CreatedDate { get; set; }
    public string UpdatedBy { get; set; }
    public DateTime? UpdatedDate { get; set; }
    public bool IsChanged { get; set; }
}

public class Student : EntityBase<Guid>
{
    public string RoleNo { get; set; }
}

public class Teacher : EntityBase<string>
{
    public string EmployeeId { get; set; }
    public string[] SubjectsTaught { get; set; }
}
```

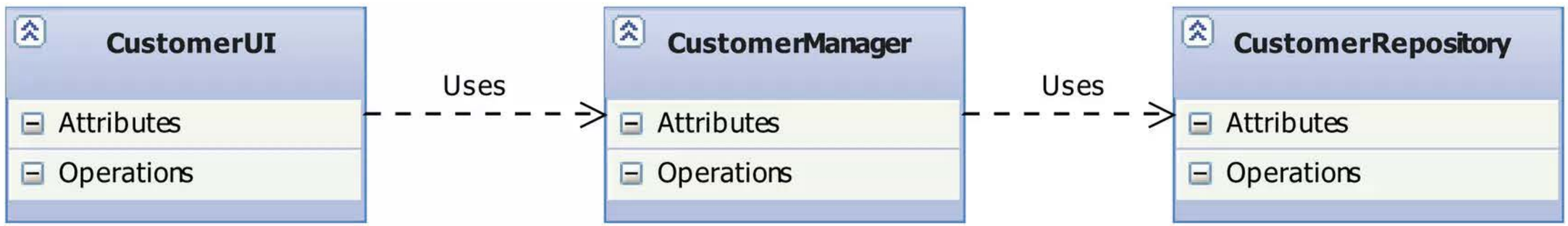


```
// Business Logic Layer
public interface IEntityManager<TEntity, TIdentity>
{
    // CRUD Operations
    List<TEntity> GetAll();
    TEntity Get(TIdentity id);
    void Save(TEntity entity);
    void Delete(TIdentity id);
}

public class StudentManager : IEntityManager<Student, Guid>
{
    public List<Student> GetAll()
    {
        // Now the following line does compile
        return new List<Student>();
        //return new List<EntityBase>();
    }
    public Student Get(Guid id)
    {
        return new Student();
    }
    public void Save(Student student)...
    public void Delete(Guid id)...
}

public class TeacherManager : IEntityManager<Teacher, string>
{
    public List<Teacher> GetAll()...
    public Teacher Get(string id)...
    public void Save(Teacher teacher)...
    public void Delete(string id)...
}
```

Class Relationship

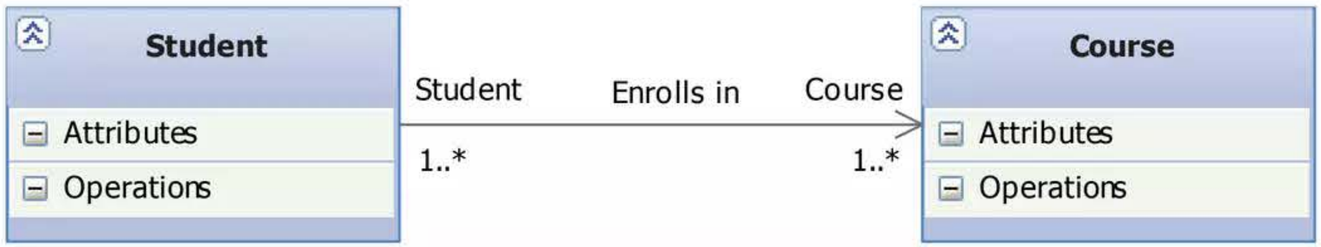
- Dependency
- Association
- Aggregation
- Composition
- Generalization / Inheritance

Dependency

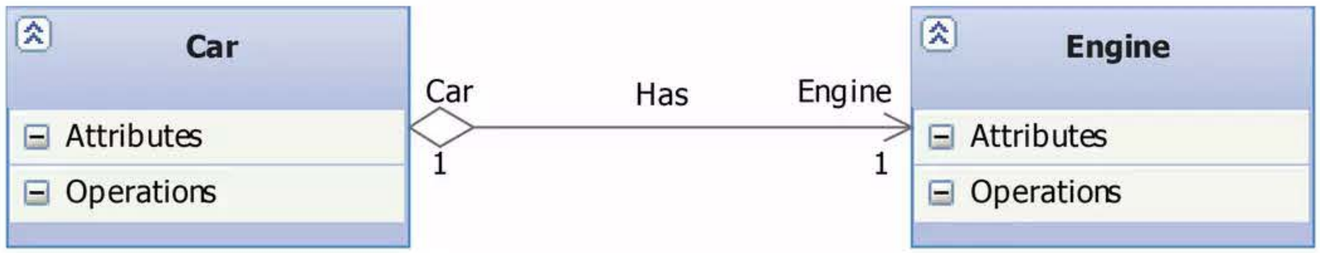


Dependency - Weaker form of relationship which indicates that one class depends on another because it uses it at some point of time.
Dependency exists if a class is a parameter variable or local variable of a method of another class.

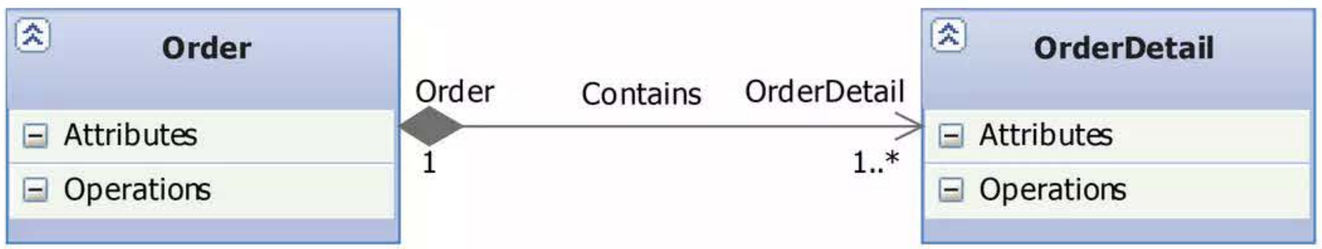
Association, Aggregation and Composition



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

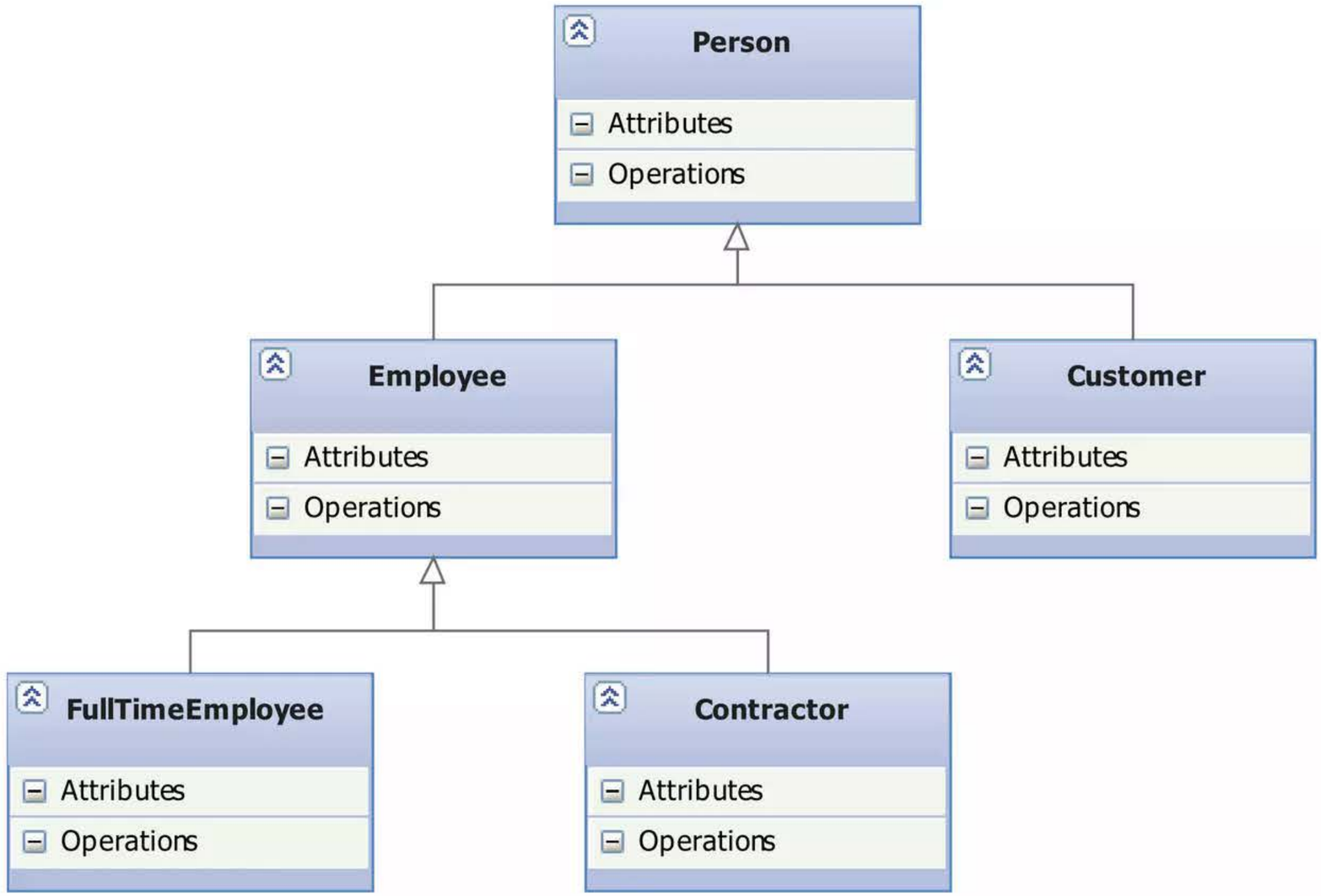


Aggregation - Whole part relationship. Part can exist without Whole.
(Engine can exist even if Car is destroyed, the same Engine could be used in a different Car)



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



Design Principles (SOLID)

- **SOLID Principles are principles of class design.**
- **S**RP: Single Responsibility Principle
  - An object should have only a single responsibility & all the responsibility should be entirely encapsulated by the class.
  - There should never be more than one reason for a class to change
- **O**CP: Open/Closed Principle
  - Software entities should be open for extension, but closed for modification
- **L**SP: Liskov Substituion Principle
  - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program
- **I**SP: Interface Segregation Principle
  - many client specific interfaces are better than one general purpose interface
  - once an interface has gotten too 'fat' split it into smaller and more specific interfaces so that any clients of the interface will only know about the methods that pertain to them. No client should be forced to depend on methods it does not use
- **D**IP: Dependency Inversion Principle
  - Depend upon Abstractions. Do not depend upon concretions.
  - Dependency Injection (DI) is one method of following this principle.

GRASP Pattern

- Acronym for **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns.
- Assigning responsibilities to classes is a critical aspect of object-oriented design.
- Appropriate assignment of responsibilities to classes is the key to successful design.
- There are fundamental principles in assigning responsibilities that experienced designers apply.
- These principles are summarized in the GRASP patterns.
- Has nine core principles that object-oriented designers apply when assigning responsibilities to classes and designing message interactions.

GRASP Patterns (Cont'd)

- Expert.
- Creator.
- Controller.
- Low Coupling.
- High Cohesion.
- Polymorphism.
- Pure Fabrication.
- Indirection.
- Don't Talk to Strangers.

Smell of good code

- What is good code?
- What is good design?
  - Maintainable/Readable/Easily understandable
  - Reusable
  - Easier to change if requirement chages
  - Performs better
  - Modular/Componentized
  - Takes care of all “ility” concerns
  - …
  - …
  - …

Smells within Class

- Comments
  - Should only be used to clarify "why" not "what". Can quickly become verbose and reduce code clarity.
- Long Method
  - The longer the method the harder it is to see what it's doing.
- Long Parameter List
  - Don't pass everything the method needs. Pass enough so that the method gets everything it needs.
- Duplicated Code
- Large Class
  - A class that is trying to do too much can usually be identified by looking at how many instance variables it has. When a class has too many instance variables, duplicated code cannot be far behind.
- Type Embedded in Name
  - Avoid redundancy in naming. Prefer Schedule.Add(course) to Schedule.AddCourse(course)
- Uncommunicative Name
  - Choose names that communicate intent (pick the best name for the time, change it later if necessary).
- Inconsistent Names
  - Use names consistently.
- Dead Code
  - A variable, parameter, method, code fragment, class is not used anywhere (perhaps other than in tests).
- Speculative Generality
  - Don't over-generalize your code in an attempt to predict future needs.

Smell Between Classes

- Primitive Obsession
  - Use small objects to represent data such as money (which combines quantity and currency) or a date range object
- Data Class
  - Classes with fields and getters and setters and nothing else (aka, Data Transfer Objects - DTO)
- Data Clumps
  - Clumps of data items that are always found together.
- Refused Bequest
  - Subclasses don't want or need everything they inherit. Liskov Substitution Principle (LSP) says that you should be able to treat any subclass of a class as an example of that class.
- Inappropriate Intimacy
  - Two classes are overly enter twined.
- Lazy Class
  - Classes that aren't doing enough should be refactored away.
- Feature Envy
  - Often a method that seems more interested in a class other than the one it's actually in. In general, try to put a method in the class that contains most of the data the method needs.
- Message Chains
  - This is the case in which a client has to use one object to get another, and then use that one to get to another, etc. Any change to the intermediate relationships causes the client to have to change.
- Middle Man
  - When a class is delegating almost everything to another class, it may be time to refactor out the middle man.
- Divergent Change
  - Occurs when one class is commonly changed in different ways for different reasons. Any change to handle a variation should change a single class.
- Shotgun Surgery
  - The opposite of Divergent Change. A change results in the need to make a lot of little changes in several classes.
- Parallel Inheritance Hierarchies
  - Special case of Shotgun Surgery. Every time you make a subclass of a class, you also have to make a subclass of another.

What is Design Pattern

- A *pattern* is a recurring solution to a standard problem, in a context.
- 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.

List of Design Patterns - 23

- 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

Singleton
Factory
Method

Builder
Abstract
Factory

Adapter
Composite
Bridge

Proxy
Command
Observer

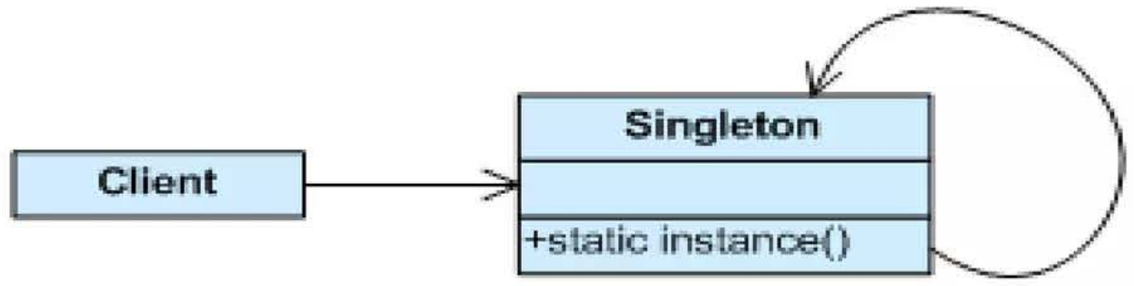
Template
Method
Strategy

Creational Design Patterns

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.
- The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects (say, five).
- 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 Class Diagram



Singleton in .NET BCL

- HttpContext class in ASP.NET implements Singleton
  - The class can not be instantiated using new. The current instance of HttpContext could be retrieved using HttpContext.Current property.
  - HttpContext.Current represents current ASP.NET Request context and this context is available throughout the course of an ASP.NET request starting with the HttpApplication.BeginRequest pipeline event all the way through HttpApplication.EndRequest
- OperationContext class in WCF (System.ServiceModel)
  - Provides access the current operation execution environment.
  - Used to access callback channels in duplex services, to store extra state data across portions of the operations, and to access incoming message headers and properties as well as add outgoing message headers and properties.

```
HttpContext context = HttpContext.Current;

// HttpContext exposes many important properties
// like Request and Response
HttpRequest request = context.Request;
HttpResponse response = context.Response;
HttpApplicationState application = context.Application;
HttpSessionState session = context.Session;
HttpServerUtility server = context.Server;
```

```
OperationContext currentOpContext =
    OperationContext.Current;

var sessionId = currentOpContext.SessionId;
var host = currentOpContext.Host;
var instanceContext = currentOpContext.InstanceContext;
var securityContext = currentOpContext.ServiceSecurityContext;
var requestContext = currentOpContext.RequestContext;
```

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. CLR takes care of the variable initialization. The class is marked **sealed** to prevent derivation, which could add instances.
- In addition, the variable is marked **readonly**, which means that it can be assigned only during static initialization (which is shown here) or in a class constructor.

Multithreaded Singleton in .NET

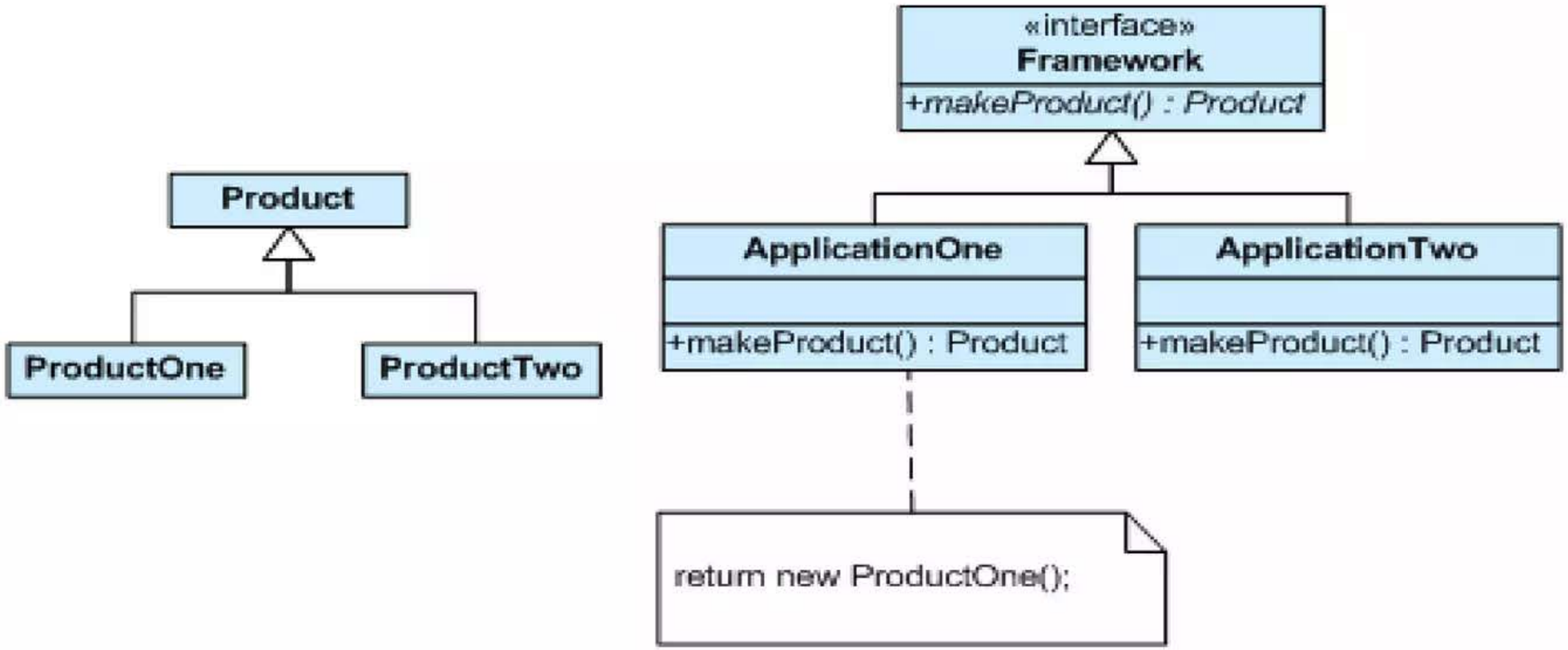
```
public sealed class Singleton
{
    private static volatile Singleton instance;
    private static object syncRoot = new Object();

    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, as in the Static Initialization example – should use specific language capabilities to ensure that only one instance of the object is created in the presence of multiple threads.
- One of the more common solutions is to use the **Double-Check Locking** idiom to keep separate threads from creating new instances of the singleton at the same time.
- Ensures only one instance is created and only when the instance is needed. Variable is declared **volatile** to ensure that assignment to instance variable completes before instance variable can be accessed.
- Lastly, it uses a **syncRoot** instance to lock on, rather than locking on the type itself, to avoid deadlocks.

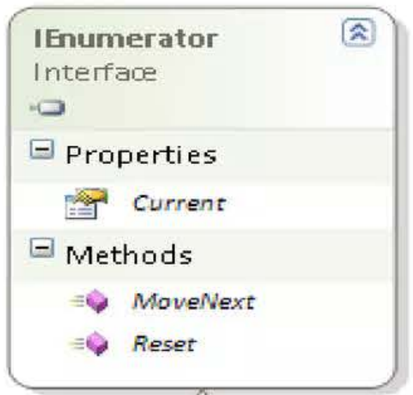
- Double-check locking approach solves thread concurrency problems while avoiding an exclusive lock in every call to the **Instance** property method. Also allows you to delay instantiation until the object is first accessed.
- In practice, an application rarely requires this type of implementation. In most cases, the static initialization approach is sufficient.

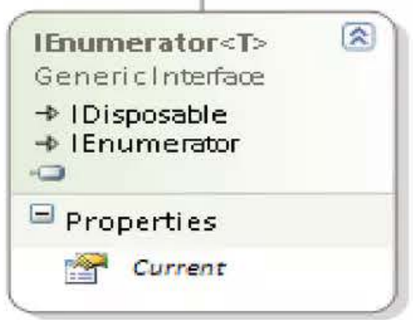
Factory Method

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

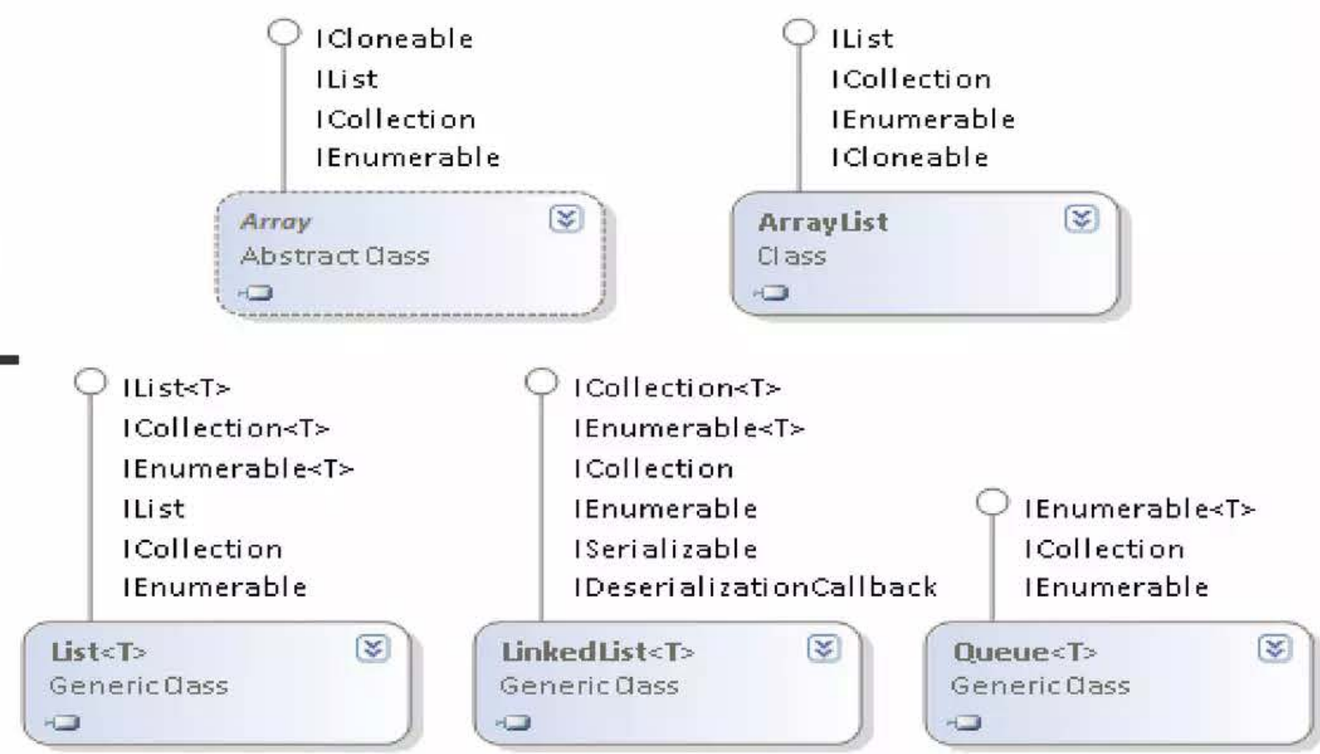


Factory Method in .NET BCL Collection



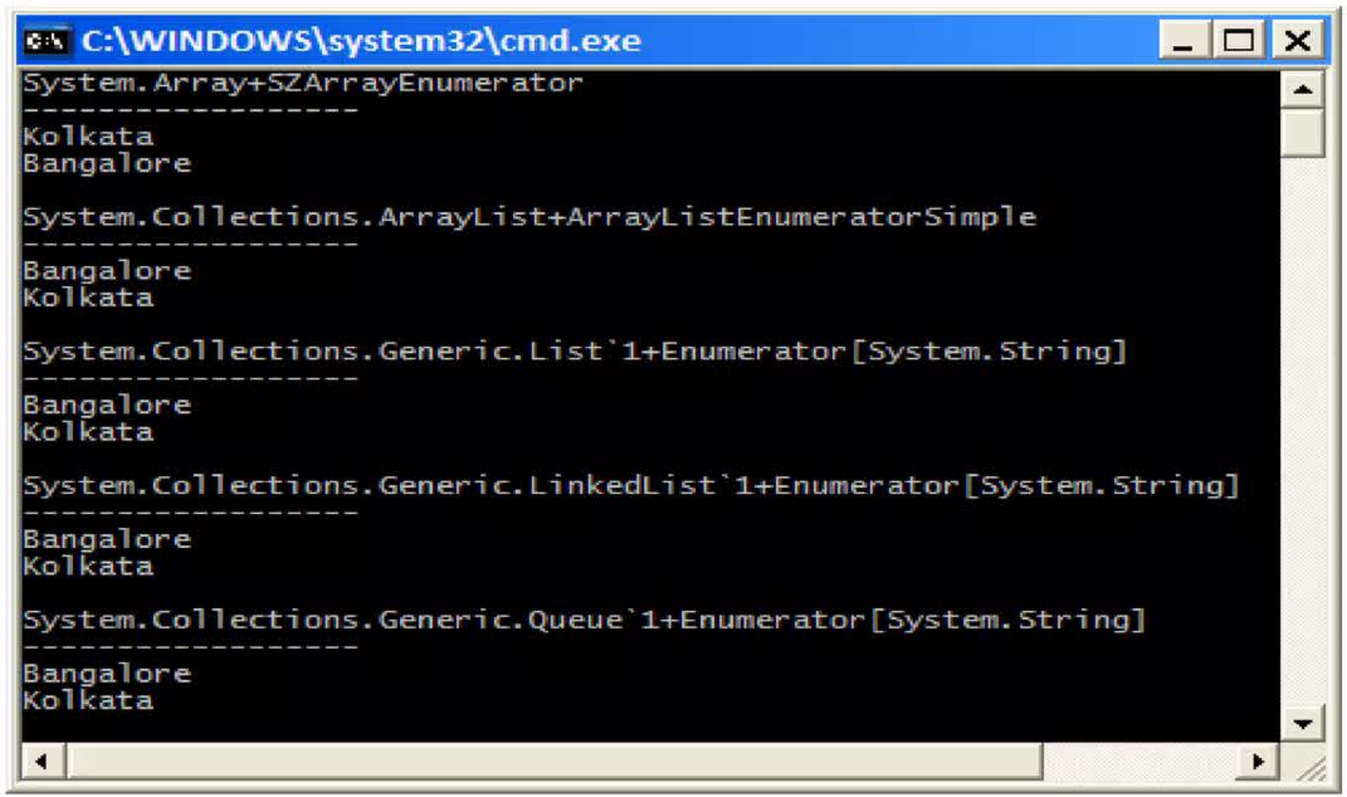


GetEnumerator() returns an specific enumerator type based on the collection class



