.NET Design Patterns and Advanced Techniques

DN7 ®

Version 5.0

Sela College

This course material has been prepared by:

Sela Software Labs Ltd.

14-18 Baruch Hirsch St. Bnei Brak 51202 Israel

Tel: 972-3- 6176666 Fax: 972-3- 6176667

The Use of this book

The material in this book is designed to assist the student during the course. It does not include all of the information that will be referred to during the course and should not be regarded as a replacement for reference manuals.

Limits of Responsibility

Sela Software Labs Ltd invests significant effort in updating this book, however, Sela Software Labs Ltd is not responsible for any errors or material which may not meet specific requirements. The user alone is responsible for decisions based on the information contained in this book.

Protected Trademarks

In this book, protected trademarks appear that are under copyright. All rights to the trademarks in this material are reserved to the authors.

SELA wishes you success in the course!

Table of Contents

SELA

Module 01 - Introduction

.NET Design Patterns

Content

- Introduction to Object Oriented Design
- Good Design Principles
- What are design patterns
- Design patterns in .NET
- Common Terminology

The Course Goals

- Get **familiar** with Design Patterns
- **Understand** their **strengths and weaknesses**
- Be familiar with **modern C#** language capabilities
- Discuss architecture **trade-off** and **when / which pattern** can be **fit** to a problem
- Get familiar with the Pattern **Jargon**

Introduction to Object Oriented Design - Just a Short Reminder

- **Objects** are **instances** of **classes** that defines the object's **structure**

Introduction to Object Oriented Design (Cont.)

- Tenets
  - **Encapsulation**
  - **Inheritance**
  - **Polymorphism**
  - **Abstraction**
- capabilities
  - **Overloading**
  - **Overriding**
  - **Operator Overloading**

.Net Interface

- **Interface = Contract**
- When Classes **implement** an interface
  it must apply the entire interface contract
- **Interface** is great tool for **API design**
  - **Abstract** the **API** to a common intent
- **Testability**: abstracting dependencies via interface,
  make it better for testing.

.Net Interface vs Abstract Class

	Interface	Abstract Class
Can Have: **State**	**X**	**V**
Can Have: **Default implementation** (will allow to extend without breaking compatibility)	**X** up to **C#8**	**V**
Support **multi inheritance / implementation**	**V**	**X**
Loosely Coupled	**Y**	**X** (Limit for specific implementation)
Visibility	**Public** (can be hidden from Intellisense)	**Public** **Protected** **Private**
Terminology	**I Know** something	**I Have** something

Extension Methods

- Compile time Extension for class or Interface
- Challenges:
  - How do you add **functionality** to a sealed class?
  - How do you add **functionality** to an interface without changing it?
- Static methods are the canonical solution

```
public static class MyExtensions
{
        public static int ToInt(this string s)
        {
                return int.Parse(s);
        }
}
```

Extension Methods

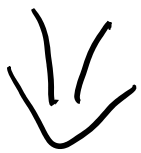
- Challenges:
  - How do you add **functionality** to a sealed class?
  - How do you add **functionality** to an interface without changing it?
- Static methods are the canonical solution

```
public static class MyExtensions
{
        public static int ToInt(this string s)
        {
                return int.Parse(s);
        }
}
```

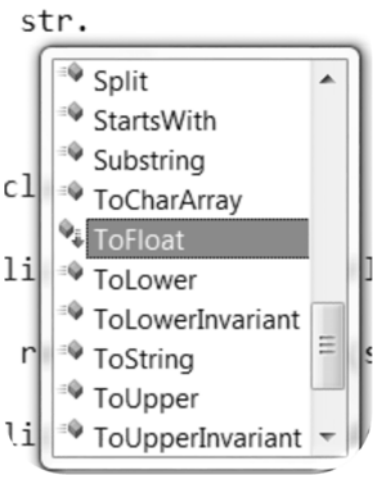

What's The Difference?

- Discoverability
  - Intellisense support
- Natural (fluent) syntax

```
int i = MyExtensions.ToInt(s);
```



```
int i = s.ToInt();
```



What is a Design Patterns?