Factory Method in .NET BCL Collection

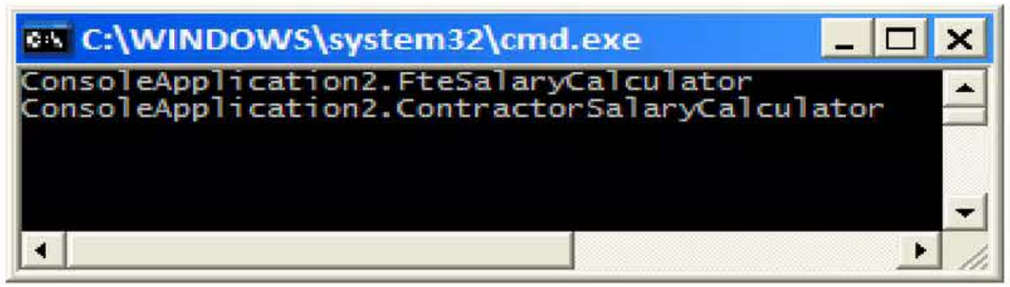
```
static void Main(string[] args)
{
    string[] cities = new string[] { "Kolkata", "Bangalore" };

    ArrayList cityAL = new ArrayList();
    cityAL.Add("Bangalore");
    cityAL.Add("Kolkata");

    List<string> cityList = new List<string>();
    cityList.Add("Bangalore");
    cityList.Add("Kolkata");

    LinkedList<string> cityLinkedList = new LinkedList<string>();
    cityLinkedList.AddLast("Bangalore");
    cityLinkedList.AddLast("Kolkata");

    Queue<string> cityQueue = new Queue<string>();
    cityQueue.Enqueue("Bangalore");
    cityQueue.Enqueue("Kolkata");

    Print(cities);
    Print(cityAL);
    Print(cityList);
    Print(cityLinkedList);
    Print(cityQueue);
}

public static void Print(IEnumerable items)
{
    Console.WriteLine(items.GetEnumerator().ToString());
    Console.WriteLine("------------------");

    foreach(object item in items)
        Console.WriteLine(item);

    Console.WriteLine();
}
```



How to implement 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);
```



Abstract Factory Pattern

- Provides a way to encapsulate a group of individual factories that have a common theme.
- In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interfaces to create the concrete objects that are part of the theme.
- Client does not know (or care) which concrete objects it gets from each of these internal factories since it uses only the generic interfaces of their products. It separates details of implementation of a set of objects from their general usage.

Factory in .NET: DbProviderFactory

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

DB Providers

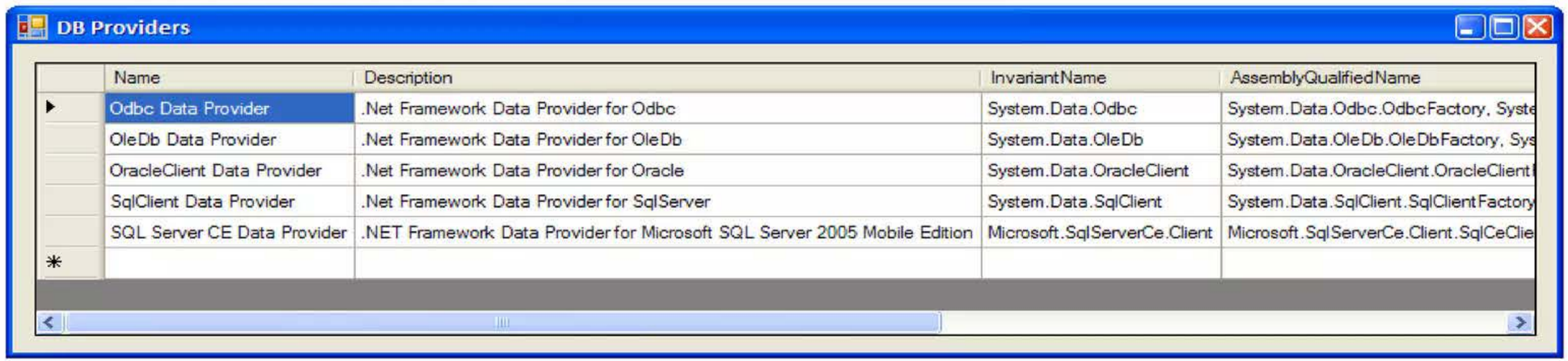
Name	Description	InvariantName	AssemblyQualifiedName
Odbc Data Provider	.Net Framework Data Provider for Odbc	System.Data.Odbc	System.Data.Odbc.OdbcFactory, System.Data, Version=4.0.0
OleDb Data Provider	.Net Framework Data Provider for OleDb	System.Data.OleDb	System.Data.OleDb.OleDbFactory, System.Data, 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

```
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());
```

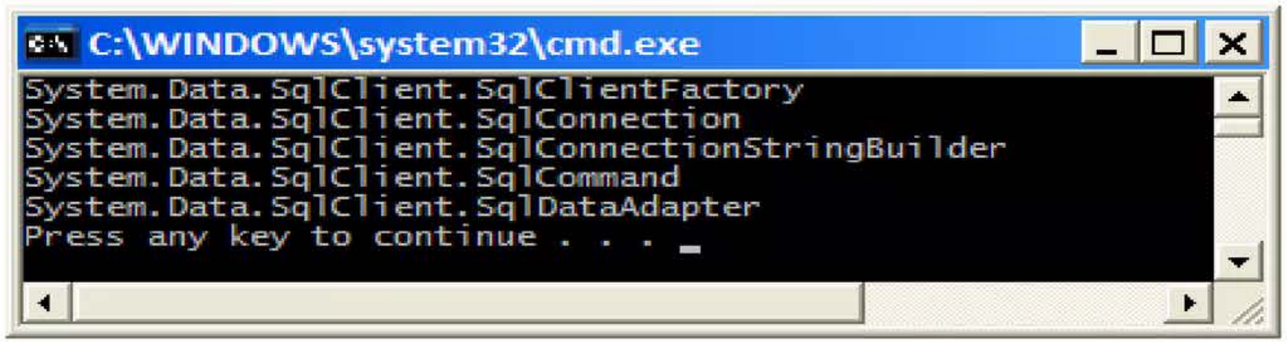
```
C:\WINDOWS\system32\cmd.exe
System.Data.SqlClient.SqlClientFactory
System.Data.SqlClient.SqlConnection
System.Data.SqlClient.SqlConnectionStringBuilder
System.Data.SqlClient.SqlCommand
System.Data.SqlClient.SqlDataAdapter
Press any key to continue . . .
```

Factory in .NET: DbProviderFactory

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

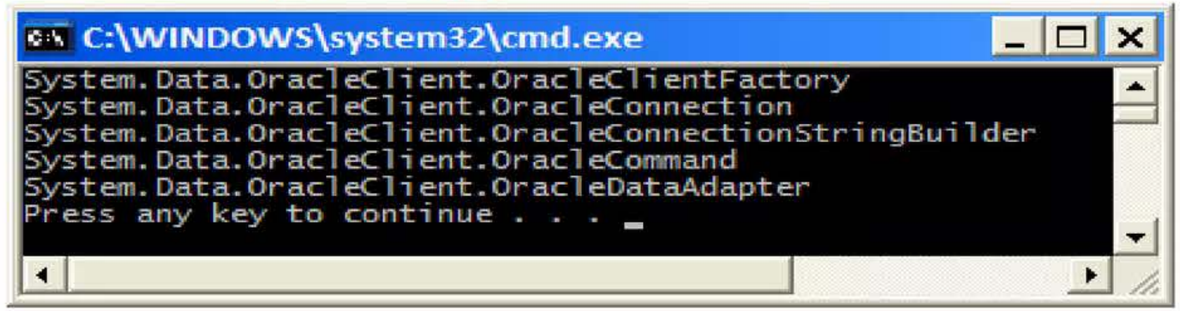



```
DbProviderFactory factory = DbProviderFactories.GetFactory("System.Data.SqlClient");
Console.WriteLine(factory.GetType());

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());
```

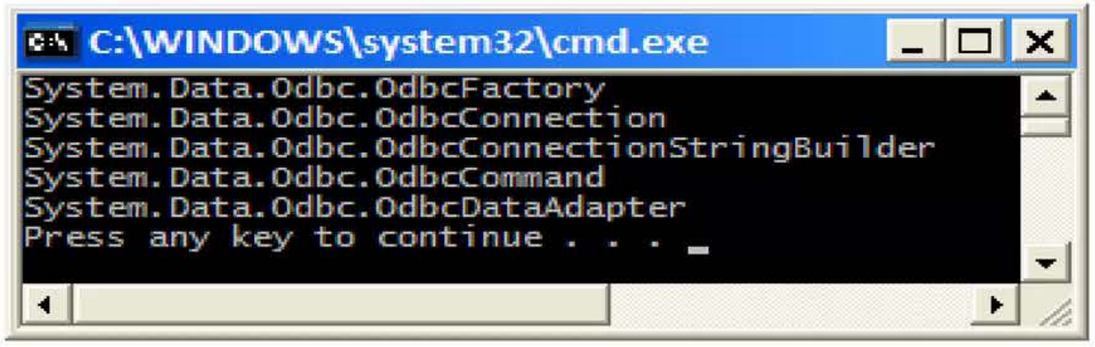


Factory in .NET: DbProviderFactory

```
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());
```




```
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());
```



Implement Factory in .NET

```
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);
```

Configuration based Abstract Factory

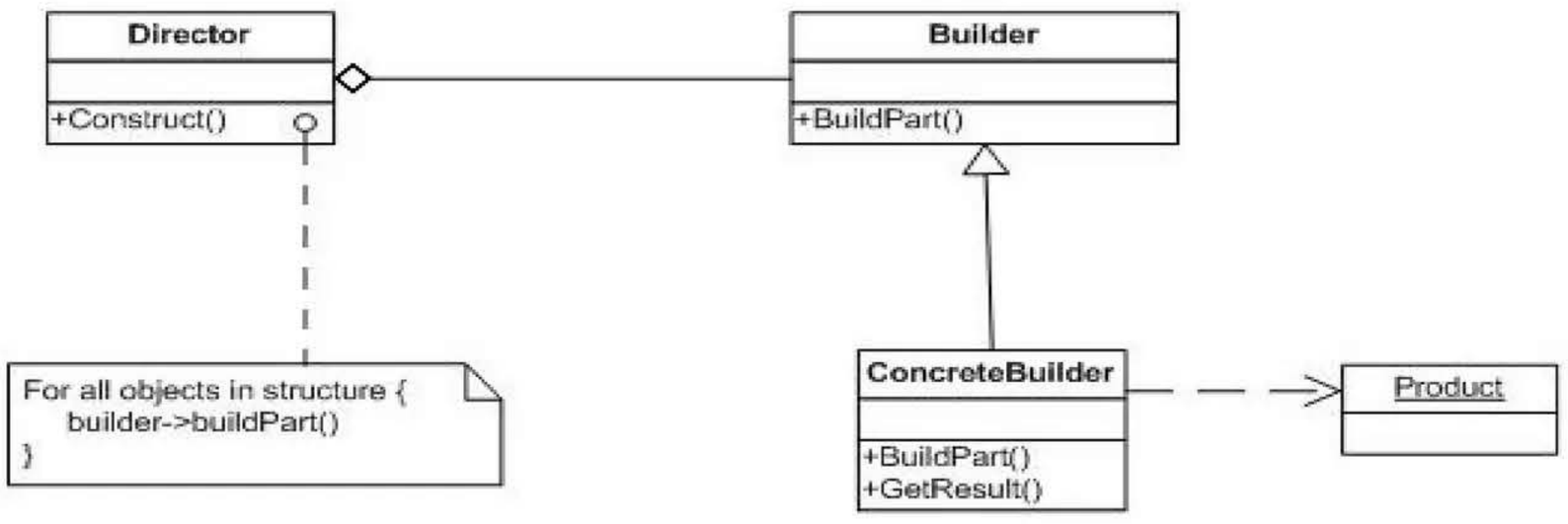
```
public static class SoftwareProfessionalFactory
{
    public static SoftwareProfessional GetSoftwareProfessional
        (WorkType workType)...

    public static SoftwareProfessional GetSoftwareProfessional(string work)
    {
        WorkType workType;

        if (Enum.TryParse<WorkType>(work, out workType))
            return GetSoftwareProfessional(workType);

        return null;
    }
}
```

```
<appSettings>
  <add key="work" value="Code" />
</appSettings>
```

```
string work;

// Do not use ConfigurationSettings class it it's obsolete
// Use ConfigurationManager class present in System.Configuration assembly
// work = ConfigurationSettings.AppSettings["work"];

work = ConfigurationManager.AppSettings["work"];
SoftwareProfessional prof = SoftwareProfessionalFactory.GetSoftwareProfessional(work);
// Returns a Developer
```

Builder Pattern

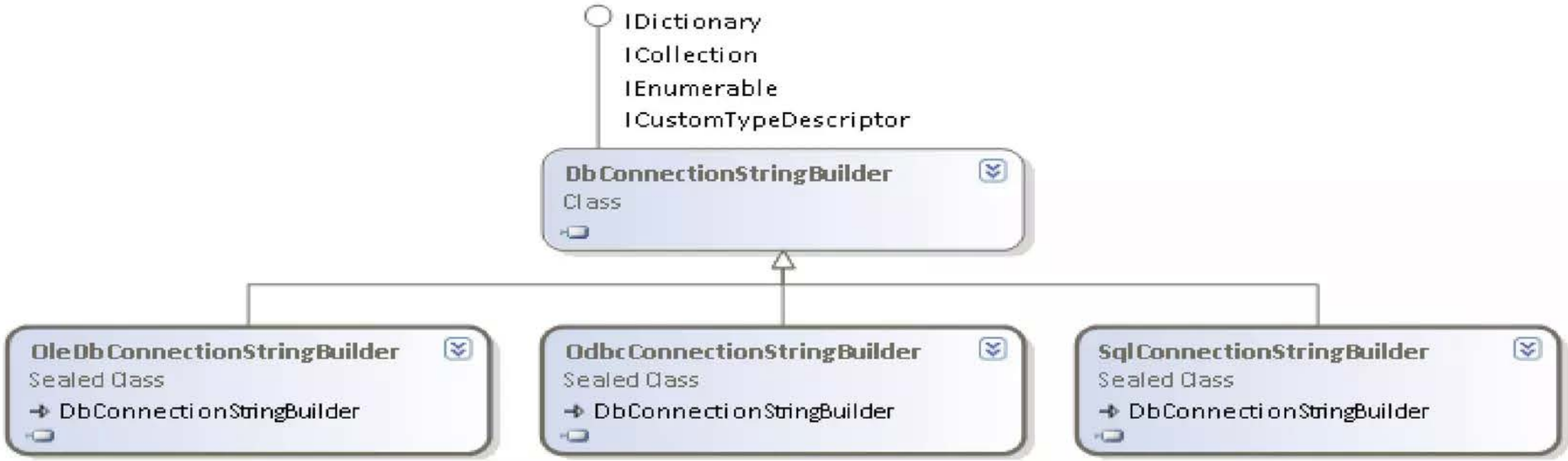
- Separate construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.
- Difference Between Builder and Factory
  - Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects - simple or complex.
  - Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
- Builder often builds a Composite.



Builder Design Pattern

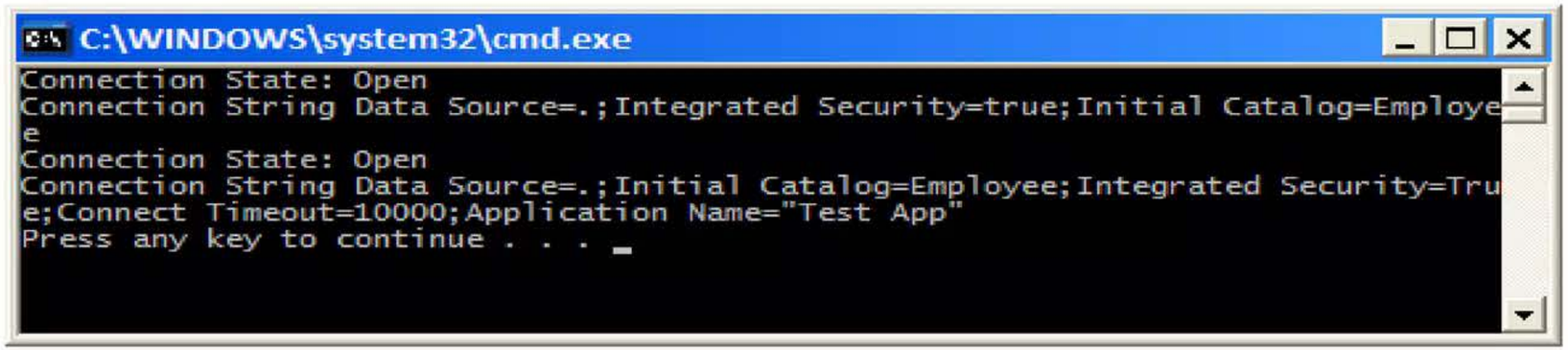
- Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
- Builder often builds a Composite.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex).
- Sometimes creational patterns are complementary: Builder can use one of the other patterns to implement which components are built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.

Builder Pattern in .NET BCL



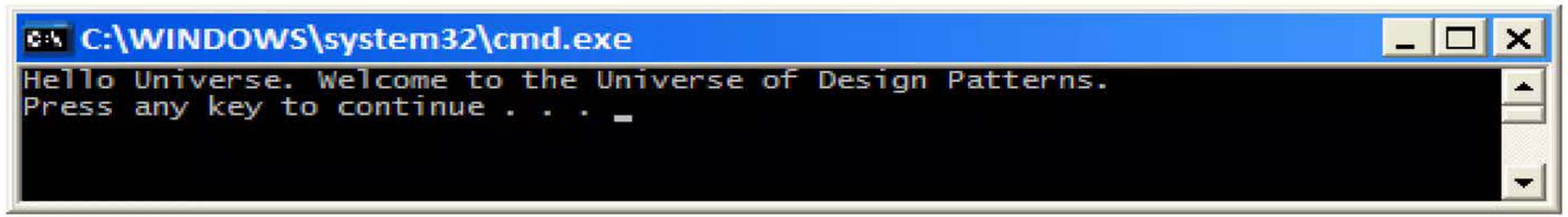
Builder: SqlConnectionStringBuilder in .NET

```
// 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);
```

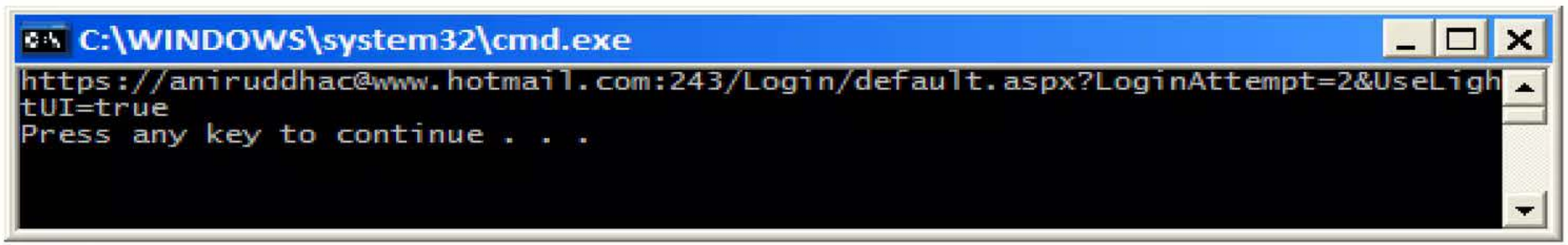


Builder: StringBuilder & UriBuilder in .NET

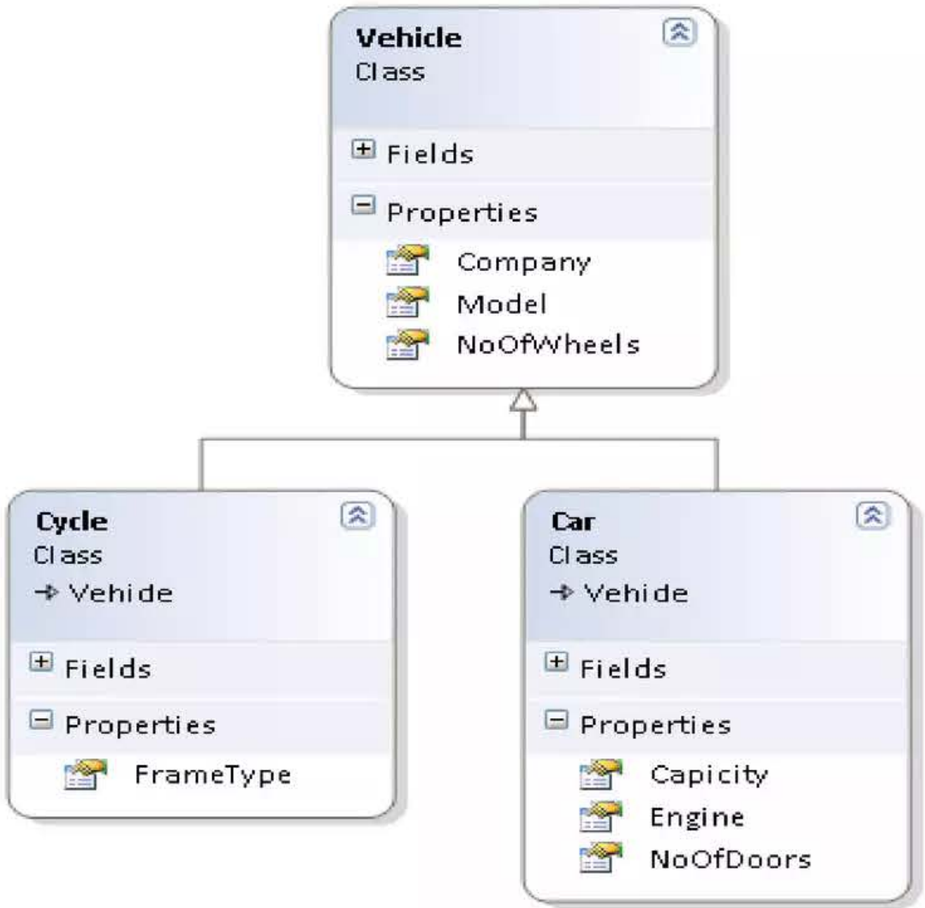
```
StringBuilder builder = new StringBuilder("Hello");
builder.Append(" Welcome to the World of Design Patterns. ");
builder.Insert(6, "World. ");
builder.Replace("World", "Universe");

string value = builder.ToString();
Console.WriteLine(value);
```

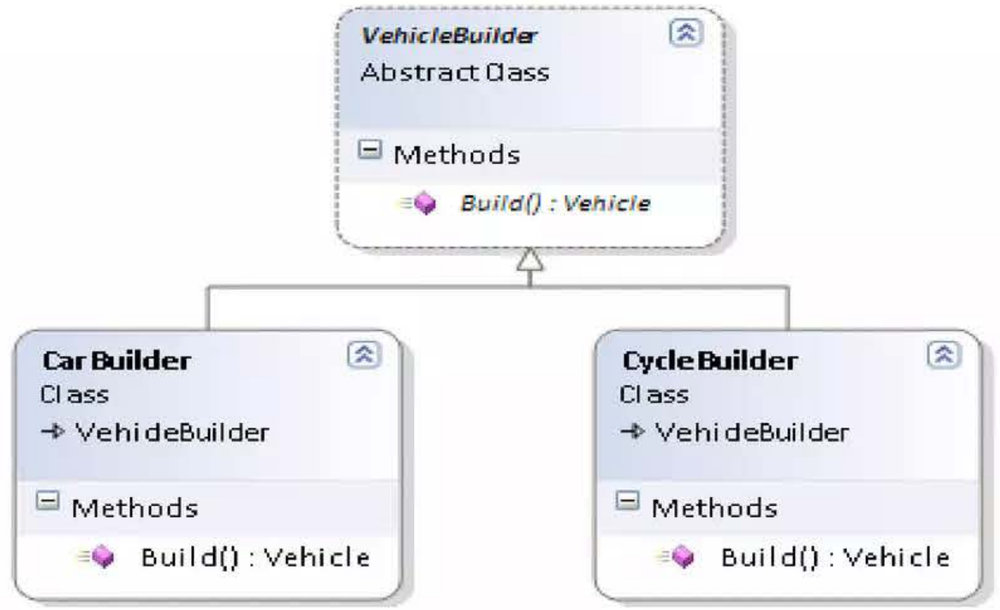



```
UriBuilder builder = new UriBuilder();
builder.Scheme = "https";
builder.Host = "www.hotmail.com";
builder.Port = 243;
builder.Path = "Login/default.aspx";
builder.UserName = "aniruddhac";
builder.Query = "LoginAttempt=2&UseLightUI=true";

string uri = builder.Uri.ToString();
Console.WriteLine(uri);
```



Implement Builder Pattern in .NET (non Generic)



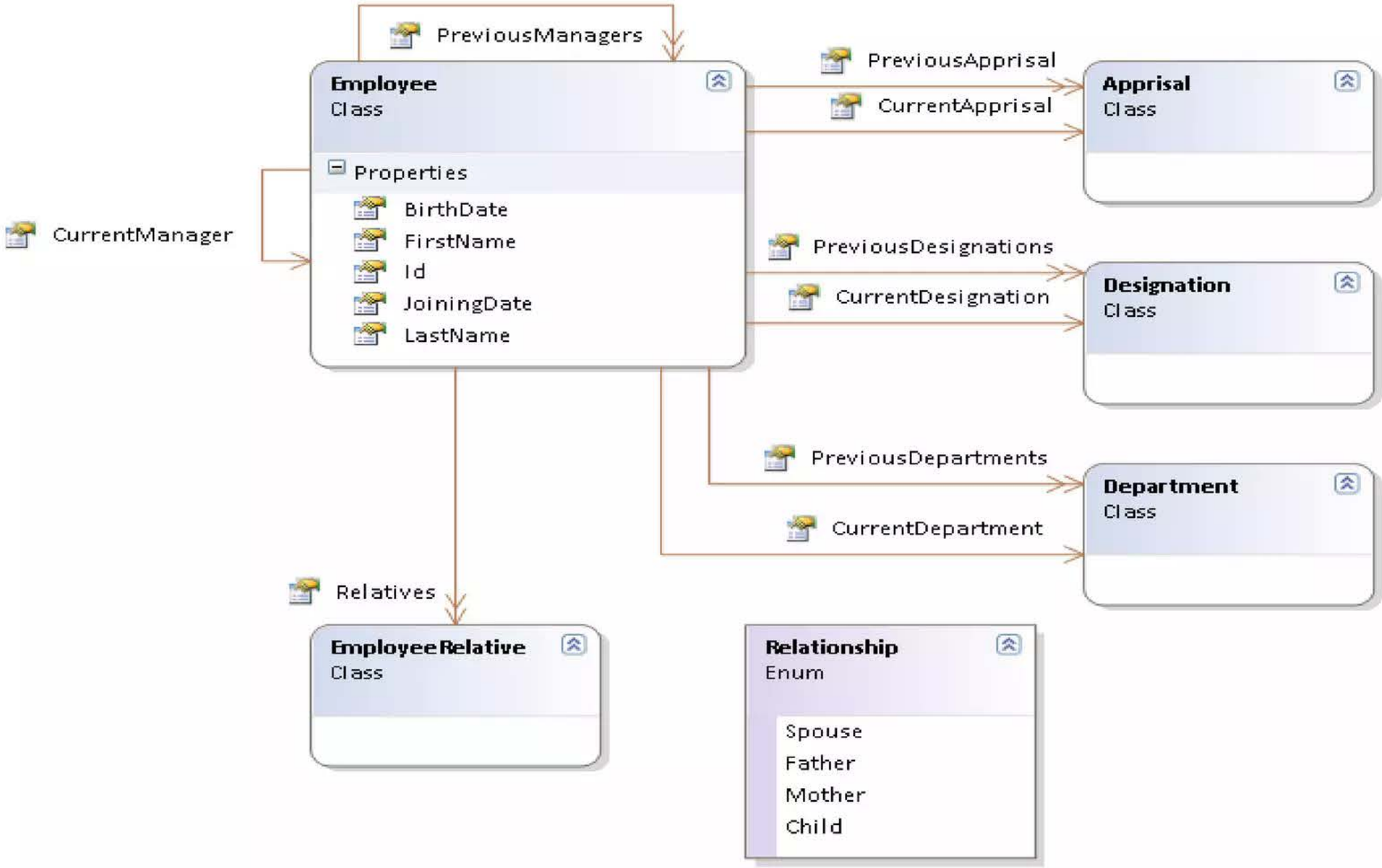



```
static void Main(string[] args)
{
    VehicleBuilder carBuilder = new CarBuilder();
    Car car = carBuilder.Build() as Car;

    VehicleBuilder cycleBuilder = new CycleBuilder();
    Cycle cycle = cycleBuilder.Build() as Cycle;
}
```

Implement Builder Pattern in .NET (Generic)

```
static void Main(string[] args)
{
    VehicleBuilder<Car> carBuilder = new CarBuilder();
    Car car = carBuilder.Build();

    VehicleBuilder<Cycle> cycleBuilder = new CycleBuilder();
    Cycle cycle = cycleBuilder.Build();
}

public abstract class VehicleBuilder<T>
{
    abstract public T Build();
}

public class CarBuilder : VehicleBuilder<Car>
{
    override public Car Build()
    {
        Car car = new Car();
        car.Company = "Maruti"; car.Model = "Alto";
        car.Engine = "900 cc"; car.NoOfDoors = 4;
        car.NoOfWheels = 4;

        return car;
    }
}

public class CycleBuilder : VehicleBuilder<Cycle>
{
    override public Cycle Build()
    {
        Cycle cycle = new Cycle();
        cycle.Company = "Hero"; cycle.Model = "Jet Master";
        cycle.NoOfWheels = 2; cycle.FrameType = "Alumunim";

        return cycle;
    }
}
```

Implement Builder Pattern in .NET

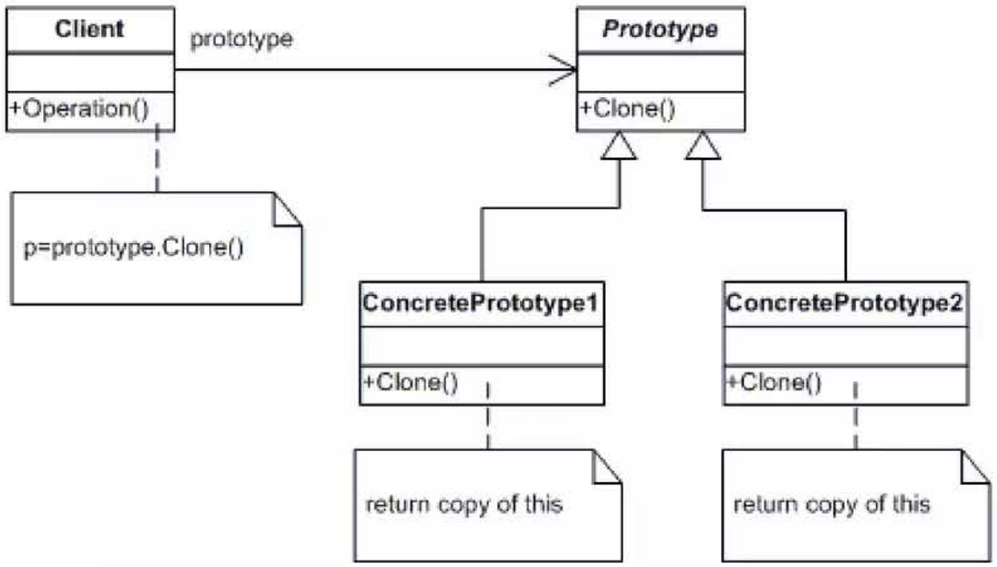


Builder Pattern in .NET

```
public interface Builder<T>
{
    T Build();
}

public class EmployeeBuilder : Builder<Employee>
{
    public Employee Build()
    {
        // Retieve the data from DB and build the employee
        Employee employee = new Employee();

        employee.Relatives = this.GetRelatives(employee.Id);

        return employee;
    }

    private List<EmployeeRelative> GetRelatives(int employeeId)
    {
        return null;
    }
}
```

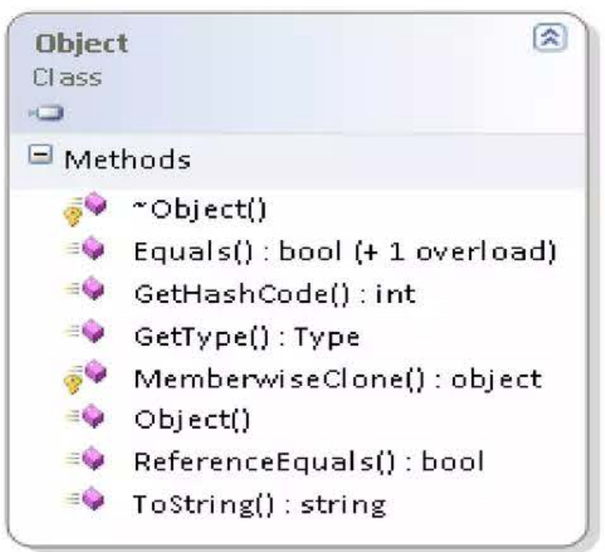
Prototype Design Pattern

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Co-opt one instance of a class for use as a breeder of all future instances.



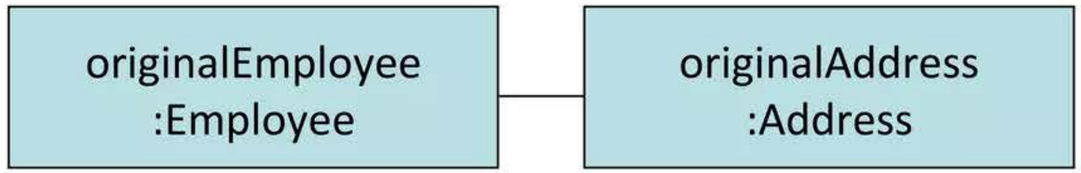
Prototype in .NET BCL

- Prototype is achieved in .NET BCL in two ways –
  - Using Object.MemberwiseClone protected method
    - Performs Shallow Copy of members
    - Shallow Copy is provided out-of the box by .NET BCL
  - Using ICloneable BCL interface
    - Need to be implemented by classes implementing ICloneable
    - Provides a method called Clone
    - Should provide Deep Copy of members

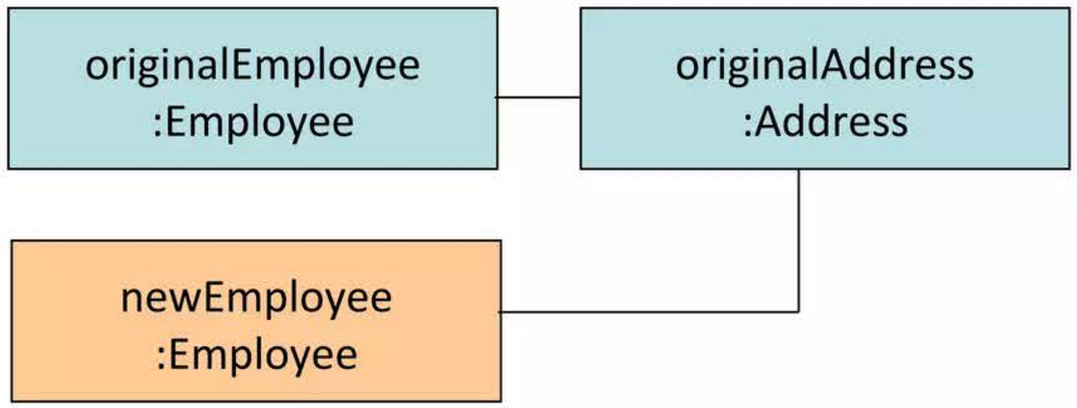


Deep Copy vs. Shallow Copy

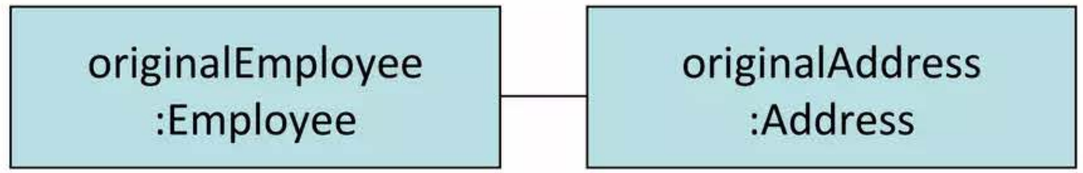
Before



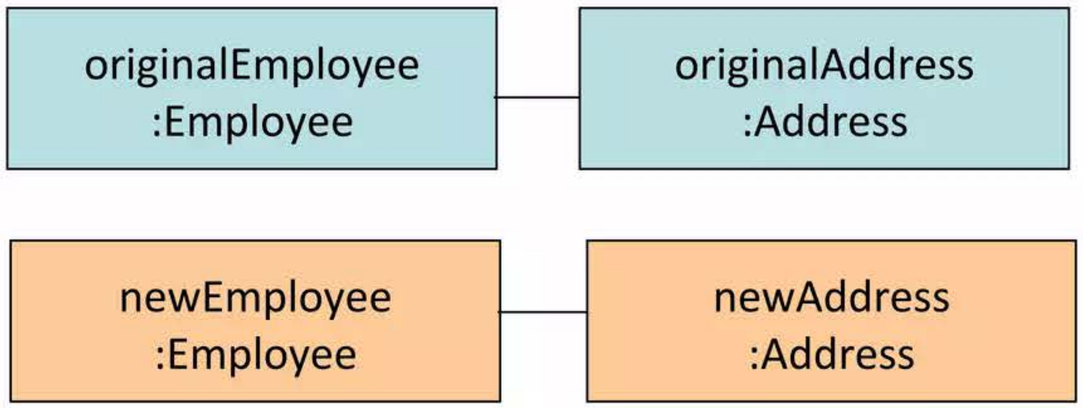
After



Shallow Copy



After



Deep Copy

Implement Prototype in .NET

Non Generic version

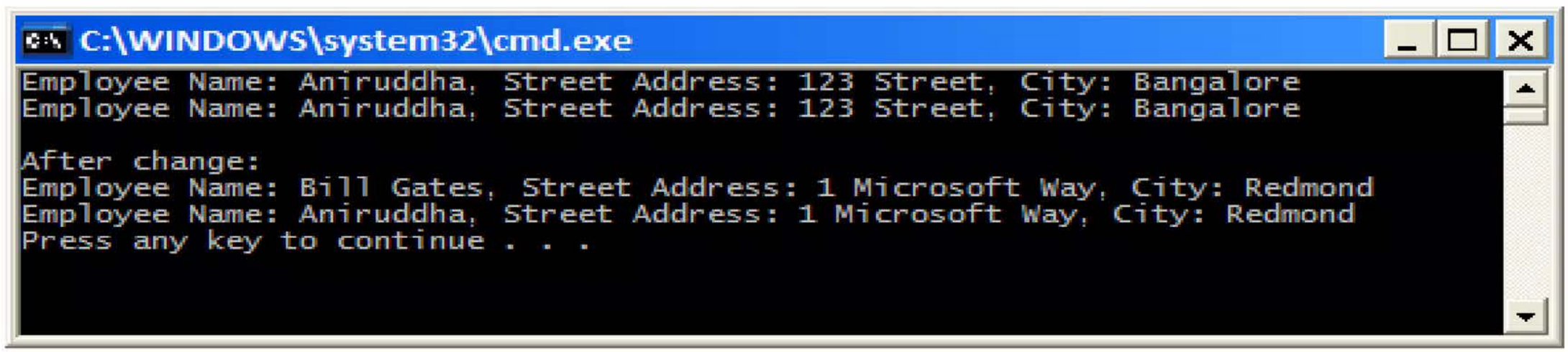
```
[Serializable]
public class Employee: ICloneable
{
    public string Name { get; set; }
    public Address Address { get; set; }

    // Performs a Shallow Clone
    public Employee MemberwiseClone()
    {
        return base.MemberwiseClone() as Employee;
    }

    // Performs a Deep Clone
    public object Clone()
    {
        BinaryFormatter formatter = new BinaryFormatter();

        // Serialize the current object
        MemoryStream stream = new MemoryStream();
        formatter.Serialize(stream, this);

        // Deserialize to create a new object
        stream.Seek(0, SeekOrigin.Begin);
        return formatter.Deserialize(stream);
    }
}

[Serializable]
public class Address
{
    public string StreetAddress { get; set; }
    public string City { get; set; }
}
```

Better Generic version

```
public interface ICloneable<T>
    where T : class
{
    T DeepClone();
}


[Serializable]
public class Employee: ICloneable, ICloneable<Employee>
{
    public string Name { get; set; }
    public Address Address { get; set; }

    // Performs a Shallow Clone
    public Employee MemberwiseClone()...

    // Performs a Deep Clone
    public object Clone()...

    public Employee DeepClone()
    {
        return this.Clone() as Employee;
    }
}
```

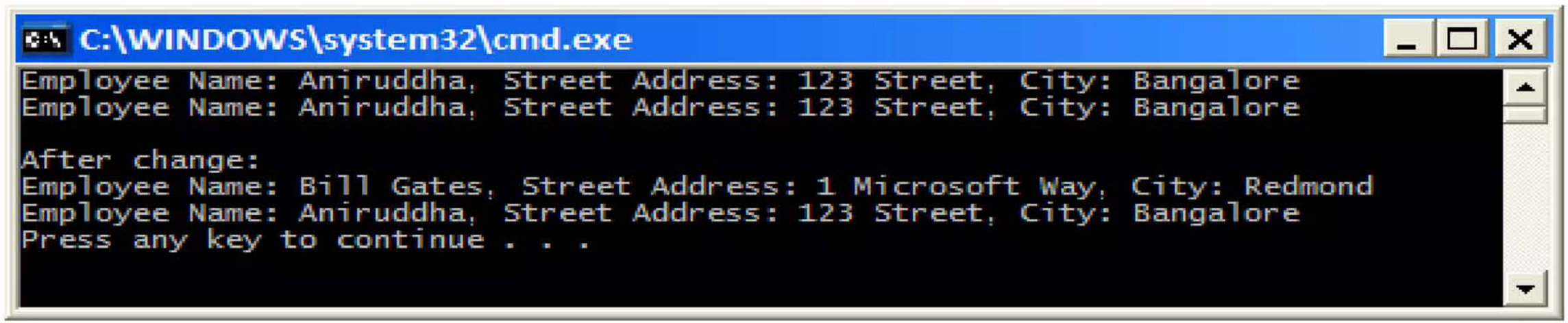
Prototype (Shallow Copy)

```
var originalEmployee = new Employee { Name = "Aniruddha",
    Address = new Address { StreetAddress = "123 Street", City = "Bangalore" } };
Employee newEmployee = originalEmployee.MemberwiseClone();

PrintEmployee(originalEmployee);
PrintEmployee(newEmployee);

// Changing the original employee changes the cloned employee also as it's a Shallow Copy.
originalEmployee.Name = "Bill Gates";
originalEmployee.Address.City = "Redmond";
originalEmployee.Address.StreetAddress = "1 Microsoft Way";

Console.WriteLine("After change:");
PrintEmployee(originalEmployee);
PrintEmployee(newEmployee);
```



Prototype (Deep Copy)

```
var originalEmployee = new Employee { Name = "Aniruddha",
    Address = new Address { StreetAddress = "123 Street", City = "Bangalore" } };
Employee newEmployee = originalEmployee.Clone() as Employee;

PrintEmployee(originalEmployee);
PrintEmployee(newEmployee);

// Changing the original employee changes the cloned employee also as it's a Shallow Copy.
originalEmployee.Name = "Bill Gates";
originalEmployee.Address.City = "Redmond";
originalEmployee.Address.StreetAddress = "1 Microsoft Way";

Console.WriteLine("\nAfter change:");
PrintEmployee(originalEmployee);
PrintEmployee(newEmployee);
```



Dependency Injection (DI)

DI Example - Unity

- Lightweight, extensible dependency injection container
- Supports interception, constructor injection, property injection, and method call injection
- Part of MS Enterprise Library Application Block

Unity Example

```
public interface ILogger
{
    void Log(string message);
}

public class EventLogger : ILogger...

public class FlatFileLogger : ILogger...

public class DatabaseLogger : ILogger...

public class DebugLogger : ILogger...

public class NullLogger : ILogger...
```

```
<configuration>
    <configSections>
        <section name="unity"
type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
                    Microsoft.Practices.Unity.Configuration"/>
    </configSections>

    <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
        <alias alias="ILogger" type="UnityPlay.ILogger, UnityPlay" />
        <namespace name="UnityPlay" />
        <assembly name="UnityPlay" />

        <container>
            <register type="ILogger" name="logger" mapTo="DebugLogger" />
            <register type="ILogger" mapTo="NullLogger" />
        </container>
    </unity>
</configuration>
```

Unity Example

```
// Construct the Unity Container.
// Unity Container is the entry point of the Unity App Block
IUnityContainer container = new UnityContainer();

// Load the unity related config sections from config file
container.LoadConfiguration();

// Get or Resolve default (unnamed) concrete instance of ILogger
// as configured in config file - using Resolve<T> method
ILogger defaultLogger = container.Resolve<ILogger>();
defaultLogger.Log("Fatal error occurred");

// Get or Resolve a named concrete instance of ILogger
// as configured in config file - - using Resolve method
ILogger logger = container.Resolve(typeof(ILogger), "logger") as ILogger;
logger.Log("Fatal error occurred");
```

Singleton
Factory
Method

Builder
Abstract
Factory

Adapter
Composite
Bridge

Proxy
Command
Observer

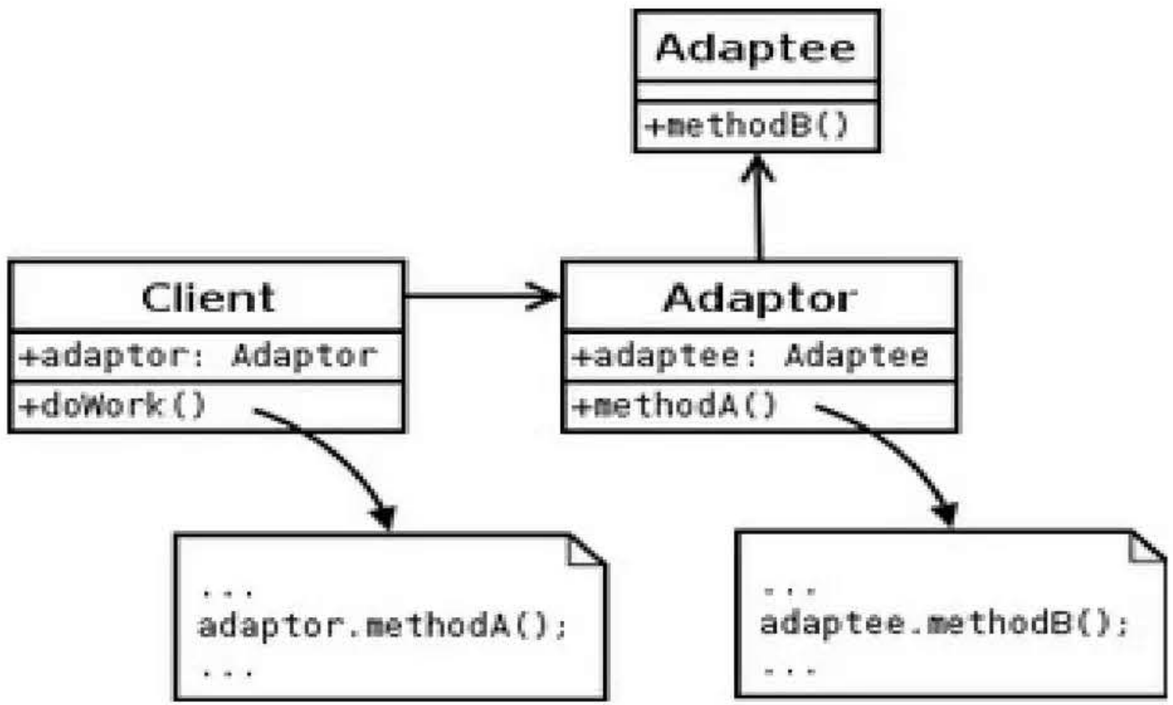
Template
Method
Strategy

Structural Design Patterns

Adapter Design Pattern

- Adapter pattern (also called wrapper pattern or wrapper) translates one interface for a class into a compatible interface.
- Allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface.
- The adapter translates calls to its interface into calls to original interface. Amount of code necessary to do this is typically small.
- The adapter is also responsible for transforming data into appropriate forms.
- Often used while working with existing API/code base and while adapting code for which no source is not available.
- Could be of two types
  - Object Adapter
  - Class Adapter

Adapter Class Diagram



Adapter Pattern in ADO.NET

- Data Adapters adapts 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

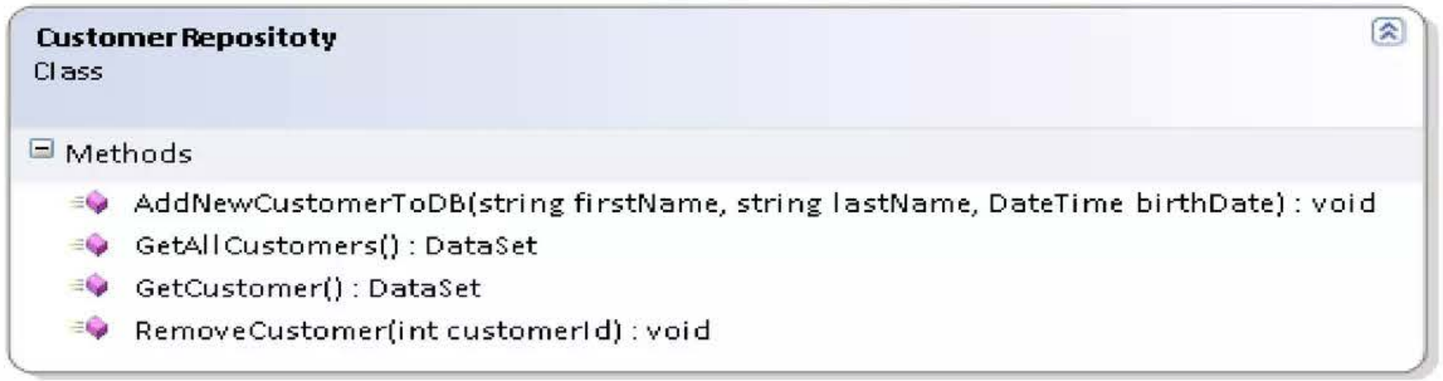
```
string connectionString = "Data Source=.;Initial Catalog=Employee;Integrated Security=true";
SqlDataAdapter adapter;
DataSet ds = new DataSet();