- **Reusable solutions** to **recurring problems**
- A Pattern is
  - **not original.** It was not invented by its author (nor by anyone)
  - Has been **tested** at least three times
  - **not a formula** (cut/paste, base-class etc)
  - Patterns are a language

Patterns in .NET

- **Language-independent**
- Implementation relies on language capabilities:
  - Inheritance
  - Reference
  - Interface
  - Etc.

What is a Principle?

- Unlike Design Patterns, Principle is more **generic way of thinking** and **addressing problems**.
  - For example: **Keep It Simple**
- Principles are **guidelines** which can guide you **into better practices**.
- You may **address principle by using Design Patterns**.

Why learning Design Patterns & Principles?

- **Reduce** developments and design **effort** by **reusing well tested patterns**.
- **Common vocabulary** for **developers** and **architects**.

Good Design Principles

- Keep it small (KISS)
- Single Responsibility Principle (**S**OLID)
- Open Close Principle (S**O**LID) - Extensibility
- Other Principals Goals
  - Changeable \ Flexibility
  - Portability
  - Scalability
  - Security
  - Reuse
  - Maintainability
- Don’t forget about efficiency

Classification

- Design patterns are originally grouped into three **categories**:
  - **Creational** Patterns: creation of objects
  - **Structural** Patterns: composition of objects
  - **Behavioral** Patterns: communication, rule and flow control.

Questions

SELA

Module 02 - C# 6 - 8

.NET Design Patterns

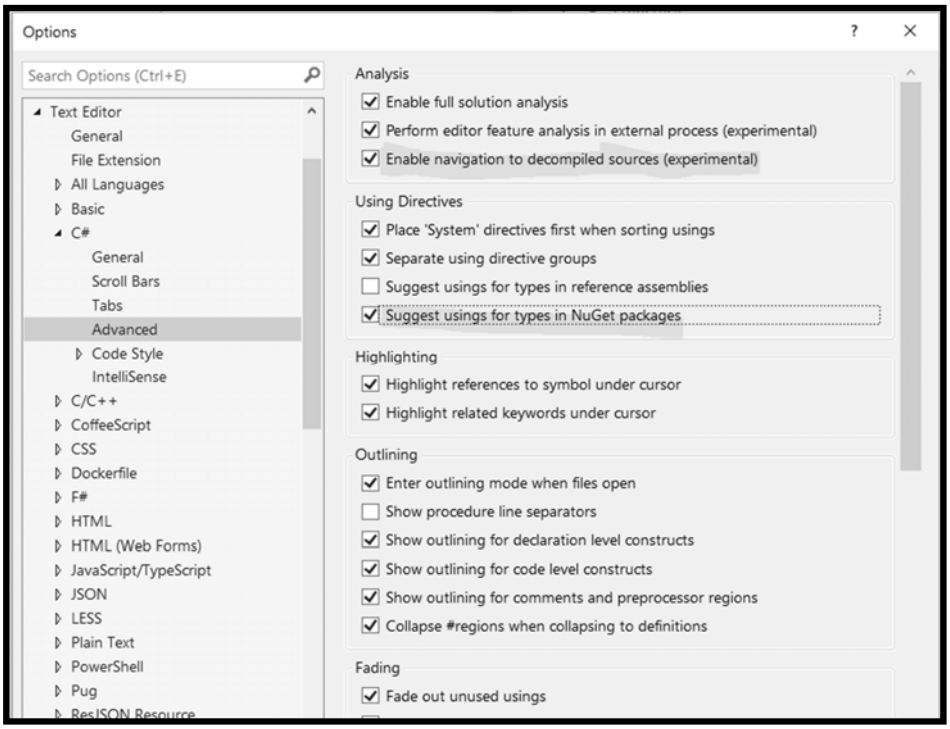
C# 6 - 8

https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8

https://www.youtube.com/watch?v=YHWCFeE2L-4
https://www.youtube.com/watch?v=5ju2MuqKf_8
https://www.youtube.com/watch?v=KFqrp4KSxio
https://www.youtube.com/watch?v=02lnO3koCq4
https://www.youtube.com/watch?v=wHqD7a4wkDI
https://www.youtube.com/watch?v=qZHwvej6xkA