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();
}
```

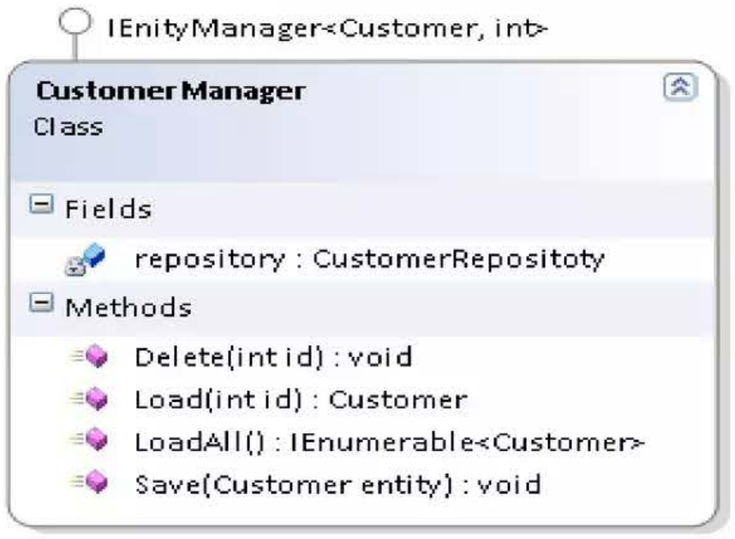
Adapter Pattern in ADO.NET – Cont'd

```
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();
}
```

Implement Adapter Pattern in .NET

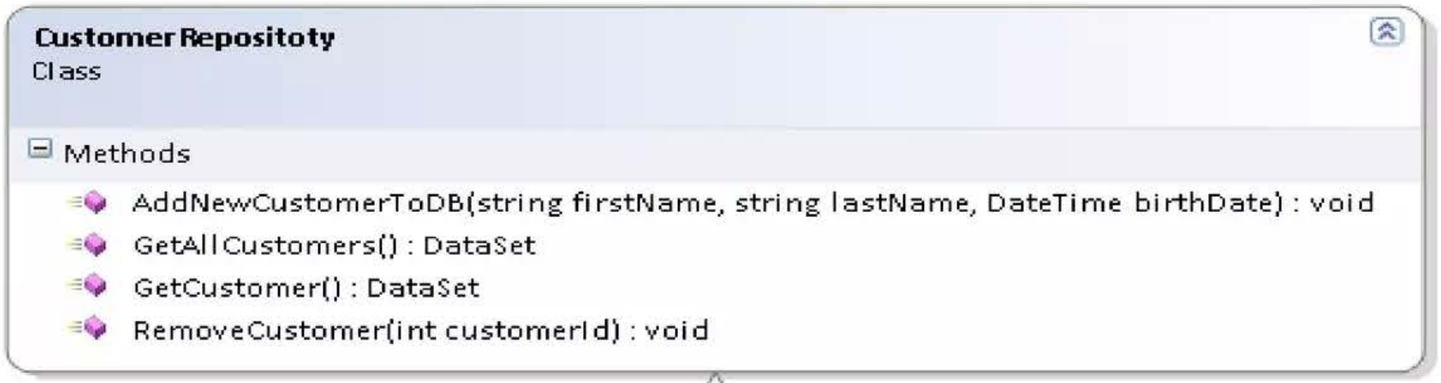


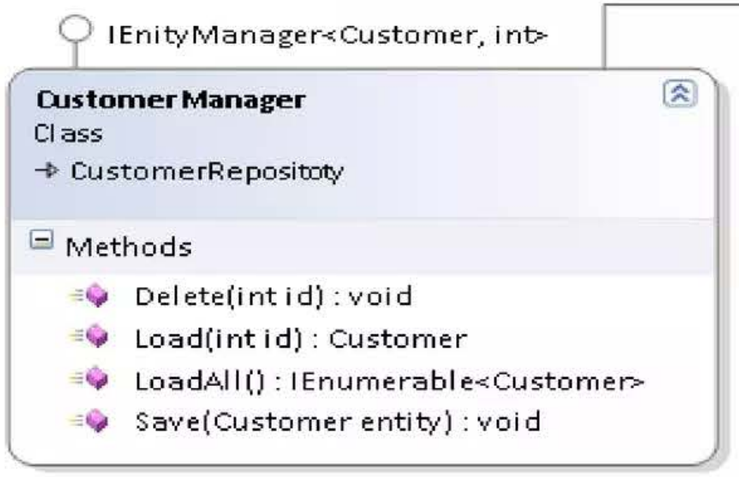


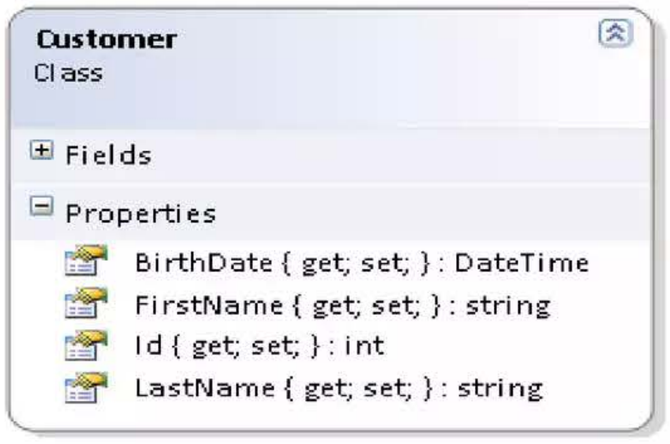
Object Adapter

```
public class CustomerManager : IEnityManager<Customer, int>
{
    CustomerRepositoty repository = new CustomerRepositoty();

    #region IEnityManager<Customer,int> Members

    public IEnumerable<Customer> LoadAll()
    {
        List<Customer> customers = new List<Customer>();
        DataSet customerSet = repository.GetAllCustomers();

        // Convert Dataset to List<Customer>
        return customers;
    }

    public Customer Load(int id)...

    public void Delete(int id)...

    public void Save(Customer entity)...

    #endregion
}
```

Implement Adapter Pattern in .NET



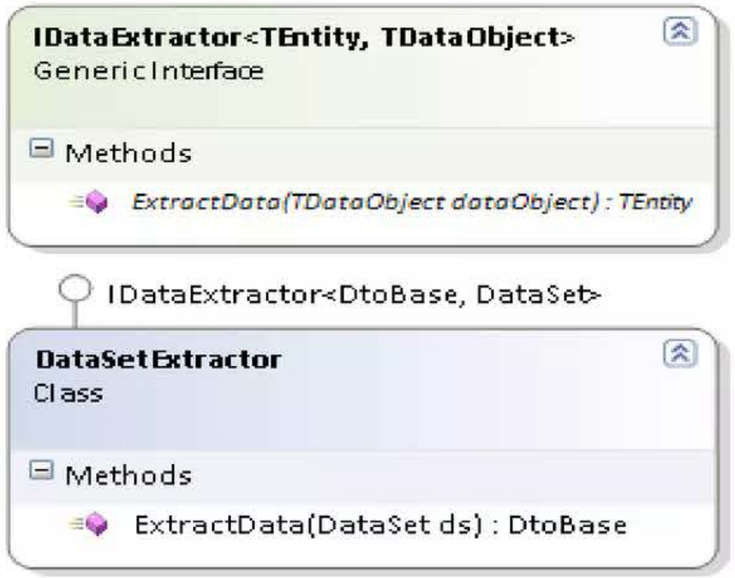




Class Adapter

```
public class CustomerManager : CustomerRepositoty, IEnityManager<Customer, int>
{
    #region IEnityManager<Customer,int> Members

    public IEnumerable<Customer> LoadAll()
    {
        List<Customer> customers = new List<Customer>();
        DataSet customerSet = base.GetAllCustomers();

        // Convert Dataset to List<Customer>
        return customers;
    }

    public Customer Load(int id)...

    public void Delete(int id)...

    public void Save(Customer entity)...

    #endregion
}
```

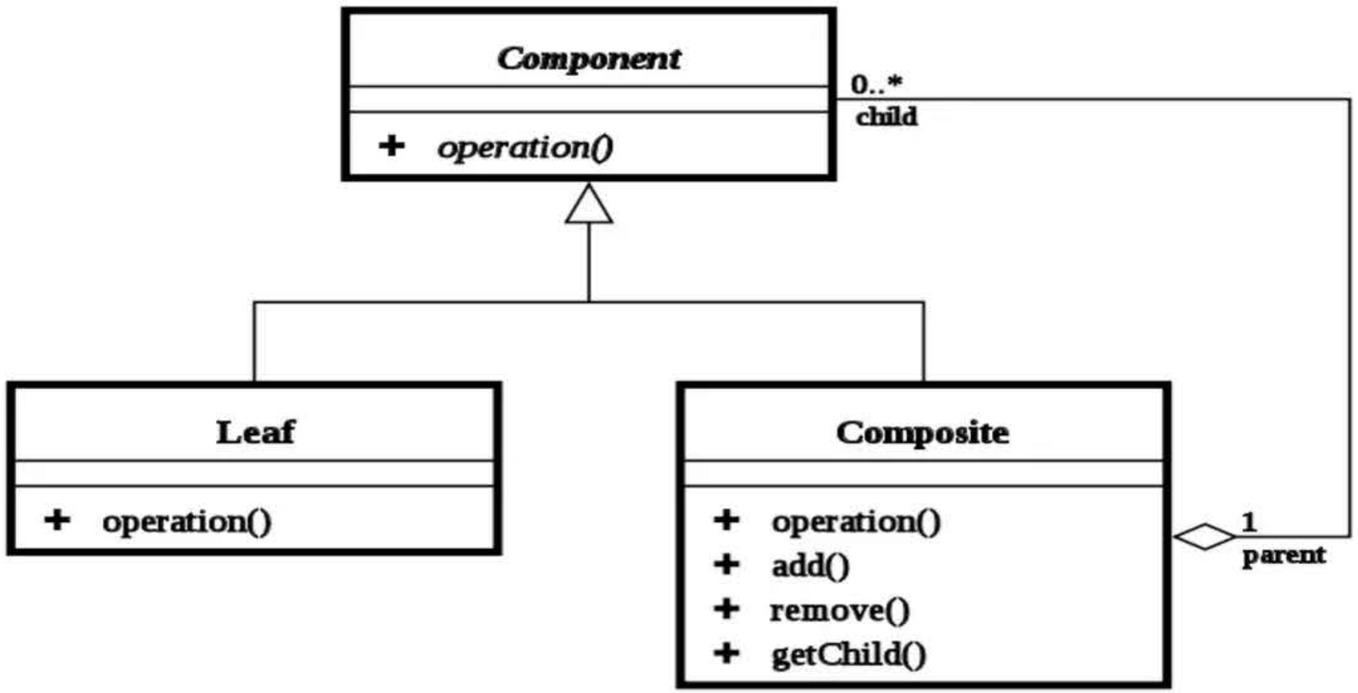
Adapter for extracting data



Composite Design Pattern

- Describes that a group of objects are to be treated in the same way as a single instance of an object. Intent is to "compose" objects into tree structures to represent part-whole hierarchies.
- Implementing the composite pattern lets clients treat individual objects and compositions uniformly
- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive Composition
- “Directories contain entries, each of which could be a directory.”
- 1-to-many “has a” up the “is a” hierarchy

Composite Pattern Class Diagram



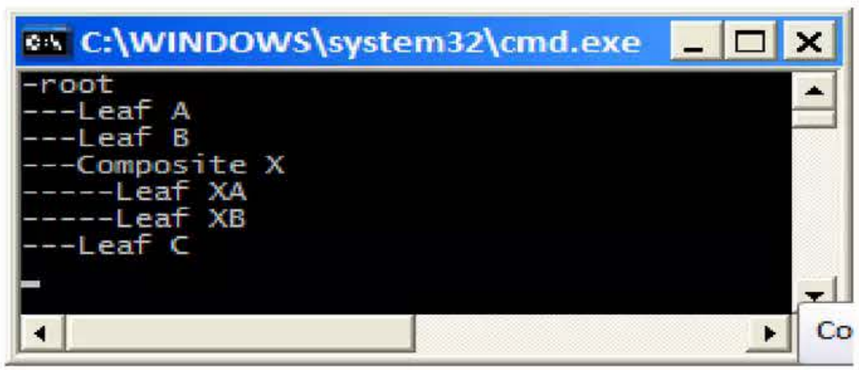
Implement Composite in .NET

```
abstract class Component
{
    protected string name;
    public Component(string name)
    {
        this.name = name;
    }

    public abstract void Add(Component c);
    public abstract void Remove(Component c);
    public abstract void Display(int depth);
}

class Composite : Component
{
    private List<Component> children = new List<Component>();

    public Composite(string name) : base(name) { }

    public override void Add(Component component)
    {
        children.Add(component);
    }

    public override void Remove(Component component)
    {
        children.Remove(component);
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
        // Recursively display child nodes
        foreach (Component component in children)
            component.Display(depth + 2);
    }
}
```

```
class Leaf : Component
{
    public Leaf(string name): base(name)
    { }

    public override void Add(Component c)
    {
        Console.WriteLine("Cannot add to a leaf");
    }

    public override void Remove(Component c)
    {
        Console.WriteLine("Cannot remove from a leaf");
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
    }
}
```

Implement Composite in .NET (Cont'd)

```
// Create a tree structure
Composite root = new Composite("root");
root.Add(new Leaf("Leaf A"));
root.Add(new Leaf("Leaf B"));

Composite comp = new Composite("Composite X");
comp.Add(new Leaf("Leaf XA"));
comp.Add(new Leaf("Leaf XB"));

root.Add(comp);
root.Add(new Leaf("Leaf C"));

// Add and remove a leaf
Leaf leaf = new Leaf("Leaf D");
root.Add(leaf);
root.Remove(leaf);

// Recursively display tree
root.Display(1);
```



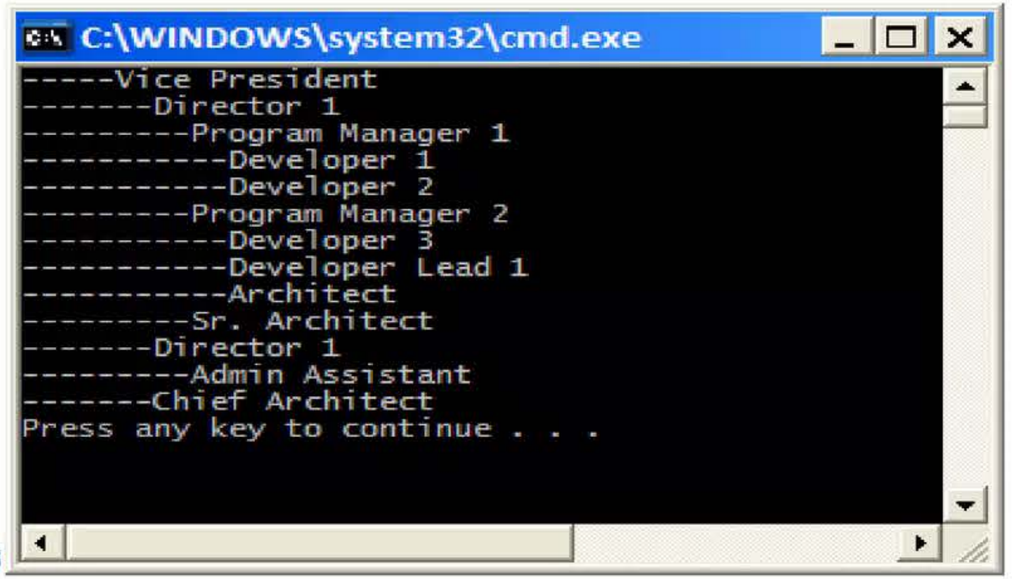
Composite Pattern in .NET – Org Tree

```
// Component
abstract class Employee
{
    protected string name;
    public Employee(string name)
    {
        this.name = name;
    }

    public abstract void Add(Employee employee);
    public abstract void Remove(Employee employee);
    public abstract void Display(int depth);
}

// Composite
class Manager : Employee
{
    private List<Employee> directReports = new List<Employee>();
    public Manager(string name) : base(name) { }

    public override void Add(Employee employee)
    {
        directReports.Add(employee);
    }

    public override void Remove(Employee employee)
    {
        directReports.Remove(employee);
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
        // Recursively display child nodes
        foreach (Employee employee in directReports)
            employee.Display(depth + 2);
    }
}
```

```
// Leaf
class IndividualContributor : Employee
{
    public IndividualContributor(string name)
        : base(name)
    { }

    public override void Add(Employee employee)
    {
        Console.WriteLine
            ("Cannot add employee to a Individual Contributor");
    }

    public override void Remove(Employee employee)
    {
        Console.WriteLine
            ("Cannot remove from a Individual Contributor");
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
    }
}
```

Composite Pattern in .NET – Org Tree

```
// Create an organizational tree
Manager vp = new Manager("Vice President");

Manager dir1 = new Manager("Director 1");
Manager dir2 = new Manager("Director 1");

Manager pm1 = new Manager("Program Manager 1");
IndividualContributor dev1 = new IndividualContributor("Developer 1");
IndividualContributor dev2 = new IndividualContributor("Developer 2");
pm1.Add(dev1);
pm1.Add(dev2);

Manager pm2 = new Manager("Program Manager 2");
IndividualContributor dev3 = new IndividualContributor("Developer 3");
IndividualContributor lead1 = new IndividualContributor("Developer Lead 1");
IndividualContributor architect = new IndividualContributor("Architect");
pm2.Add(dev3);
pm2.Add(lead1);
pm2.Add(architect);

IndividualContributor srArchitect = new IndividualContributor("Sr. Architect");

dir1.Add(pm1);
dir1.Add(pm2);
dir1.Add(srArchitect);

IndividualContributor chiefArchitect = new IndividualContributor("Chief Architect");

IndividualContributor adminAssistant = new IndividualContributor("Admin Assistant");
dir2.Add(adminAssistant);

vp.Add(dir1);
vp.Add(dir2);
vp.Add(chiefArchitect);

vp.Display(5);
```



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.**

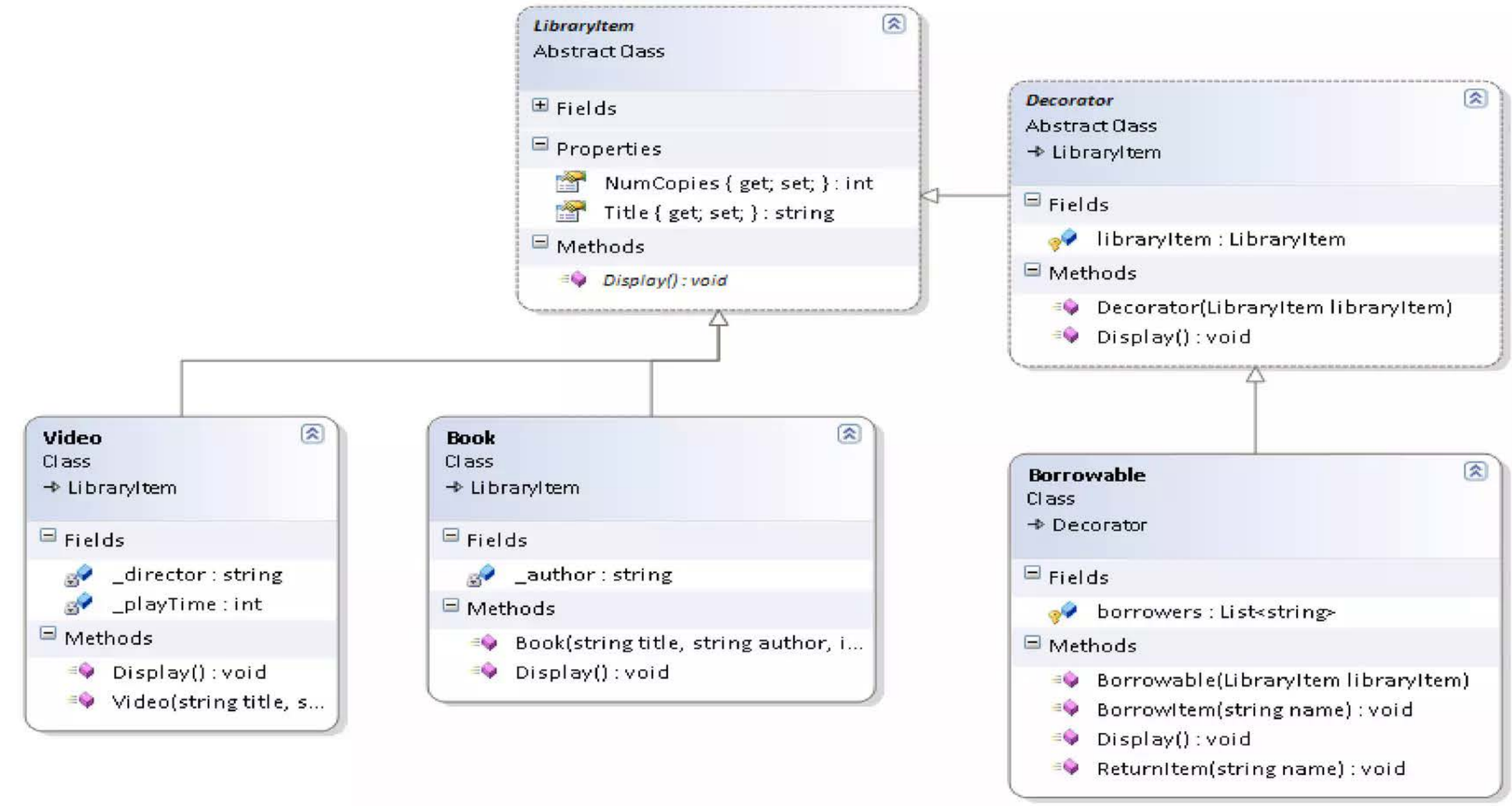
Decorator in .NET BCL - System.IO

```
FileStream fileStream = new FileStream(@"C:\Aniruddha\Test.txt", FileMode.Open, FileAccess.Read);

// Without StreamReader
byte[] content = new byte[1024];

while (fileStream.Read(content, 0, content.Length) > 0)
{
    Console.WriteLine(ASCIIEncoding.ASCII.GetString(content));
}

// Using StreamReader (Decorator Pattern)
StreamReader reader = new StreamReader(fileStream);
Console.WriteLine(reader.ReadToEnd());
```

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>
```

Implementing Decorator in .NET



Implementing Decorator in .NET (Cont'd)

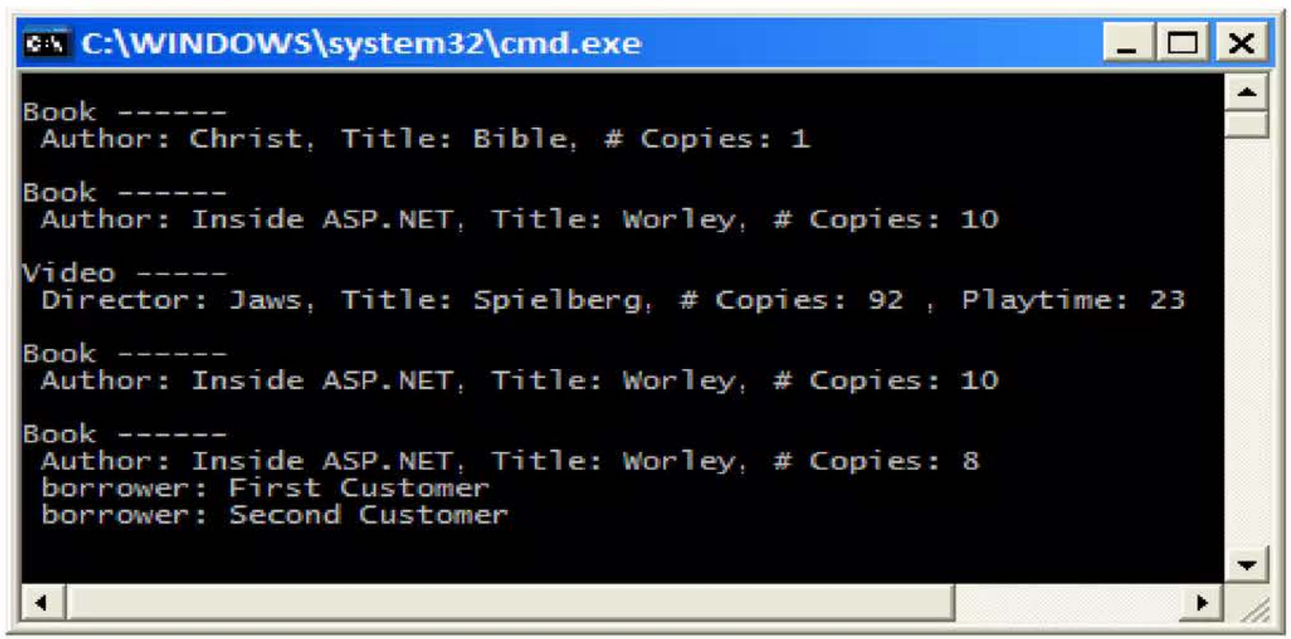
```
abstract class LibraryItem
{
    private int _numCopies;
    private string _title;

    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();
    }
}
```

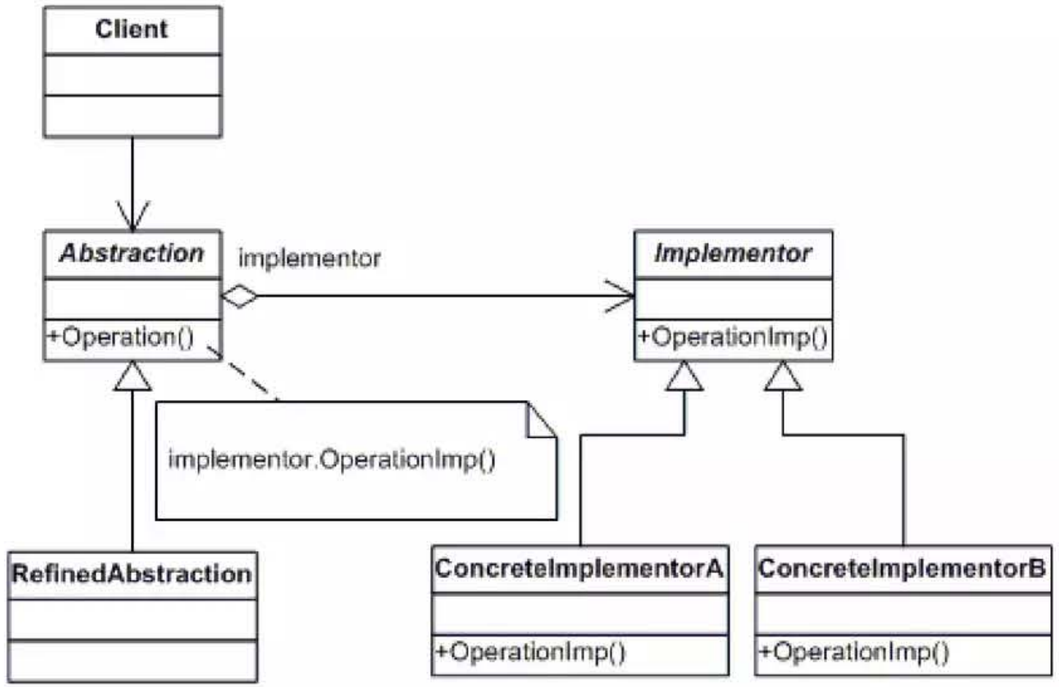
```
class Borrowable : Decorator
{
    protected List<string> borrowers = new List<string>();
    public Borrowable(LibraryItem libraryItem)
        : base(libraryItem)
    { }

    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();
```

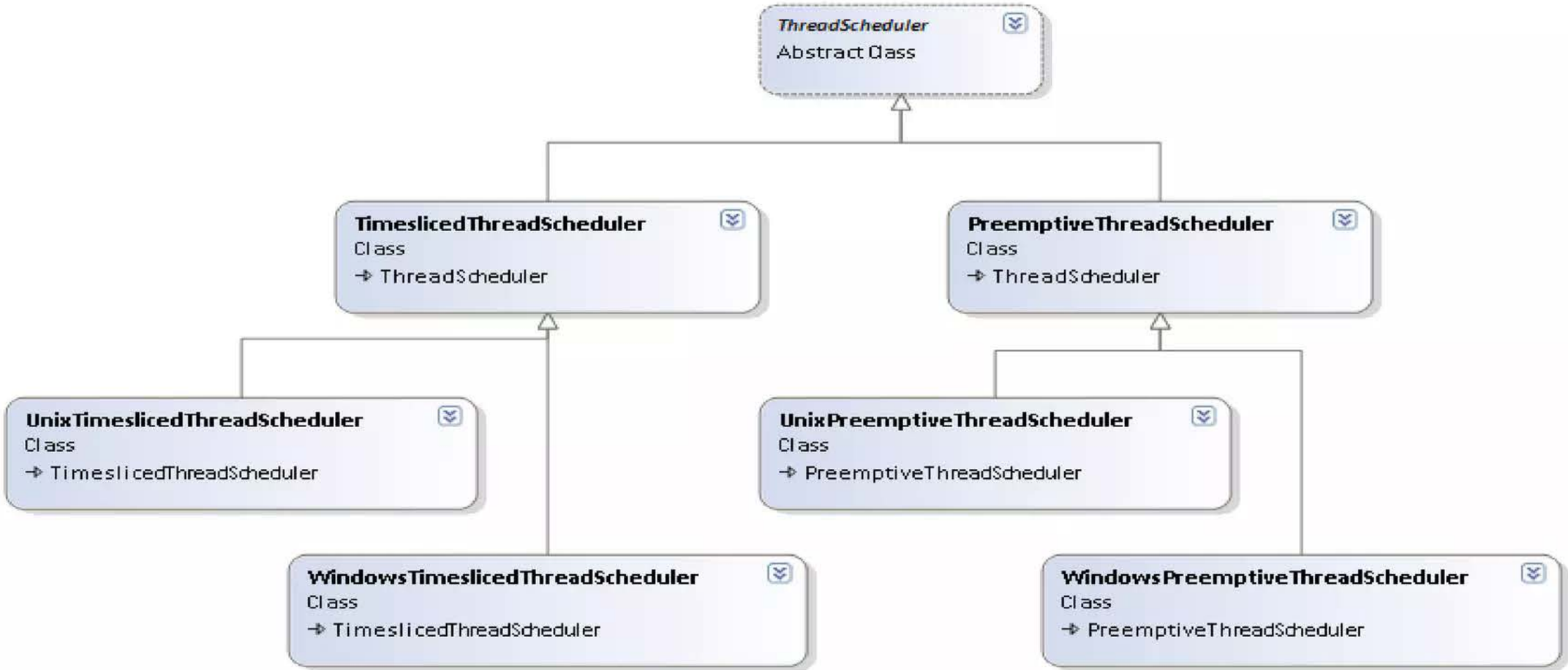


Bridge

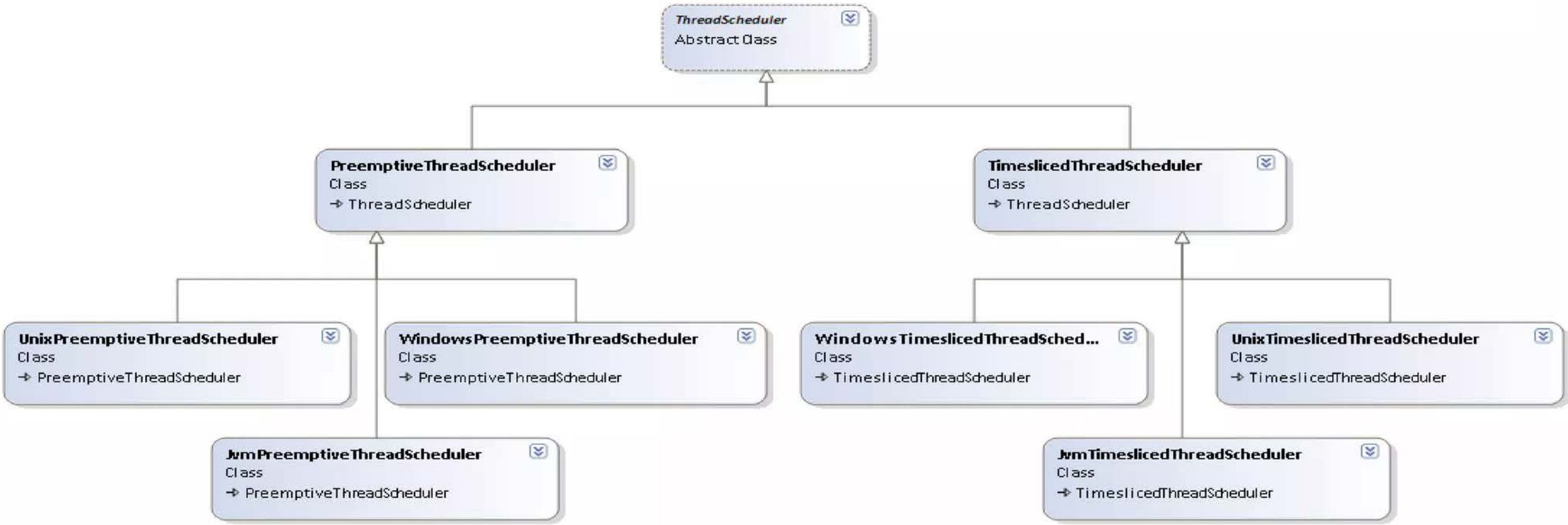
- Decouple an abstraction from its implementation so that the two can vary independently.
- Based on the principle “Prefer Composition (and Association, Aggregation) over Inheritance”
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy
- Beyond encapsulation, to insulation



Bridge Pattern – Thread Scheduler

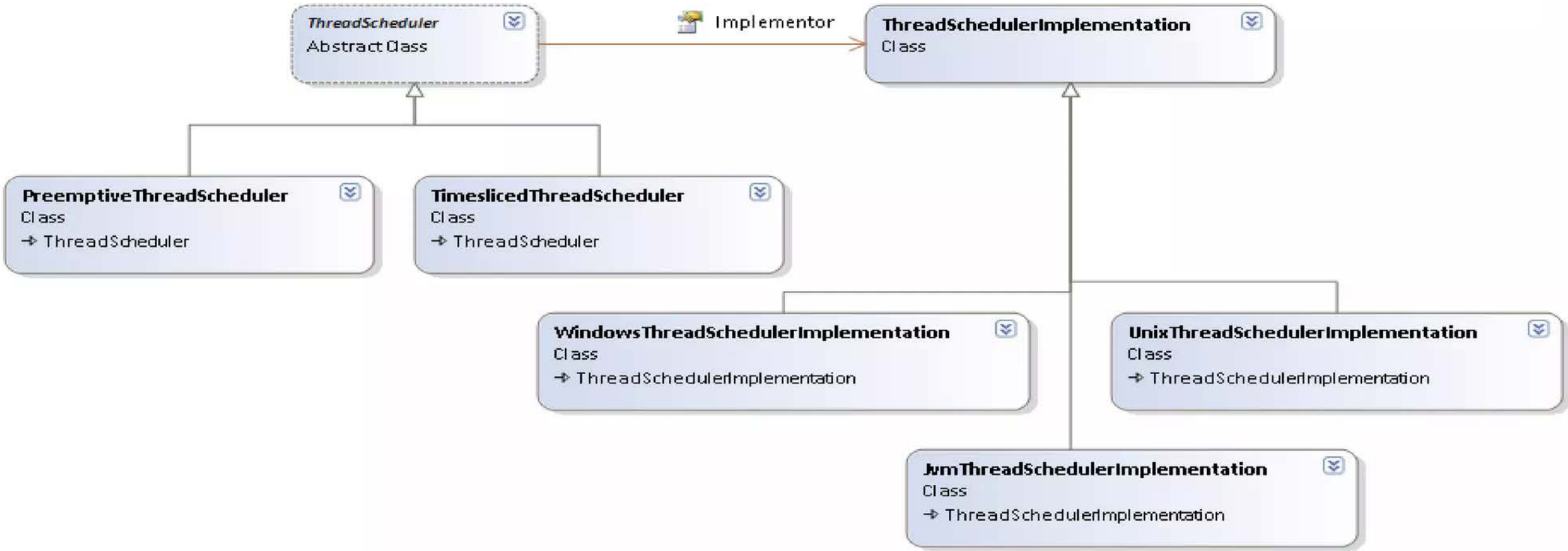


Bridge Pattern – Thread Scheduler (Cont'd)



- A new type of Platform (JVM) is added.
- What happens if another type of scheduler is added and No of Platforms increase to four from three
- No of classes increases exponentially - Inheritance Hierarchy become complex, bloated and difficult to maintain.

Bridge Example



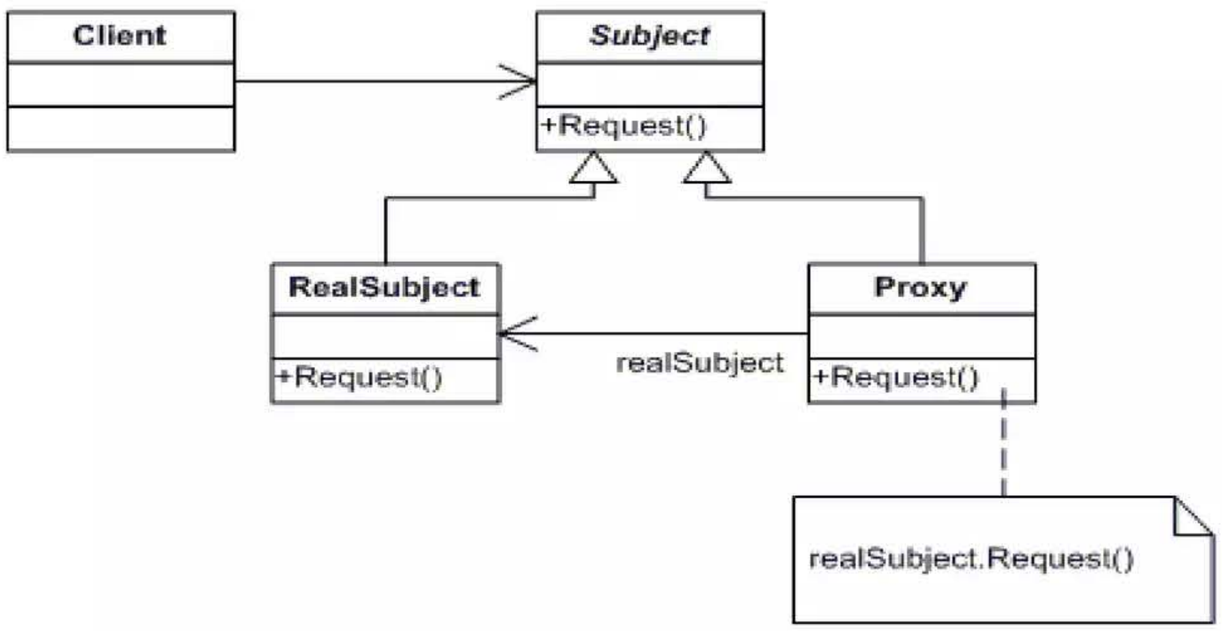
- Decompose the component's interface and implementation into orthogonal class hierarchies.
- Abstract/Interface class contains a pointer to abstract implementation class. This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class.
- Client interacts with interface class - "delegates" all requests to implementation class.

Bridge

```
// Construct a Windows Preemptive Thread Scheduler
ThreadScheduler winPreemtiveTS = new PreemptiveThreadScheduler();
winPreemtiveTS.Implementor = new WindowsThreadSchedulerImplementation();

// Construct a JVM Timesliced Time Scheduler
ThreadScheduler jvmTimeslicedTS = new TimeslicedThreadScheduler();
jvmTimeslicedTS.Implementor = new JvmThreadSchedulerImplementation();
```

Proxy Design Pattern

- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.



Proxy in .NET (WCF)

```
// Service
[ServiceContract]
public interface ICalculatorService
{
    [OperationContract]
    int Add(int i, int j);

    [OperationContract]
    int Add(int i, int j, int k);

    [OperationContract]
    int Subtract(int i, int j);
    [OperationContract]
    int Multiply(int i, int j);
    [OperationContract]
    int Divide(int i, int j);
}
```

Contract

```
public class Calculator : ICalculatorService
{
    public int Add(int i, int j)...
    public int Add(int i, int j, int k)
    {
        return i + j + k;
    }

    public int Subtract(int i, int j)...
    public int Multiply(int i, int j)...
    public int Divide(int i, int j)...
}
```

Contract Implementation

```
ServiceHost host = new ServiceHost(typeof(Calculator), new Uri[] { });
host.Open();
Console.WriteLine("Hosted ... ");
```

Host

Proxy in .NET (WCF) – Cont'd

```
// Client/Proxy
public class CalculatorServiceClient : ClientBase<ICalculatorService>, ICalculatorService
{
    public CalculatorServiceClient() : base("ep") { }

    public int Add(int i, int j)...

    public int Add(int i, int j, int k)
    {
        return base.Channel.Add(i, j, k);
    }

    public int Subtract(int i, int j)...
    public int Multiply(int i, int j)...
    public int Divide(int i, int j)...
}
```

Proxy

```
CalculatorServiceClient client = new CalculatorServiceClient();

Console.WriteLine(client.Add(10, 20).ToString());
Console.WriteLine(client.Add(10, 20, 40).ToString());

Console.ReadLine();
```

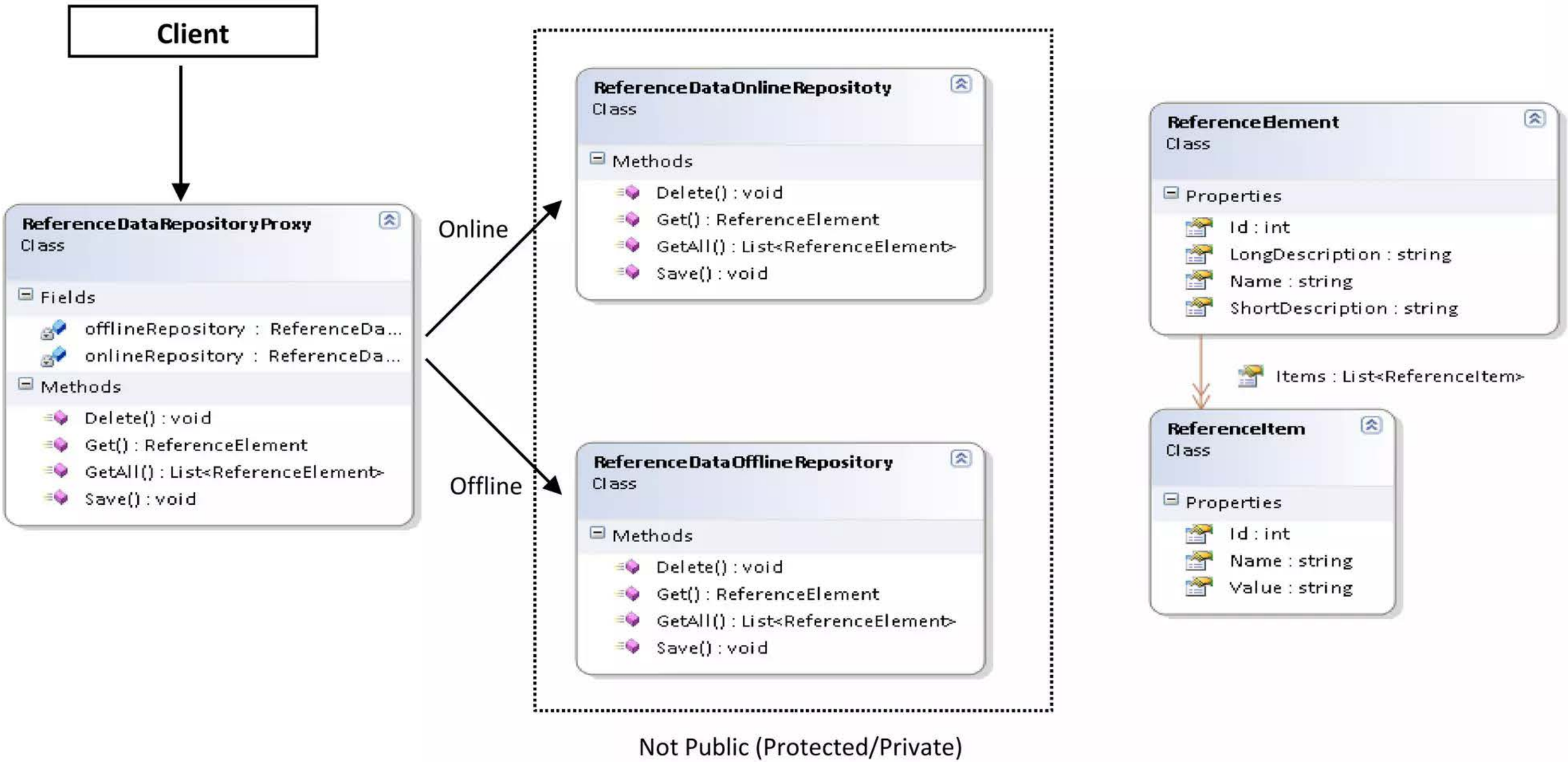
Client

Cannot have two operations in the same contract with the same name, methods Add and Add in type ConsoleApplication1.ICalculatorService violate this rule. You can change the name of one of the operations by changing the method name or by using the Name property of OperationContractAttribute.

Solution

```
// Service
[ServiceContract]
public interface ICalculatorService
{
    [OperationContract(Name="AddTwo")]
    int Add(int i, int j);

    [OperationContract(Name = "AddThree")]
    int Add(int i, int j, int k);

    [OperationContract]
    int Subtract(int i, int j);
    [OperationContract]
    int Multiply(int i, int j);
    [OperationContract]
    int Divide(int i, int j);
}
```

How to implement Proxy in .NET



Singleton
Factory
Method

Builder
Abstract
Factory

Adapter
Composite
Bridge

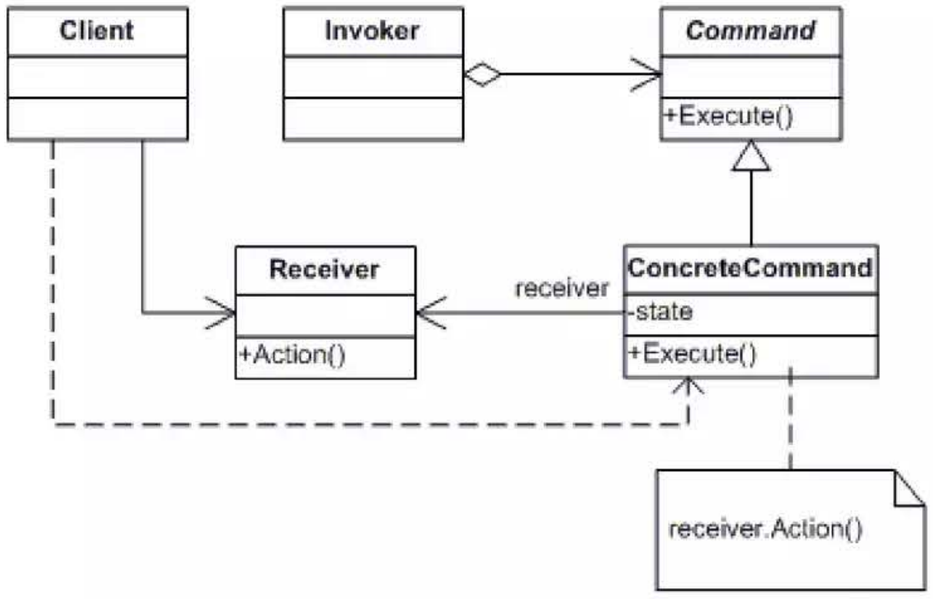
Proxy
Command
Observer

Template
Method
Strategy

Behavioral Design Patterns

Command Design Pattern

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Design Pattern in which an object is used to represent and encapsulate all infos needed to call a method later - info include method name, object that owns the method and values of method parameters.
- Promote "invocation of a method on an object" to full object status
- An object-oriented callback



Observer Design Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
- The “View” part of Model-View-Controller.

Observer design pattern in .NET

- **delegates** and **events** provides a powerful means of implementing the Observer pattern.
- As delegates and events are first class members of CLR, the foundation of this pattern is incorporated into core of .NET Framework.
- FCL makes extensive use of Observer pattern throughout its structure.