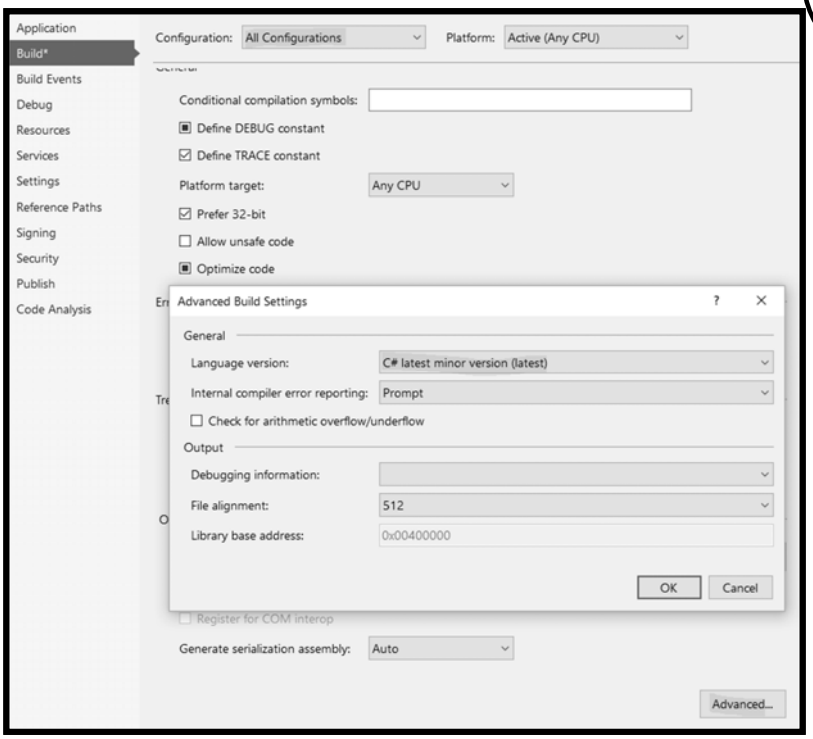
https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-6
https://blogs.msdn.microsoft.com/dotnet/2016/08/24/whats-new-in-csharp-7-0/
https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7
https://www.youtube.com/watch?v=Bj2FukRm6Ok
http://www.dotnetcurry.com/csharp/1401/csharp-7-1-csharp-8-new-features
https://msdn.microsoft.com/en-us/magazine/mt829270.aspx
https://devblogs.microsoft.com/dotnet/building-c-8-0/

Tune Visual Studio (2017)



Tune Project Build

C# 7.1



C# 6

Property, Method, String

```
class Foo
{
    public Foo(string sign)
    {
        Sign = sign;
    }

    public DateTime Start { get; } = DateTime.UtcNow;
    public string Sign { get; }

    public string FromStart =>
        $"{Sign} {DateTime.UtcNow - Start} {Sign}";
}
```

Read-only auto-properties

Auto-Property Initializers

Expression-bodied function members

String Interpolation

Index Initializers

```
private Dictionary<int, string> webErrors = new
Dictionary<int, string>
{
    [404] = "Page not Found",
    [302] = "Page moved, but left a forwarding address.",
    [500] = "The web server can't come out to play today."
};
```

Initializers

Using Static

```
using static System.Math;

namespace Sela.Samples
{
    class Foo
    {
        public int Calc(int a, int b) => Max(Abs(a), Abs(b));

    }
}
```

Don't have to repeatedly typing the class name

Null-conditional operators

```
string first = person?.FirstName ?? "Unspecified";
```

Don't have to check if null before invoke

Exception Filters

```
public static async Task<string> MakeRequest(string url)
{
    using (var client = new HttpClient())
    {
        try
        {
            var responseText = await client.GetStringAsync(url);
            return responseText;
        }
        catch (HttpRequestException e)
               when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
    }
}
```