Implement an Event

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

    private int salary;
    public int Salary...

    public void RaiseSalary(int raiseSalaryBy)
    {
        salary += raiseSalaryBy;

        OnSalaryRaised(new EventArgs());
    }

    public event EventHandler SalaryRaisedEvent;

    protected virtual void OnSalaryRaised(EventArgs e)
    {
        if (SalaryRaisedEvent != null)
            SalaryRaisedEvent(this, e);
    }
}
```

3. Determine when to raise the event in the class. Call OnEventName to raise the event.

1. Define a public event member in class. Set the type of event member to **System.EventHandler** delegate.

2. Provide a protected method in the class that raises the event. Name the method OnEventName. Raise the event within the method

```
Employee employee = new Employee();
employee.Name = "Bill G";
employee.Salary = 100;
employee.SalaryRaisedEvent += new EventHandler(employee_SalaryRaisedEvent);

employee.RaiseSalary(10);

employee.SalaryRaisedEvent -= new EventHandler(employee_SalaryRaisedEvent);
employee.RaiseSalary(10);

void employee_SalaryRaisedEvent(object sender, EventArgs e)
{
    MessageBox.Show("SalaryRaisedEvent called");
}
```

Client can **register** for the events they are interested.

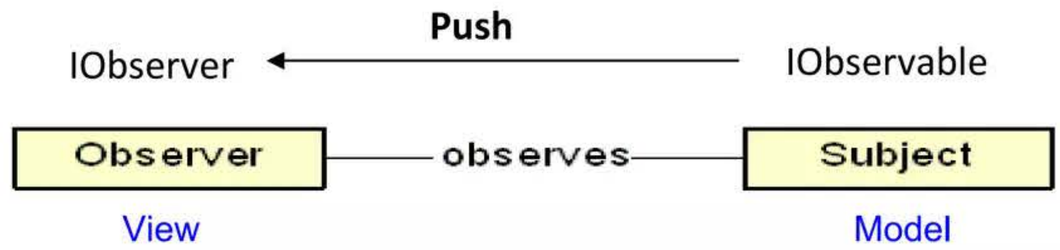
Client can **un-register** for the events they have already registered.

Implement an Event with Event specific data

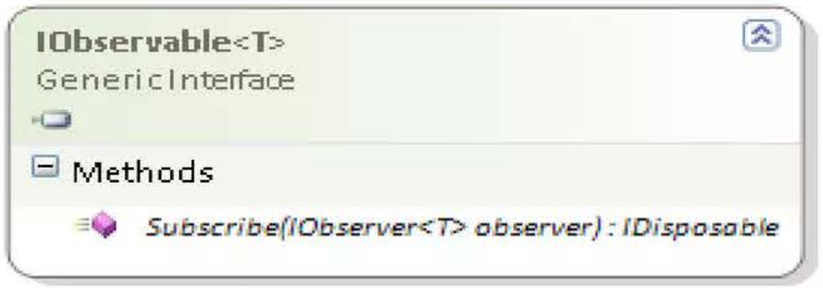
```
public class Employee
{
    private string name;
    public string Name...

    private int salary;
    public int Salary...

    public void RaiseSalary(int raiseSalaryBy)
    {
        salary += raiseSalaryBy;

        SalaryRaiseEventArgs e = new SalaryRaiseEventArgs();
        e.SalaryRaiseBy = raiseSalaryBy;
        OnSalaryRaised(e);
    }

    public event SalaryRaiseEventHandler SalaryRaisedEvent;

    protected virtual void OnSalaryRaised(SalaryRaiseEventArgs e)
    {
        if (SalaryRaisedEvent != null)
            SalaryRaisedEvent(this, e);
    }
}

public delegate void SalaryRaiseEventHandler(object sender, SalaryRaiseEventArgs e);

public class SalaryRaiseEventArgs : EventArgs
{
    private int salaryRaiseBy;
    public int SalaryRaiseBy...
}
```

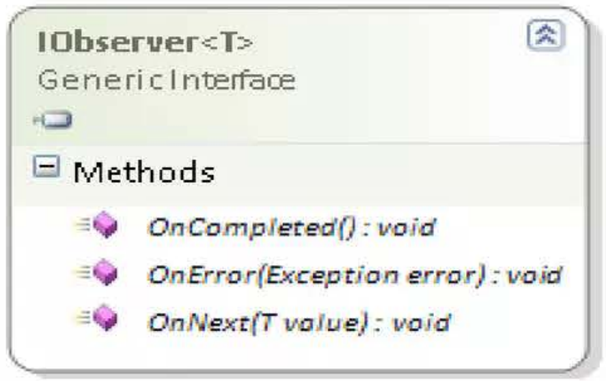
IObserver<T> and IObservable<T> in .NET 4

- .NET 4.0 introduces two new interfaces in BCL that implements Observer pattern - are part of larger Reactive/Rx framework
- The IObserver<T> and IObservable<T> interfaces provide a generalized mechanism for push-based notification
  - IObservable<T> interface represents the class that sends notifications (the provider)
  - IObserver<T> interface represents the class that receives them (the observer)



IObserver<T> and IObservable<T>



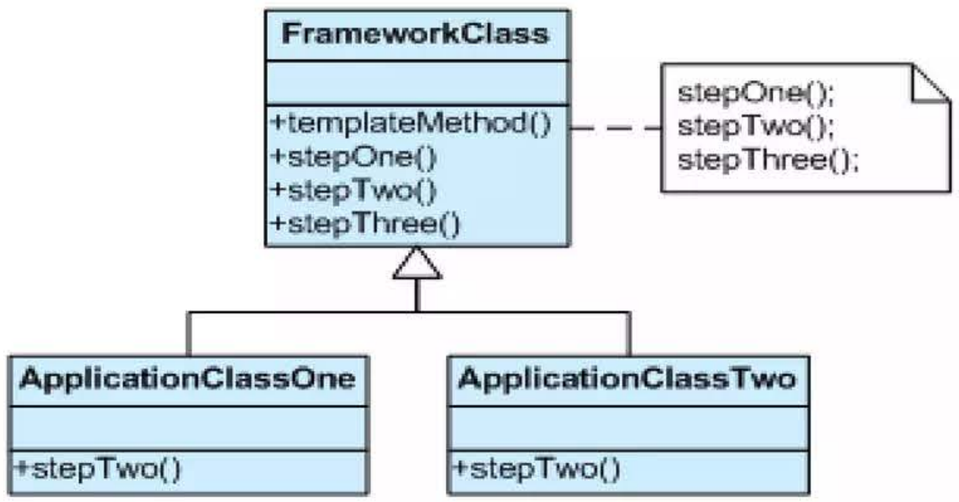



```
namespace System
{
    ...public interface IObservable<out T>
    {
        ...IDisposable Subscribe(IObserver<T> observer);
    }
}
```

```
namespace System
{
    ...public interface IObserver<in T>
    {
        ...void OnCompleted();
        ...void OnError(Exception error);
        ...void OnNext(T value);
    }
}
```

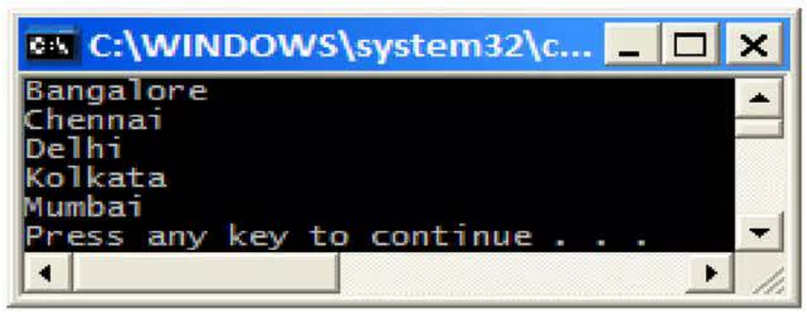
Template Method Pattern

- Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing algorithm's structure.
- Base class declares algorithm 'placeholders', and derived classes implement the placeholders.
- Has similarity with Strategy pattern.



Template Method Pattern in .NET BCL (Sorting)

```
string[] cities = { "Kolkata", "Bangalore", "Delhi", "Chennai", "Mumbai" };

Array.Sort(cities);

foreach (string city in cities)
    Console.WriteLine(city);
```



Since String class implements IComparable Array.Sort (and Sort method of all collection classes including Generic and non-Generic collection classes) uses string's IComparable implementation to sort the array (String.CompareTo is used).

```
namespace System
{
    ...public sealed class String : IComparable, ICloneable, IConvertible, IComparable<string>,
IEnumerable<char>, IEnumerable, IEquatable<string>
    {
        ...public static readonly string Empty;
```

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

```
public class Employee
{
    private string firstName;
    public string FirstName...

    private string lastName;
    public string LastName...

    private double salary;
    public double Salary...

    private DateTime birthDate;
    public DateTime BirthDate...

    public Employee(string firstName, string lastName, double salary, DateTime birthDate)...

}
```

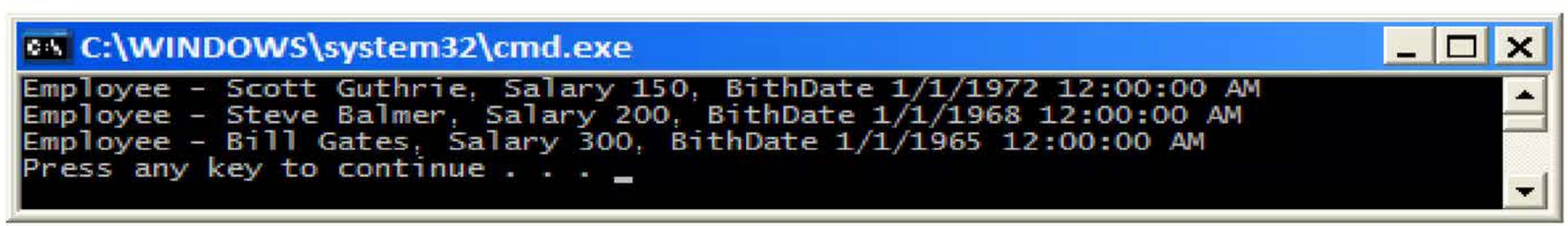
```
Employee emp1 = new Employee("Bill", "Gates", 300, DateTime.Parse("1/1/1965"));
Employee emp2 = new Employee("Scott", "Guthrie", 150, DateTime.Parse("1/1/1972"));
Employee emp3 = new Employee("Steve", "Balmer", 200, DateTime.Parse("1/1/1968"));

Employee[] employees = { emp1, emp2, emp3 };

Array.Sort(employees);

foreach (Employee employee in employees)
    Console.WriteLine("Employee - {0} {1}, Salary {2}, BithDate {3}", employee.FirstName,
        employee.LastName, employee.Salary, employee.BirthDate);
```

```
C:\WINDOWS\system32\cmd.exe

Unhandled Exception: System.InvalidOperationException: Failed to compare two ele
ments in the array. ---> System.ArgumentException: At least one object must impl
ement IComparable.
   at System.Collections.Comparer.Compare(Object a, Object b)
   at System.Collections.Generic.ObjectComparer`1.Compare(T x, T y)
```

Template Method in .NET BCL – IComparable

```
public class Employee : IComparable
{
    private string firstName;
    public string FirstName...

    private string lastName;
    public string LastName...

    private double salary;
    public double Salary...

    private DateTime birthDate;
    public DateTime BirthDate...

    public Employee(string firstName, string lastName, double salary, DateTime birthDate)...

    #region IComparable Members
    public int CompareTo(object obj)
    {
        Employee otherEmployee = obj as Employee;

        if (otherEmployee == null)
            throw new ArgumentException("object is not an Employee");

        return this.Salary.CompareTo(otherEmployee.Salary);
    }
    #endregion
}
```

Employee class now implements IComparable interface which has a single method – int CompareTo(object)



Template Method in .NET BCL – IComparable<T>

```
public class Employee : IComparable, IComparable<Employee>
{
    private string firstName;
    public string FirstName...

    private string lastName;
    public string LastName...

    private double salary;
    public double Salary...

    private DateTime birthDate;
    public DateTime BirthDate...

    public Employee(string firstName, string lastName, double salary, DateTime birthDate)...

    #region IComparable Members
    public int CompareTo(object obj)
    {
        Employee otherEmployee = obj as Employee;

        if (otherEmployee == null)
            throw new ArgumentException("object is not an Employee");

        return this.CompareTo(otherEmployee);
    }
    #endregion

    #region IComparable<Employee> Members
    public int CompareTo(Employee other)
    {
        return this.Salary.CompareTo(other.Salary);
    }
    #endregion
}
```

Employee class implements two interface –

- IComparable (non Generic) which has a single method – int CompareTo(object)
- IComparable<T> (Generic) which has another overload of CompareTo – int CompareTo(T)

- Generic version of CompareTo is used by CLR if both non generic and generic versions are present.
- Always prefer implementing IComparable<T> for better performance (no boxing, unboxing overhead)

Template Method in .NET BCL - IComparer

```
public class EmployeeBirthDateComparer : IComparer
{
    #region IComparer Members
    public int Compare(object x, object y)
    {
        Employee emp1 = x as Employee;
        if (emp1 == null)
            throw new ArgumentException("object is not an Employee");

        Employee emp2 = y as Employee;
        if (emp2 == null)
            throw new ArgumentException("object is not an Employee");

        return emp1.BirthDate.CompareTo(emp2.BirthDate);
    }
    #endregion
}
```

```
Employee[] employees = { emp1, emp2, emp3 };

IComparer employeeBirthDateComparer = new EmployeeBirthDateComparer();
Array.Sort(employees, employeeBirthDateComparer);

foreach (Employee employee in employees)
    Console.WriteLine("Employee - {0} {1}, Salary {2}, BithDate {3}", employee.FirstName,
        employee.LastName, employee.Salary, employee.BirthDate);
```

```
C:\WINDOWS\system32\cmd.exe
Employee - Bill Gates, Salary 300, BithDate 1/1/1965 12:00:00 AM
Employee - Steve Balmer, Salary 200, BithDate 1/1/1968 12:00:00 AM
Employee - Scott Guthrie, Salary 150, BithDate 1/1/1972 12:00:00 AM
Press any key to continue . . .
```

Template Method in .NET BCL - IComparer<T>

```
public class EmployeeBirthDateComparer : IComparer, IComparer<Employee>
{
    #region IComparer Members
    public int Compare(object x, object y)
    {
        Employee emp1 = x as Employee;
        if (emp1 == null)
            throw new ArgumentException("object is not an Employee");

        Employee emp2 = y as Employee;
        if (emp2 == null)
            throw new ArgumentException("object is not an Employee");

        return this.Compare(emp1, emp2);
    }
    #endregion

    #region IComparer<Employee> Members
    public int Compare(Employee x, Employee y)
    {
        return x.BirthDate.CompareTo(y.BirthDate);
    }
    #endregion
}
```

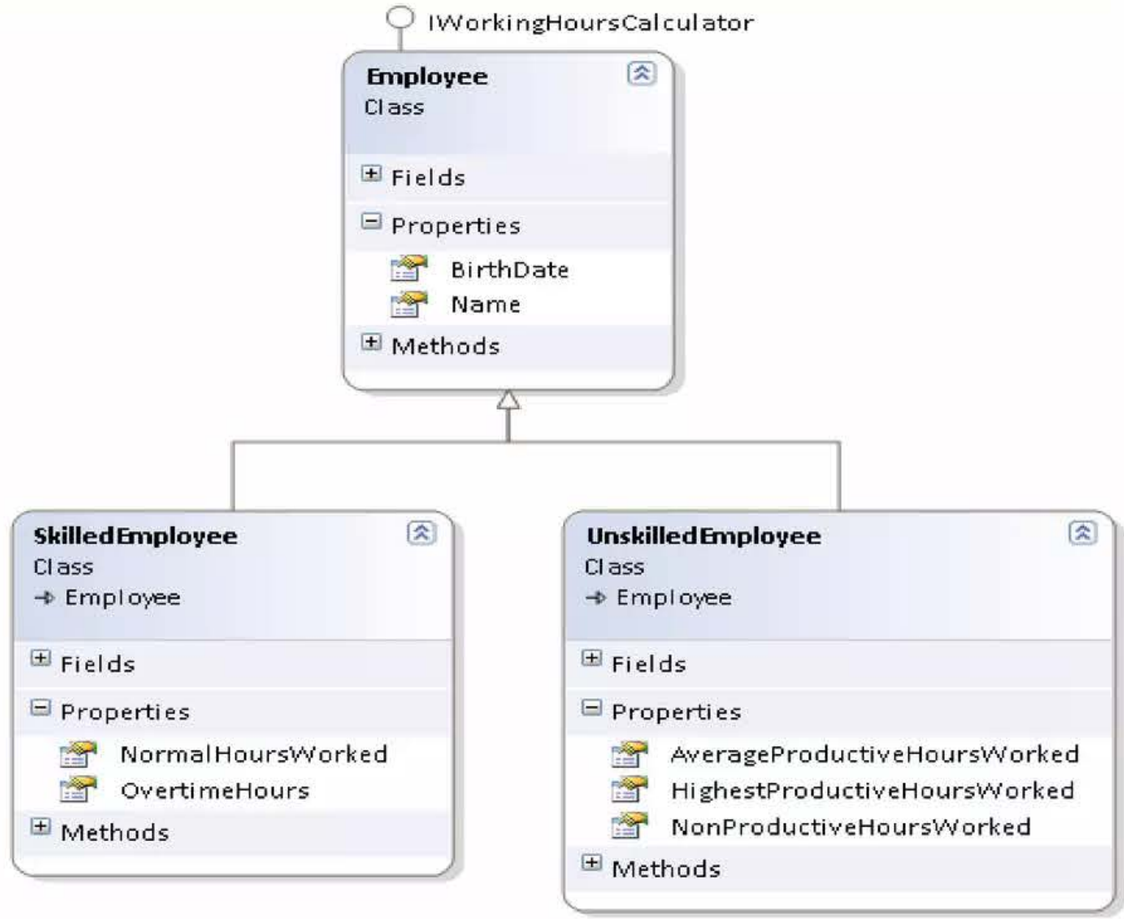
- Generic version of IComparer is used by CLR if both non Generic and Generic version is present.
- Always prefer implementing IComparer <T> for better performance (no boxing, unboxing overhead)

```
List<Employee> employeeList = new List<Employee>(employees);
IComparer<Employee> employeeBirthDateComparer = new EmployeeBirthDateComparer();
employeeList.Sort(employeeBirthDateComparer);

foreach (Employee employee in employeeList)
    Console.WriteLine("Employee - {0} {1}, Salary {2}, BithDate {3}", employee.FirstName,
        employee.LastName, employee.Salary, employee.BirthDate);
```

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;
    }
}

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;
    }
}
```

int SkilledEmployee.normalHoursWorked



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));
```

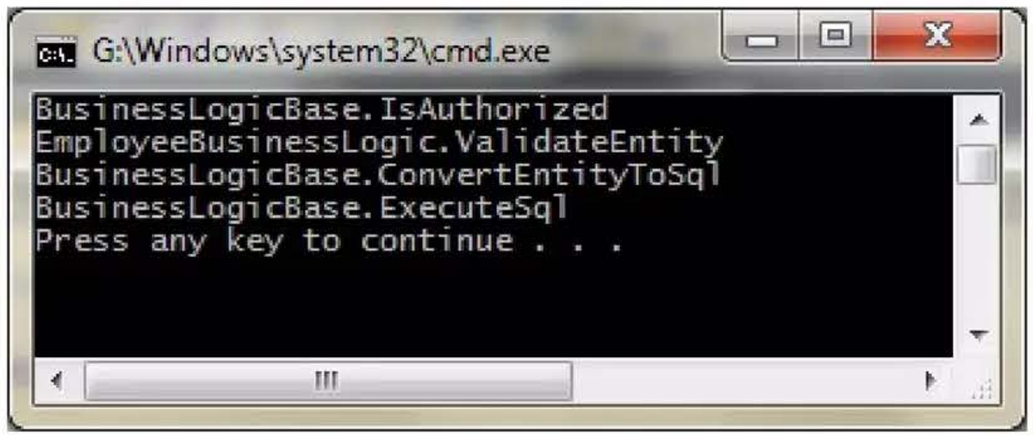
Template Method Pattern – Example 2

```
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);
```



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);
```

Strategy Design Pattern

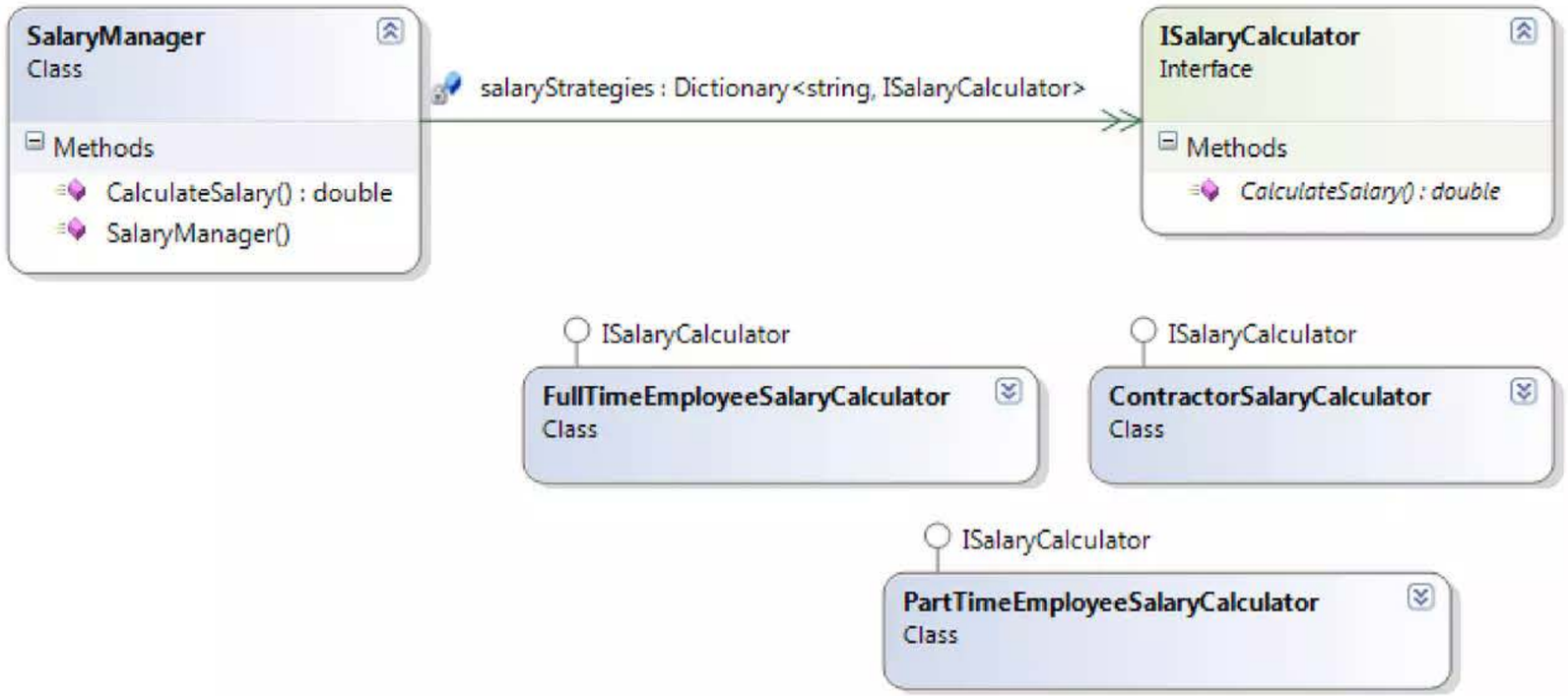
- **Strategy pattern** (also known as the **policy pattern**) is a design pattern, whereby algorithms can be selected at runtime.
- Intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable.
- The strategy pattern lets the algorithms vary independently from clients that use them.
- For example, while sorting has many algorithm (like binary sort, quick sort), the sorting algorithm is independent of how to objects/elements are compared. So the sorting algorithm can vary/change independent of compare algorithm.

Before using Strategy

```
public class Employee
{
    public string Name { get; set; }
    public DateTime JoiningDate { get; set; }
    public EmployeeType Type { get; set; }
}

public enum EmployeeType
{
    FullTimeEmployee,
    PartTimeEmployee,
    Contractor
}
public class SalaryCalculator
{
    public double CalculateSalary(Employee employee)
    {
        double salary = 0;
        switch(employee.Type)
        {
            case EmployeeType.FullTimeEmployee:
                // Calculation logic goes here
                salary = 100;
                break;
            case EmployeeType.PartTimeEmployee:
                // Calculation logic goes here
                salary = 50;
                break;
            case EmployeeType.Contractor:
                // Calculation logic goes here
                salary = 10;
                break;
        }
        return salary;
    }
}
```

```
Employee fte = new Employee
    { Name = "Scott Guthrie", Type = EmployeeType.FullTimeEmployee };
Employee contractor = new Employee
    { Name = "Amit Chatterjee", Type = EmployeeType.Contractor};

SalaryCalculator calculator = new SalaryCalculator();
Console.WriteLine(calculator.CalculateSalary(fte));
Console.WriteLine(calculator.CalculateSalary(contractor));
```

After Refactoring with Strategy



After Refactoring with Strategy

```
public interface ISalaryCalculator
{
    double CalculateSalary(Employee employee);
}

public class FullTimeEmployeeSalaryCalculator : ISalaryCalculator
{
    public double CalculateSalary(Employee employee)
    {
        return 100;
    }
}

public class PartTimeEmployeeSalaryCalculator : ISalaryCalculator
{
    public double CalculateSalary(Employee employee)
    {
        return 50;
    }
}

public class ContractorSalaryCalculator : ISalaryCalculator
{
    public double CalculateSalary(Employee employee)
    {
        return 10;
    }
}
```

After Refactoring with Strategy (Cont'd)

```
public class SalaryManager
{
    Dictionary<string, ISalaryCalculator> salaryStrategies =
        new Dictionary<string, ISalaryCalculator>();

    public SalaryManager()
    {
        salaryStrategies.Add("FullTimeEmployee", new FullTimeEmployeeSalaryCalculator());
        salaryStrategies.Add("PartTimeEmployee", new PartTimeEmployeeSalaryCalculator());
        salaryStrategies.Add("Contractor", new ContractorSalaryCalculator());
    }

    public double CalculateSalary(Employee employee)
    {
        return salaryStrategies[employee.Type.ToString()].CalculateSalary(employee);
    }
}
```

```
Employee fte = new Employee
    { Name = "Scott Guthrie", Type = EmployeeType.FullTimeEmployee };
Employee contractor = new Employee
    { Name = "Amit Chatterjee", Type = EmployeeType.Contractor};

var manager = new SalaryManager();
Console.WriteLine(manager.CalculateSalary(fte));
Console.WriteLine(manager.CalculateSalary(contractor));
```

Template Method in ASP.NET Controls

- Different options for custom controls
  - For simple controls used in single project, user control is best choice.
  - Control used in several Web applications or requires more functionality, a custom server control may be a better fit. There are two general types:
    - Controls that combine the functionality of several existing controls (called composite controls),
    - Controls with a unique visual representation.
    - Process for creating both of these types is very similar. For composite controls, create a new class that inherits from control base classes (like Control or WebControl) and then override the CreateChildControls method. In this method add controls whose functionality you are combining to the collection of child controls, called Controls.
    - For other custom controls, you override Render instead and use the HtmlTextWriter parameter to output the HTML for your control directly.
    - Regardless of which style of custom control you choose, you don't have to write any code to handle the functionality that's common to all controls, like loading and saving ViewState at the right time, allowing PostBack events to be handled, and making sure the control lifecycle events are raised in the correct order. The main algorithm for how a control should be loaded, rendered, and unloaded is contained in the control base class.

Difference between Template and Strategy

- Strategy is used to allow callers to vary an entire algorithm, like how to compare two objects, while Template Method is used to vary steps in an algorithm.
- Because of this, Strategy is more coarsely grained.
- There can be vast differences between different client implementations, while with Template Method the algorithm remains fundamentally the same.
- Strategy uses delegation while Template Method uses inheritance. In the sorting example of Strategy, the comparison algorithm is delegated to the IComparer parameter, but with custom controls you subclass the base and override methods to make changes. Both, however, let you easily alter processes to fit your specific needs.

AntiPatterns

- Pattern that may be commonly used but is ineffective and/or counterproductive in practice.
- The term was coined in 1995 by Andrew Koenig inspired by Gang of Four's book *Design Patterns*,
- The term was widely popularized three years later by the book *AntiPatterns*

Organizational AntiPatterns

- Analysis paralysis: Devoting disproportionate effort to the analysis phase of a project
- Cash cow: A profitable legacy product that often leads to complacency about new products
- Design by committee: The result of having many contributors to a design, but no unifying vision
- Moral hazard: Insulating a decision-maker from the consequences of his or her decision.
- Stovepipe or Silos: A structure that supports mostly up-down flow of data but inhibits cross organizational communication
- Vendor lock-in: Making a system excessively dependent on an externally supplied component

Software Design AntiPatterns

- Abstraction inversion: Not exposing implemented functionality required by users, so that they re-implement it using higher level functions
- Ambiguous viewpoint: Presenting a model (usually Object-oriented analysis and design (OOAD)) without specifying its viewpoint
- Big ball of mud: A system with no recognizable structure
- Database-as-IPC: Using database as message queue for interprocess communication where a more lightweight mechanism would be suitable
- Gold plating: Continuing to work on a task or project well past the point at which extra effort is adding value
- Inner-platform effect: A system so customizable as to become a poor replica of the software development platform
- Input kludge: Failing to specify and implement the handling of possibly invalid input
- Interface bloat: Making an interface so powerful that it is extremely difficult to implement
- Magic pushbutton: Coding implementation logic directly within interface code, without using abstraction.
- Race hazard: Failing to see the consequence of different orders of events
- Stovepipe system: A barely maintainable assemblage of ill-related components

Object Oriented Design AntiPattern

- Anemic Domain Model: Use of domain model without business logic. The domain model's objects cannot guarantee their correctness at any moment, because their validation logic is placed somewhere outside (most likely in multiple places).
- BaseBean: Inheriting functionality from a utility class rather than delegating to it
- Call super: Requiring subclasses to call a superclass's overridden method
- Circle-ellipse problem: Subtyping variable-types on the basis of value-subtypes
- Circular dependency: Introducing unnecessary direct or indirect mutual dependencies between objects or software modules
- Constant interface: Using interfaces to define constants
- God object: Concentrating too many functions in a single part of design (class)
- Object cesspool: Reusing objects whose state does not conform to the (possibly implicit) contract for re-use
- Object orgy: Failing to properly encapsulate objects permitting unrestricted access to their internals
- Poltergeists: Objects whose sole purpose is to pass information to another object
- Sequential coupling: A class that requires its methods to be called in a particular order
- Yo-yo problem: A structure (e.g., of inheritance) that is hard to understand due to excessive fragmentation
- Dependency hell: Problems with versions of required products
- DLL hell: Inadequate management of dynamic-link libraries (DLLs), specifically on Microsoft Windows

Programming AntiPattern

- Accidental complexity: Introducing unnecessary complexity into a solution
- Action at distance: Unexpected interaction between widely separated parts of system
- Blind faith: Lack of checking of correctness of a bug fix or result of a subroutine
- Boat anchor: Retaining a part of a system that no longer has any use
- Busy spin: Consuming CPU while waiting for something to happen, usually by repeated checking instead of messaging
- Caching failure: Forgetting to reset an error flag when an error has been corrected
- Cargo cult programming: Using patterns and methods without understanding why
- Coding by exception: Adding new code to handle each special case as it is recognized
- Error hiding: Catching an error message before it can be shown to the user and either showing nothing or showing a meaningless message
- Hard code: Embedding assumptions about environment of a system in its implementation
- Lava flow: Retaining undesirable (redundant or low-quality) code because removing it is too expensive or has unpredictable consequences
- Loop-switch sequence: Encoding a set of sequential steps using a switch within a loop statement
- Magic numbers: Including unexplained numbers in algorithms
- Magic strings: Including literal strings in code, for comparisons, as event types etc.
- Soft code: Storing business logic in configuration files rather than source code[7]
- Spaghetti code: Programs whose structure is barely comprehensible, especially because of misuse of code structures

Methodological AntiPatterns

- Copy and paste programming: Copying (and modifying) existing code rather than creating generic solutions
- Golden hammer: Assuming that a favorite solution is universally applicable
- Improbability factor: Assuming that it is improbable that a known error will occur
- Not Invented Here (NIH) syndrome: The tendency towards reinventing the wheel (Failing to adopt an existing, adequate solution)
- Premature optimization: Coding early-on for perceived efficiency, sacrificing good design, maintainability, and sometimes even real-world efficiency
- Programming by permutation (or "programming by accident"): Trying to approach a solution by successively modifying the code to see if it works
- Reinventing the wheel: Failing to adopt an existing, adequate solution
- Reinventing the square wheel: Failing to adopt an existing solution and instead adopting a custom solution which performs much worse than the existing one.
- Silver bullet: Assuming that a favorite technical solution can solve a larger process or problem
- Tester Driven Development: Software projects in which new requirements are specified in bug reports

Resources

- Discover the Design Patterns You're Already Using in the .NET Framework
- Exploring the Observer Design Pattern
- Observer Design Pattern in .NET 4
- AntiPattern
- AntiPattern: Wikipedia

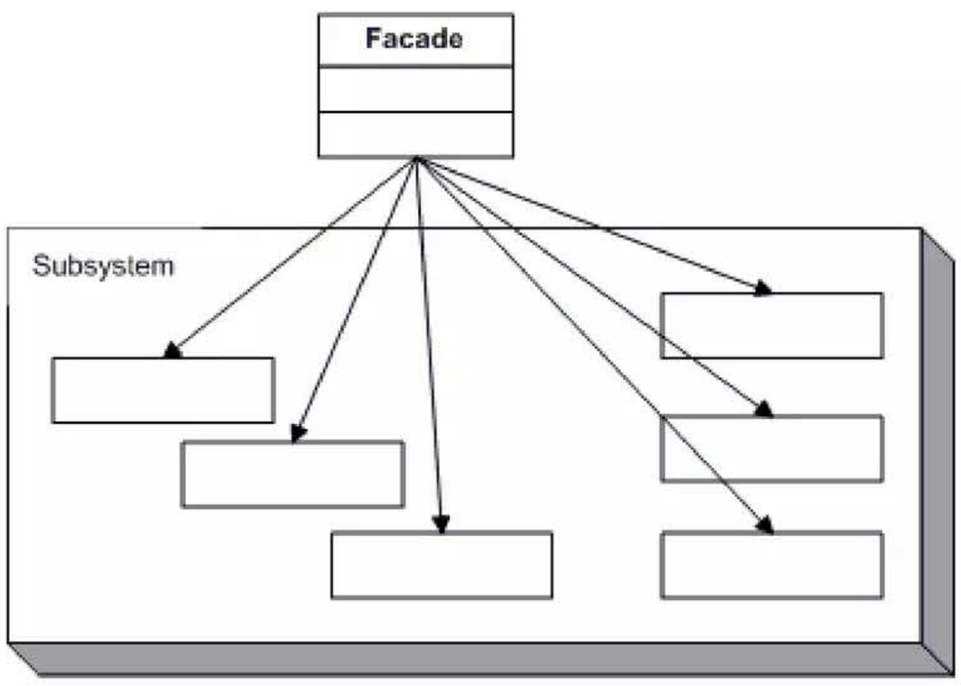
Facade

- 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".

Carlo Maderno's monumental façade of Saint Peter's basilica in Vatican City

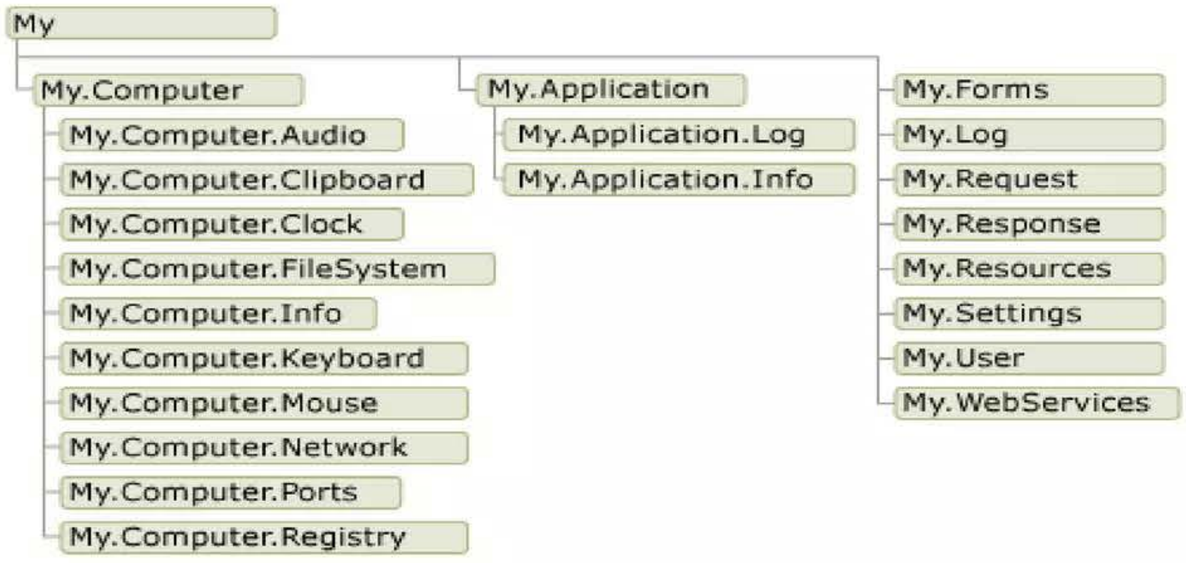
Façade Pattern

- 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, since the facade has convenient methods for common tasks
  - Make code that uses the library more readable, for the same reason
  - Reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system
  - Wrap a poorly-designed collection of APIs with a single well-designed API



Façade in .NET Visual Basic - My namespace

- My namespace
  - **Application:** Gets access to Application object
  - **Computer:** Contains info about computer like Audio, Clock Clipboard, File System, Registry, Keyboard, Mouse, Screen, Network, Ports etc
  - **Forms:** Provides access to all Forms in a Win Forms App
  - **Settings:** Provides access to Settings collection
  - **User:** Points to the current logged in user (Principal, Role)
  - **WebService:** Provides access to all Web Services referenced



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

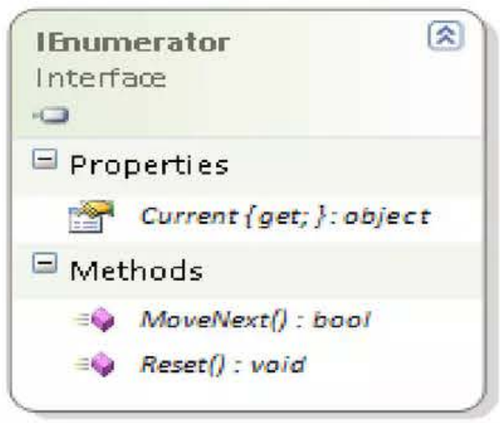
```
class Loan
{
    public bool HasNoBadLoans(Customer cust)
    {
        Console.WriteLine("Check loans for " + cust.Name);
        return true;
    }
}
```

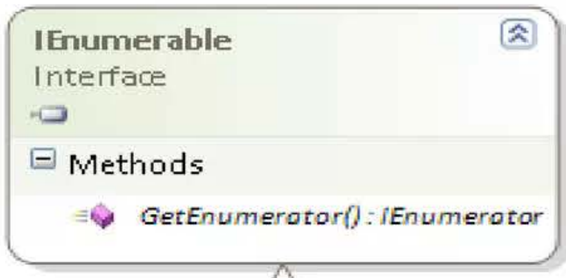
```
class MortgageFacade
{
    private static Bank bank = new Bank();
    private static Loan loan = new Loan();
    private static Credit credit = new Credit();

    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;
    }
}
```

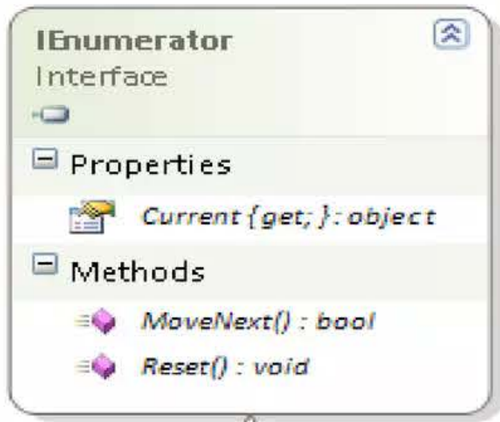
Implementing Façade in .NET (Cont'd)

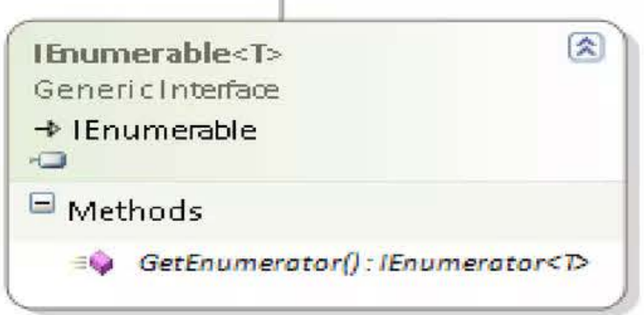
```
// Evaluate mortgage eligibility for customer
Customer customer = new Customer() { Name = "Bill Gates" };
bool eligible = MortgageFacade.IsEligible(customer, 125000);

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

Iterator in .NET: Enumerator

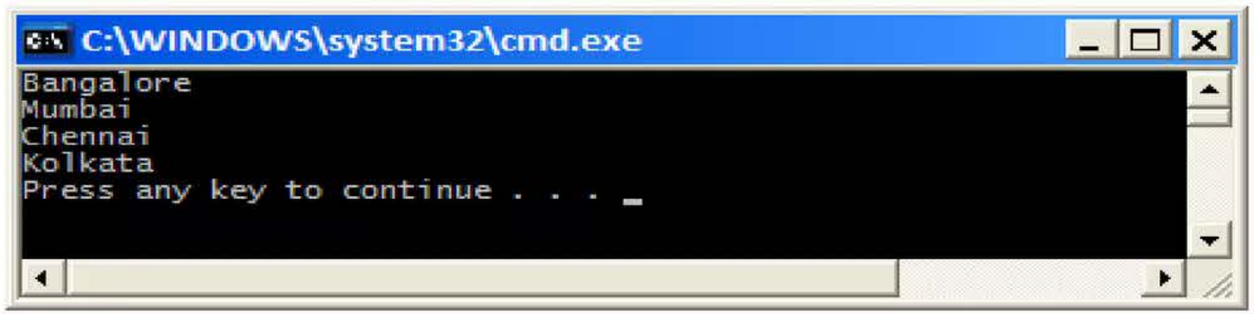






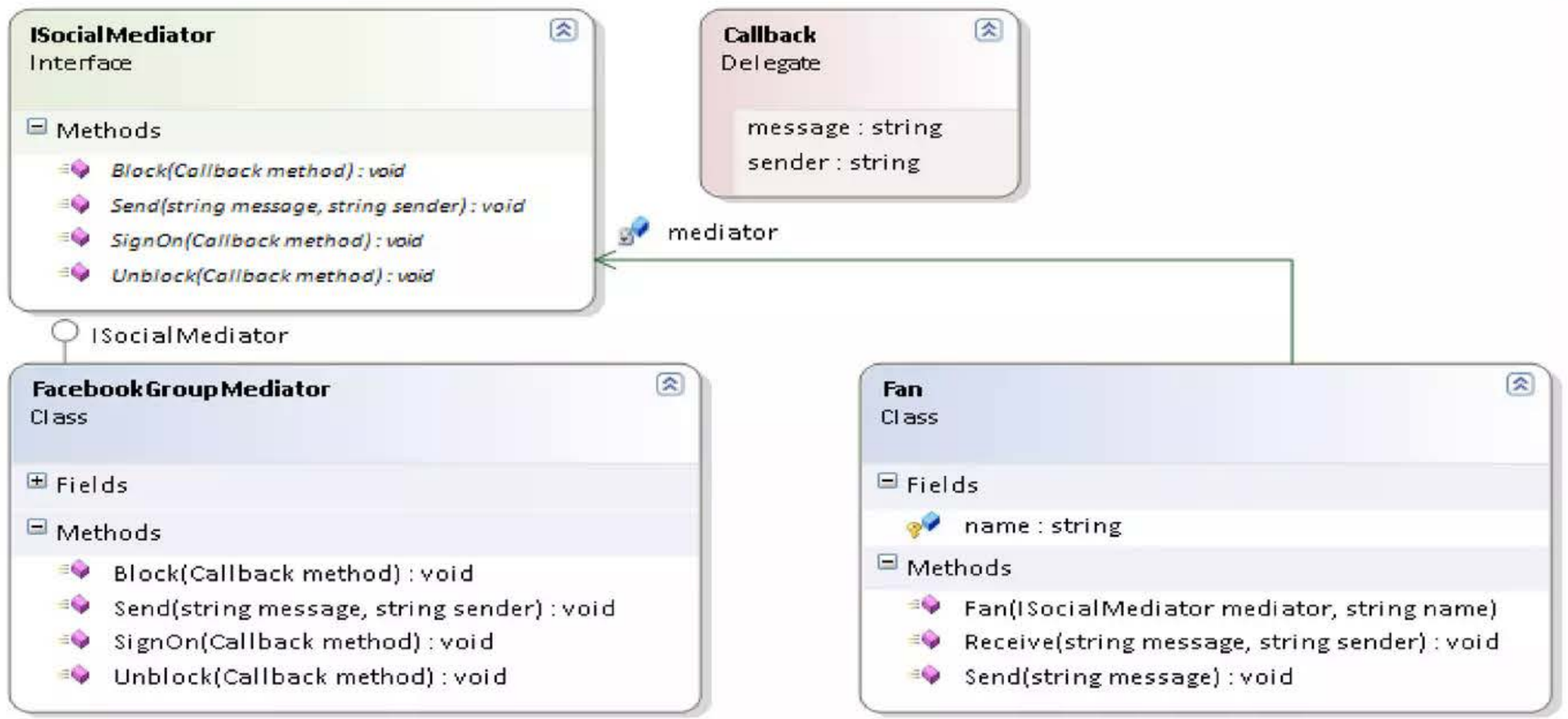



```
string[] cities = { "Bangalore", "Mumbai", "Chennai", "Kolkata" };

foreach (string city in cities)
    Console.WriteLine(city);
```




```
// Since array impmlenets IEnumerable interface calling
// GetEnumerator returns the appropraite enumerator
IEnumerator enumerator = cities.GetEnumerator();

// MoveNext returns false when current element is the
// last element in array/collection
while(enumerator.MoveNext())
    Console.WriteLine(enumerator.Current);
```

Mediator class diagram



Mediator

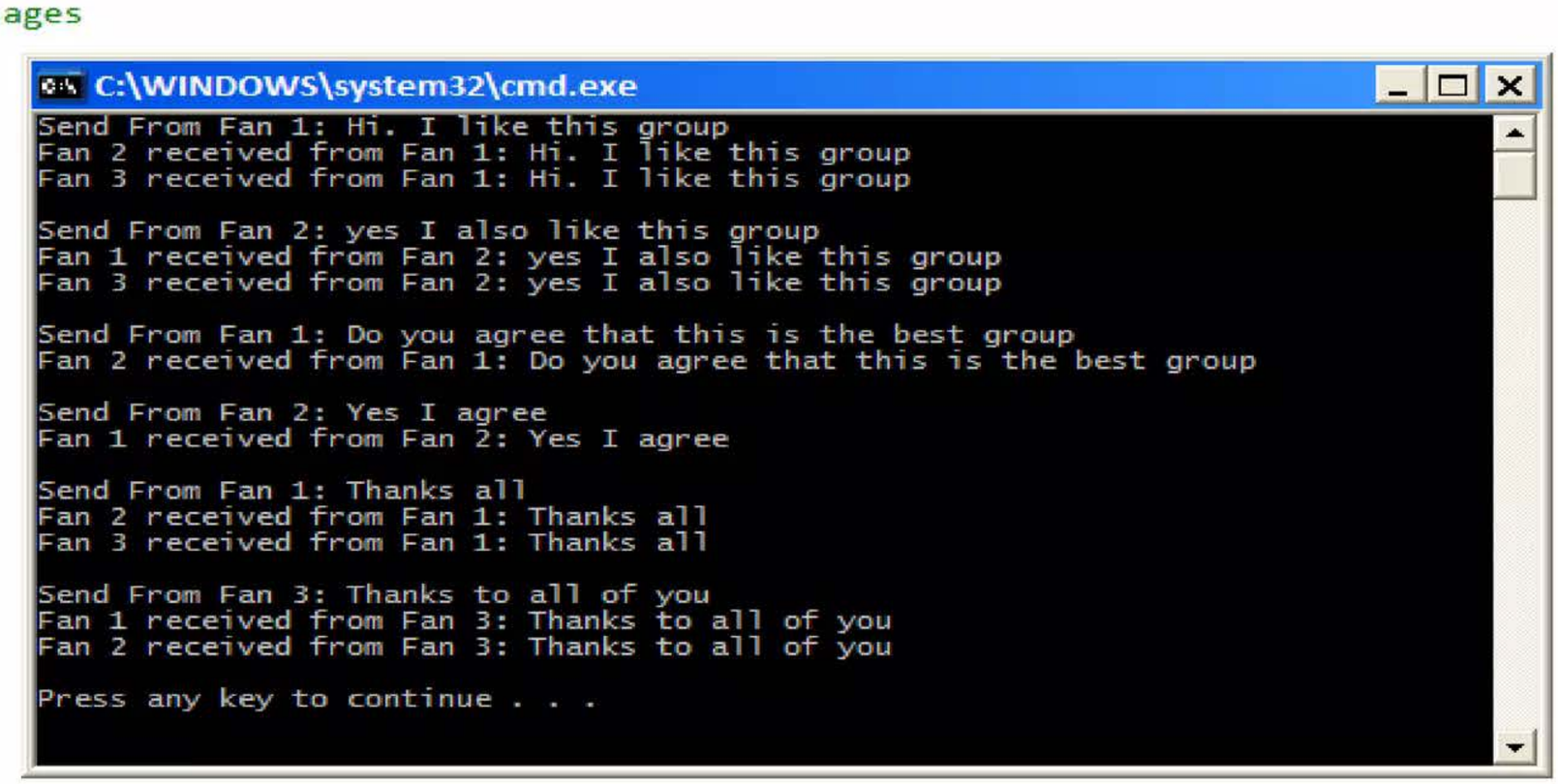
```
public delegate void Callback(string message, string sender);

public interface ISocialMediator
{
    void SignOn(Callback method); // Signon to Facebook
    void Block(Callback method); // Block the particular Fan
    void Unblock(Callback method); // Unblock the particular Fan
    void Send(string message, string sender); // Send the message to group
}

public class FacebookGroupMediator : ISocialMediator
{
    Callback respond;

    public void SignOn(Callback method)
    {
        respond += method;
    }
    public void Block(Callback method)
    {
        respond -= method;
    }
    public void Unblock(Callback method)
    {
        respond += method;
    }
    // Send is implemented as a broadcast
    public void Send(string message, string sender)
    {
        respond(message, sender);
        Console.WriteLine();
    }
}
```

Mediator

```
class Fan
{
    ISocialMediator mediator; // Reference to mediator
    protected string name; // Fan Name

    public Fan(ISocialMediator mediator, string name)
    {
        this.mediator = mediator;
        this.name = name;
        mediator.SignOn(Receive);
    }
    public virtual void Receive(string message, string sender)
    {
        if (!String.Equals(sender, name)) // message should not be send to sender
            Console.WriteLine(name + " received from " + sender + ": " + message);
    }
    public void Send(string message)
    {
        Console.WriteLine("Send From " + name + ": " + message);
        mediator.Send(message, name);
    }
}
```

Mediator

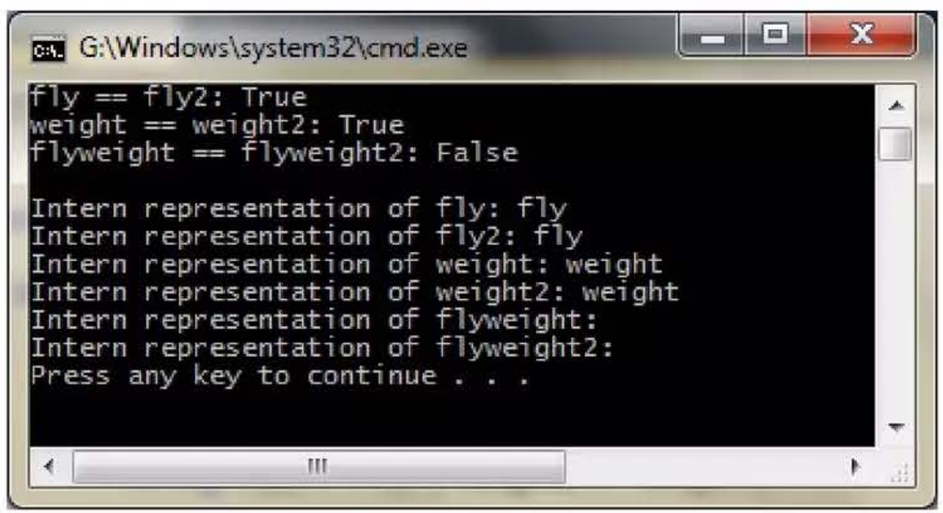
```
static void Main(string[] args)
{
    FacebookGroupMediator mediator = new FacebookGroupMediator();
    Fan fan1 = new Fan(mediator, "Fan 1");
    Fan fan2 = new Fan(mediator, "Fan 2");
    Fan fan3 = new Fan(mediator, "Fan 3");

    fan1.Send("Hi. I like this group");
    fan2.Send("yes I also like this group");

    // Block fan3 temporarily so that he does not get messages
    mediator.Block(fan3.Receive);
    fan1.Send("Do you agree that this is the best group");
    fan2.Send("Yes I agree");

    // Unblock fan3 so that he gets messages
    mediator.Unblock(fan3.Receive);
    fan1.Send("Thanks all");
    fan3.Send("Thanks to all of you");
}
```



Flyweight in .NET BCL

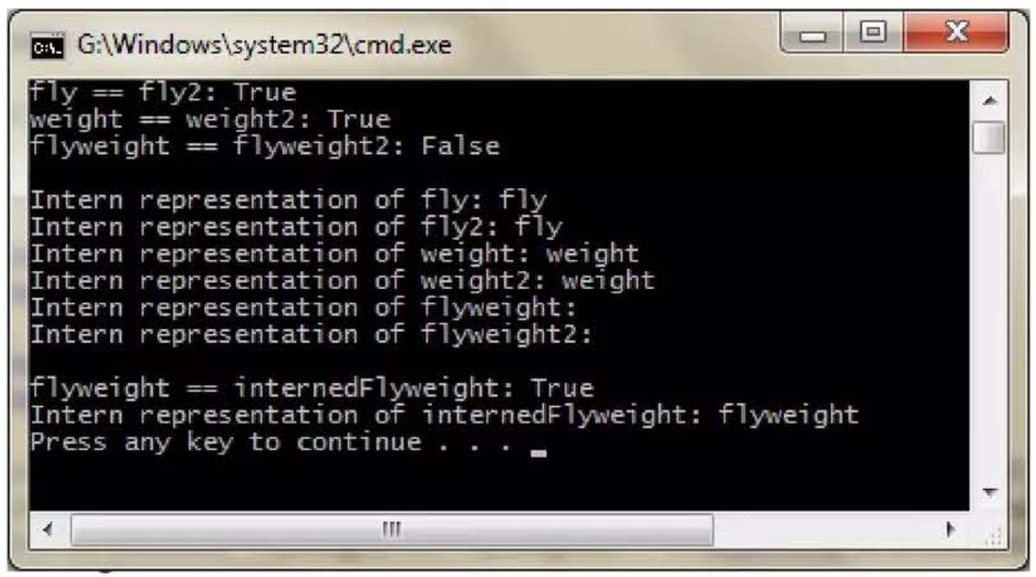
- String Interning in .NET (Java also does the same)

```
string fly = "fly", weight = "weight";
string fly2 = "fly", weight2 = "weight";

string flyweight = fly + weight;
string flyweight2 = fly2 + weight2;