Better catch resolution

Expressions

```
public void Exec()
{
    try
    {
        CallSomething();
    }
    catch (Exception ex)
    {
        Trace.WriteLine(
               $"Error at {nameof(Foo)} on {nameof(Exec)}");
    }
}
```

nameof: refactoring proof

Await in Catch and Finally blocks

```
public static async Task RunAsync()
{
    try
    {
        await ExecAsync();
    }
    catch (Exception e)
    {
        string log = await FormatErrorAsync(e);
        Trace.WriteLine(log);
    }
    finally
    {
        await LogMethodExitAsync();
    }
}
```

Can await at catch and finally

C# 7

More Expression-Bodied Members

```
class Foo
{
    public Foo(string label) => this.Label = label;

    ~Foo() => Console.Error.WriteLine("Finalized!");

    private string label;

    public string Label
    {
        get => label;
        set => this.label = value ?? "Default label";
    }
}
```

constructor

Finalizer

Get / Set

Throw Expressions

```
private Config config = LoadOrDefault() ??
    throw new InvalidOperationException);

public string Name
{
    get => name;
    set => name = value ??
            throw new ArgumentNullException();
}
```

Numeric Literal Syntax Improvements

```
public const int Sixteen = 0b0001_0000;
public const int ThirtyTwo = 0b0010_0000;
public const int SixtyFour = 0b0100_0000;
public const int OneHundredTwentyEight = 0b1000_0000;
```

```
public const long BillionsAndBillions = 100_000_000_000;
```

```
public const double MagicFactor = 6.022_140_857_747_474e23;
public const decimal GoldenRatio = 1.618_033_988_749_894_84M;
```

Out Variables

Inline declaration

```
if (int.TryParse(input, out int result))
    WriteLine(result);
else
    WriteLine("Could not parse input");
```

Pattern Matching

```
public int CalcArea(IShape value)
{
    int area = 0;

    if (value is IRect rect)
        area = rect.Width * rect.Height;
    else if (value is ITriangle triangle)
        area = triangle.Base * triangle.Height / 2;

    return area;
}
```

Inline declaration

Pattern Matching

```
public int CalcArea(IShape value)
{
    int area = 0;

    switch (value)
    {
        case IRect rect:
            area = rect.Width * rect.Height;
            break;
        case ITriangle triangle:
            area = triangle.Base * triangle.Height / 2;
            break;
    }

    return area;
}
```

Inline declaration

Pattern Matching (C# 8)

```
CalcArea(IShape value)
{
  var area = value switch
  {
    IRect rect => rect.Width * rect.Height,
    ITriangle triangle => triangle.Base * triangle.Height / 2,
    _ => throw new NotSupportedException()
  };

  return area;
}
```

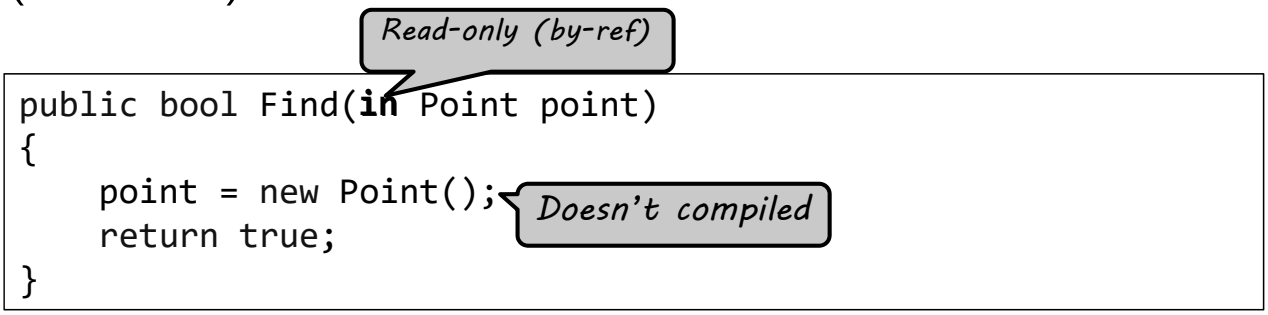
Pattern Matching (C# 8)

```
public decimal GetTaxFactor(IAddress location)
{
    return location switch
            {
                { Country: "ISR" } => 0.17M,
                { Country: "US" } => location switch
                {
                    { State: "WA" } => 0.06M,
                    { State: "MN" } => 0.75M,
                    { State: "MI" } => 0.05M,
                     _ => 0M
                },
                 _ => throw new NotSupportedException()
            };
}
```