Console.WriteLine("fly == fly2: " + object.ReferenceEquals(fly, fly2));
Console.WriteLine("weight == weight2: " + object.ReferenceEquals(weight, weight2));
Console.WriteLine("flyweight == flyweight2: " + object.ReferenceEquals(flyweight, flyweight2));
Console.WriteLine("");

Console.WriteLine("Intern representation of fly: " + string.IsInterned(fly));
Console.WriteLine("Intern representation of fly2: " + string.IsInterned(fly2));
Console.WriteLine("Intern representation of weight: " + string.IsInterned(weight));
Console.WriteLine("Intern representation of weight2: " + string.IsInterned(weight2));
Console.WriteLine("Intern representation of flyweight: " + string.IsInterned(flyweight));
Console.WriteLine("Intern representation of flyweight2: " + string.IsInterned(flyweight2));
```



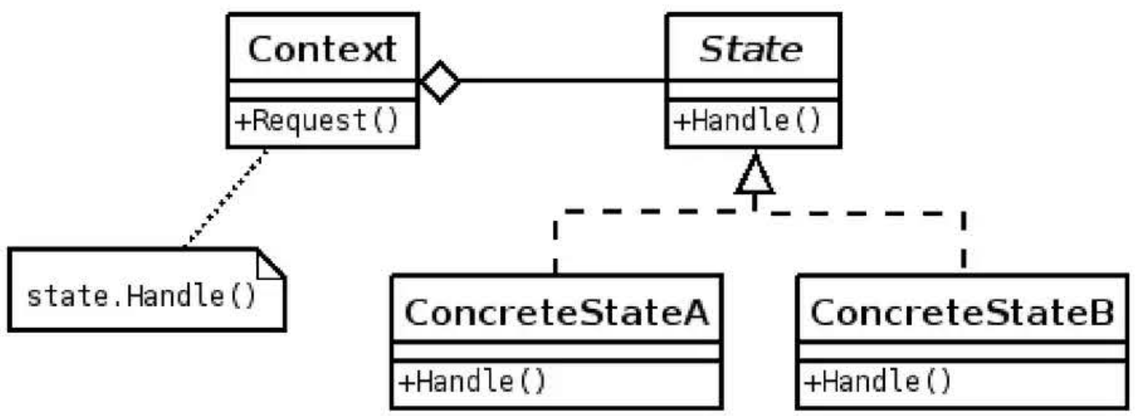
Flyweight in .NET BCL (Cont'd)

```
string fly = "fly", weight = "weight";
string fly2 = "fly", weight2 = "weight";

string flyweight = fly + weight;
string flyweight2 = fly2 + weight2;

Console.WriteLine("fly == fly2: " + object.ReferenceEquals(fly, fly2));
Console.WriteLine("weight == weight2: " + object.ReferenceEquals(weight, weight2));
Console.WriteLine("flyweight == flyweight2: " + object.ReferenceEquals(flyweight, flyweight2));
Console.WriteLine("");

Console.WriteLine("Intern representation of fly: " + string.IsInterned(fly));
Console.WriteLine("Intern representation of fly2: " + string.IsInterned(fly2));
Console.WriteLine("Intern representation of weight: " + string.IsInterned(weight));
Console.WriteLine("Intern representation of weight2: " + string.IsInterned(weight2));
Console.WriteLine("Intern representation of flyweight: " + string.IsInterned(flyweight));
Console.WriteLine("Intern representation of flyweight2: " + string.IsInterned(flyweight2));
Console.WriteLine("");

string internedFlyweight = string.Intern(flyweight);
Console.WriteLine("flyweight == internedFlyweight: " + object.ReferenceEquals(flyweight, internedFlyweight));
Console.WriteLine("Intern representation of internedFlyweight: " + string.IsInterned(internedFlyweight));
```

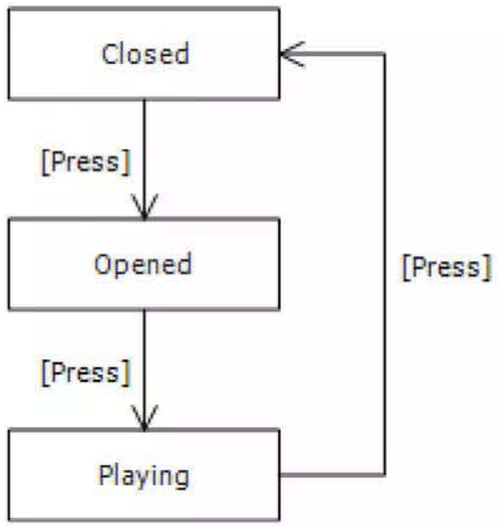


State Design Pattern

- Supports an object that has several states
- The state an object determines the behavior of several methods. Without State pattern could be implemeted with if else / swicth case statements in each method
- Better solution: state pattern
- Have a reference to a state object
  - Normally, state object doesn't contain any fields
  - Change state: change state object
  - Methods delegate to state object



Without State Pattern

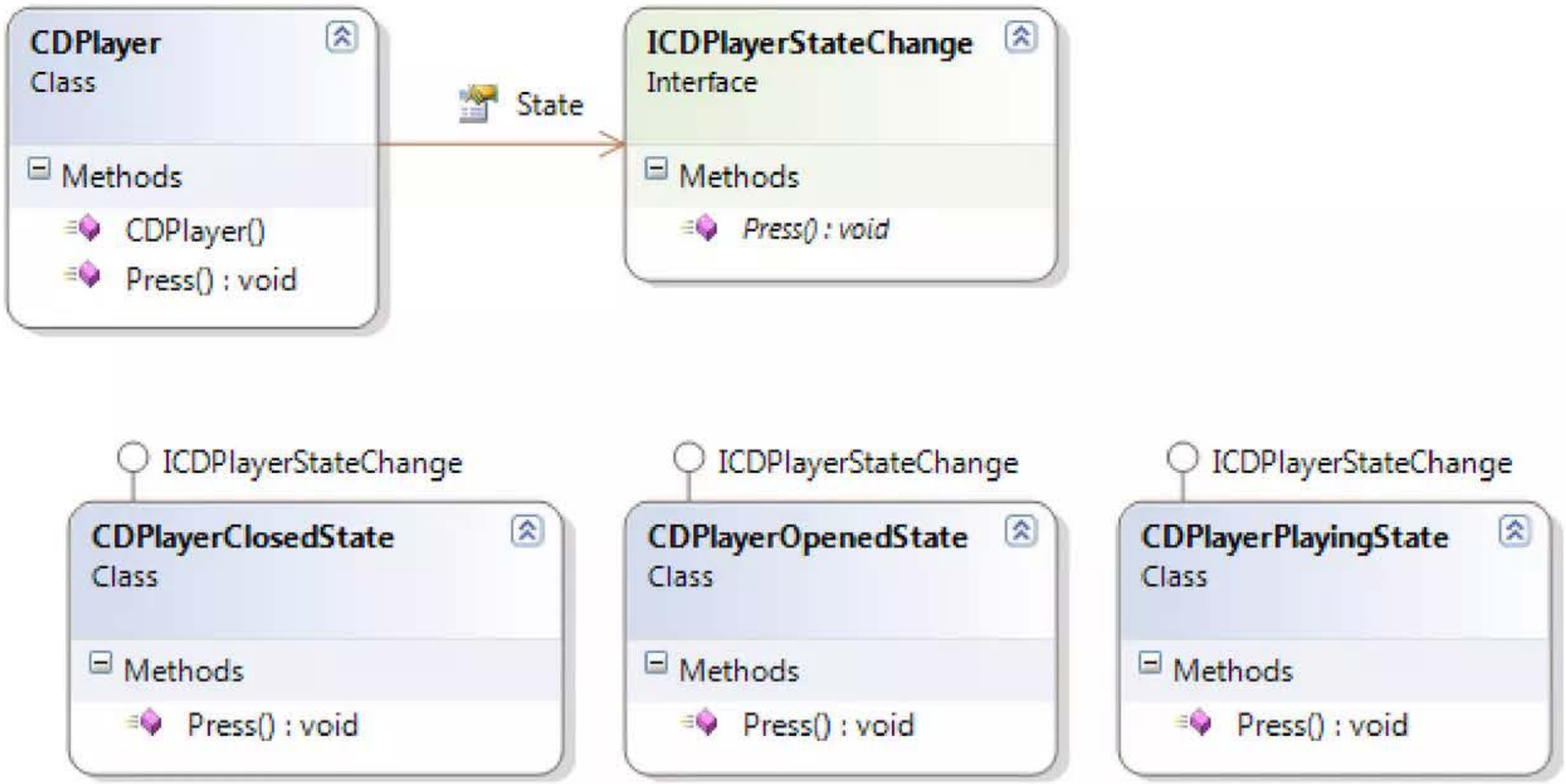


CD Player State Diagram

```
static void Main(string[] args)
{
    CDPlayer cdPlayer = new CDPlayer();
    // Change CD Player state from Closed to Opened
    cdPlayer.Press();

    // Change CD Player state from Opened to Playing
    cdPlayer.Press();

    // Change CD Player state from Playing to Closed
    cdPlayer.Press();
}
```

```
public class CDPlayer
{
    public CDPlayerState State { get; set; }

    public void Press()
    {
        if (State == CDPlayerState.Closed)
        {
            State = CDPlayerState.Opened;
            Console.WriteLine("CD Player earlier State: Closed");
            Console.WriteLine("CD Player new State: Opened\n");
        }
        else if (State == CDPlayerState.Opened)
        {
            State = CDPlayerState.Playing;
            Console.WriteLine("CD Player earlier State: Opened");
            Console.WriteLine("CD Player new State: Playing\n");
        }
        else if (State == CDPlayerState.Playing)
        {
            State = CDPlayerState.Closed;
            Console.WriteLine("CD Player earlier State: Playing");
            Console.WriteLine("CD Player new State: Closed\n");
        }
    }
}

public enum CDPlayerState
{
    Closed,
    Opened,
    Playing
}
```

Implementing State Pattern in .NET



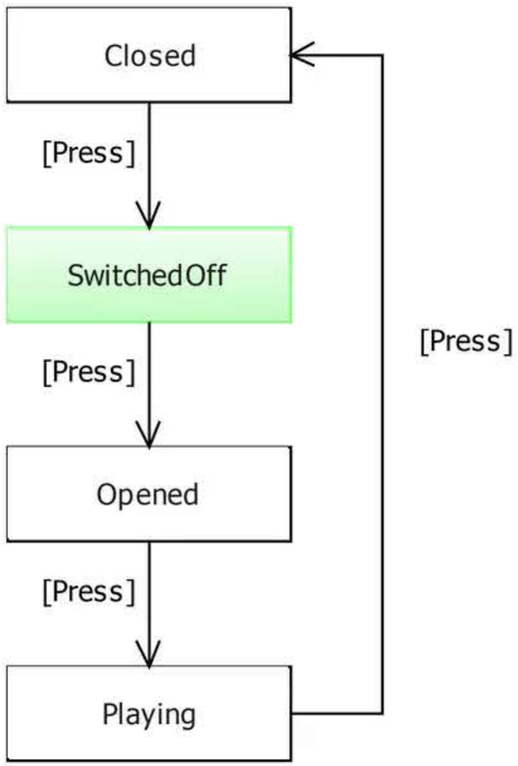
- Different classes representing different states of the object (CD Player)
- Each State specific class implements an interface (ICDPlayerStateChange)
- Each State specific class contains the code for the state change. The object (CDPlayer) simply delegates the call to these state specific classes.
- State of the object (CD Player) is managed by new State specific classes.

Implementing State Pattern in .NET (Cont'd)

```
public class CDPlayer
{

    public ICDPlayerStateChange State { get; set; }

    public CDPlayer()
    {
        State = new CDPlayerClosedState();
    }

    public void Press()
    {
        State.Press(this);
    }
}
```

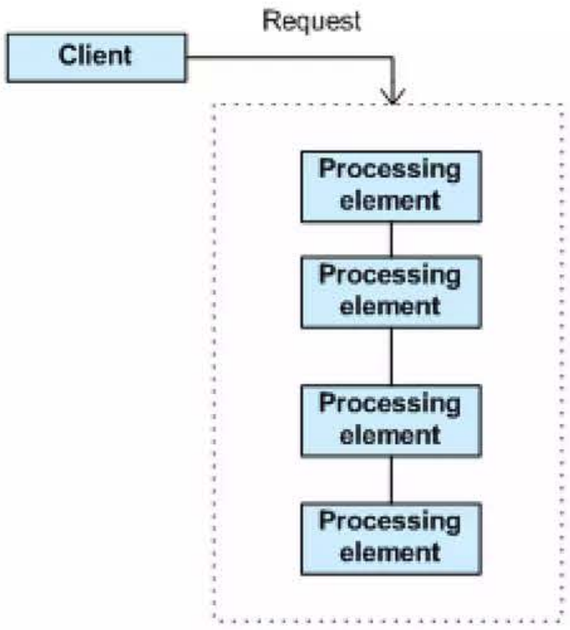

```
public interface ICDPlayerStateChange
{
    void Press(CDPlayer cdPlayer);
}

partial class CDPlayerClosedState : ICDPlayerStateChange
{
    public void Press(CDPlayer cdPlayer)
    {
        cdPlayer.State = new CDPlayerOpenedState();
        Console.WriteLine("CD Player earlier State: Closed");
        Console.WriteLine("CD Player new State: Opened\n");
    }
}

partial class CDPlayerOpenedState : ICDPlayerStateChange
{
    public void Press(CDPlayer cdPlayer)
    {
        cdPlayer.State = new CDPlayerPlayingState();
        Console.WriteLine("CD Player earlier State: Opened");
        Console.WriteLine("CD Player new State: Playing\n");
    }
}

partial class CDPlayerPlayingState : ICDPlayerStateChange
{
    public void Press(CDPlayer cdPlayer)
    {
        cdPlayer.State = new CDPlayerClosedState();
        Console.WriteLine("CD Player earlier State: Playing");
        Console.WriteLine("CD Player new State: Closed\n");
    }
}
```

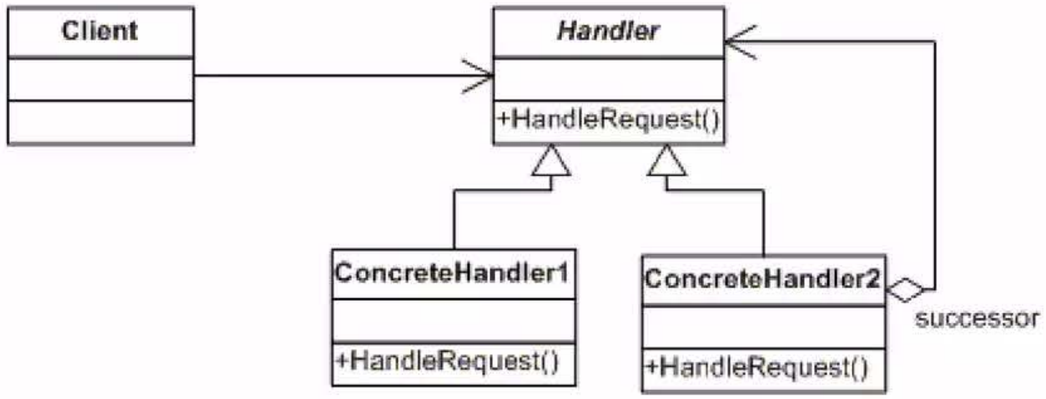
Implementing State Pattern in .NET (Cont'd)



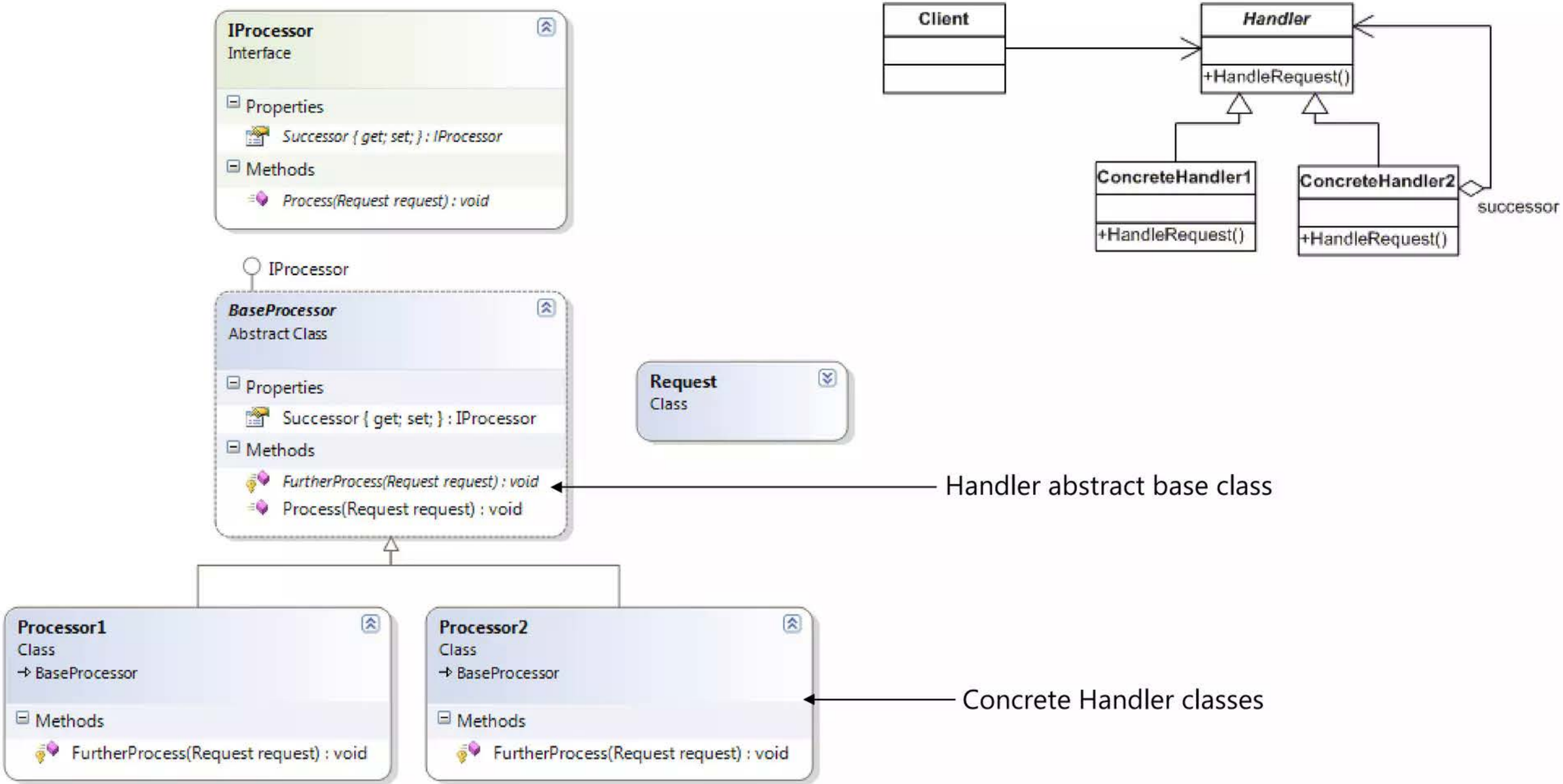
Chain of Responsibility

- Decouples the sender of the request to the receiver. The only link between sender and the receiver is the request which is sent. Based on the request data sent, the receiver is picked. This is called “data-driven”.
- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
- Promotes the idea of loose coupling.





Implementing Chain of Resp. in .NET

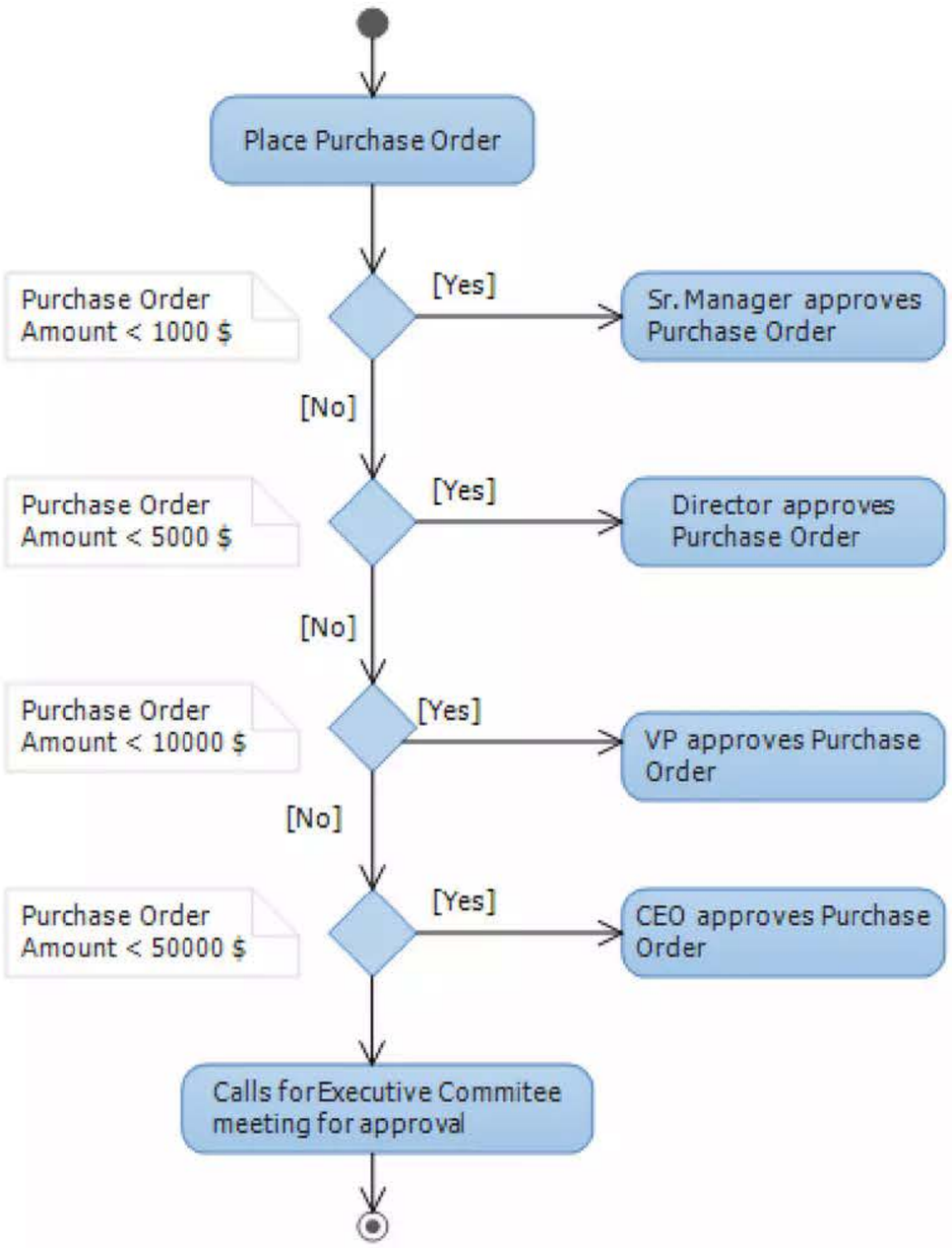


Implementing Chain of Resp. in .NET

```
public class Request { }

public interface IProcessor
{
    IProcessor Successor
    { get; set; }

    void Process(Request request);
}

public abstract class BaseProcessor : IProcessor
{
    public IProcessor Successor { get; set; }

    public void Process(Request request)
    {
        // Some base class behaviour
        FurtherProcess(request);
        if (null != Successor)
        {
            Successor.Process(request);
        }
    }

    protected abstract void FurtherProcess(Request request);
}
```

```
public class Processor1 : BaseProcessor
{
    protected override void FurtherProcess(Request request)
    {
        // Do something usefull
    }
}

public class Processor2 : BaseProcessor
{
    protected override void FurtherProcess(Request request)
    {
        // Do something usefull
    }
}
```

Implementing Chain of Resp. in .NET



Implementing Chain of Resp. in .NET

```
public class PurchaseOrder
{
    public double TotalAmount { get; set; }
    public string Item { get; set; }
    public double Rate { get; set; }
    public string Purpose { get; set; }
}

public abstract class Approver
{
    public Approver Successor { get; set; }
    public double ApprovalLimit { get; set; }

    public void Approve(PurchaseOrder purchaseOrder)
    {
        if (purchaseOrder.TotalAmount <= ApprovalLimit)
        {
            Console.WriteLine("Purchase Order {0} Total Amount {1} processed by {2}",
                purchaseOrder.Item, purchaseOrder.TotalAmount, this.GetType().Name);
        }

        else
        {
            Successor.Approve(purchaseOrder);
        }
    }
}
```

```
public class SeniorManager : Approver
{
    public SeniorManager()
    {
        this.ApprovalLimit = 1000.0;
        this.Successor = new Director();
    }
}

public class Director : Approver
{
    public Director()
    {
        this.ApprovalLimit = 5000.0;
        this.Successor = new VicePresident();
    }
}

public class VicePresident : Approver
{
    public VicePresident()
    {
        this.ApprovalLimit = 10000.0;
        this.Successor = new CEO();
    }
}

public class CEO : Approver
{
    public CEO()
    {
        this.ApprovalLimit = 50000.0;
        this.Successor = new ExecutiveCommittee();
    }
}

public class ExecutiveCommittee : Approver
{
    public ExecutiveCommittee()
    {
        this.ApprovalLimit = 9999999999999999.99;
    }
}
```

Implementing Chain of Resp. in .NET

```
public class ApprovalManager
{
    public static void Approve(PurchaseOrder purchaseOrder)
    {
        Approver approver = new SeniorManager();
        approver.Approve(purchaseOrder);
    }
}
```

- ApprovalManager class acts like a Façade. You may not use this class - you can directly instantiate SeniorManager class from client.
- The advantage of using a Façade is any change in hierarchy would be encapsulated from the client, as it would be handled in ApprovalManager class.

```
static void Main(string[] args)
{
    PurchaseOrder po = new PurchaseOrder { Item = "Load Balancers", TotalAmount = 2000.0 };
    ApprovalManager.Approve(po);

    po = new PurchaseOrder { Item = "Xeon Servers", TotalAmount = 40000.0 };
    ApprovalManager.Approve(po);

    po = new PurchaseOrder { Item = "Setting up new ODC", TotalAmount = 900000.0 };
    ApprovalManager.Approve(po);
}
```