Pattern Matching (C# 8)

```
public string GetQuadrant((int x, int y) point)
{
    return point switch
    {
        (0, 0) => "origin",
        (int x, int y) when x > 0 && y > 0 => "Quadrant 1",
        (int x, int y) when x < 0 && y > 0 => "Quadrant 2",
        (int x, int y) when x < 0 && y < 0 => "Quadrant 3",
        (int x, int y) when x > 0 && y < 0 => "Quadrant 4",
        _ => "unknown"
    };
}
```

Reference Semantics With Value Types (C# 7.2)



Ref Locals and Returns

```
class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511};

    public ref int Find(int[] matrix, Func<int, bool> predicate)
    {
      for (int i = 0; i < matrix.Length; i++)
      {
        if (predicate(matrix[i]))
            return ref matrix[i];
      }
      throw new InvalidOperationException("Not found");
    }
}
```

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/ref-returns

Ref Locals and Returns

```
public ref int Find(int[] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.Length; i++)
    {
        if (predicate(matrix[i]))
            return ref matrix[i];
    }
    throw new InvalidOperationException("Not found");
}
```

By value call

```
int i = Find(arr, x => x % 2 == 0);
Console.WriteLine($"{i}, {arr[1]}");
i = 10;
Console.WriteLine($"{i}, {arr[1]}");
```

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/ref-returns

Ref Locals and Returns

```
public ref int Find(int[] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.Length; i++)
    {
        if (predicate(matrix[i]))
            return ref matrix[i];
    }
    throw new InvalidOperationException("Not found");
}
```

By ref call

```
ref int i = Find(arr, x => x % 2 == 0);
Console.WriteLine($"{i}, {arr[1]}");
i = 10;
Console.WriteLine($"{i}, {arr[1]}");
```

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/ref-returns

Ref readonly returns (C# 7.2)

```
public readonly ref int Find(int[] matrix, Func<int, bool> p)
{
    for (int i = 0; i < matrix.Length; i++)
    {
        if (p(matrix[i]))
            return ref matrix[i];
    }
    throw new InvalidOperationException("Not found");
}
```

Doesn't compiled

```
ref int i = ref Find(arr, x => x % 2 == 0);
Console.WriteLine($"{i}, {arr[1]}");
i = 10;
Console.WriteLine($"{i}, {arr[1]}");
```

Default literal expressions (C# 7.1)

```
void Execute(CancellationToken duration =
                                default(CancellationToken))
{
    …
}
```

```
void Execute(CancellationToken duration = default)
{
    …
}
```

Reference semantics with Value Types (C# 7.2)

```
// indicate that a struct is immutable and
// should be passed as an in parameter
public readonly struct Point
{
    public Point(int x, int y)
    {
        Y = y;
        X = x;
    }

    public int X { get; }
    public int Y { get; }
}
```

Value Tuples

```
(int Value, Exception Error) result1 = TryParse("42");
(int Item, Exception Fault) result2 = TryParse("42");
// Deconstruct
(int value1, Exception error1) = TryParse("42");
(var value2, var error2) = TryParse("42");
var (value3, error3) = TryParse("42");
// Discards
(int value4, _) = TryParse("42");
```

```
public static (int Value, Exception Error) TryParse(string data)
{
    if (int.TryParse(data, out int i))
        return (i, default); // C# 7.1: Default literal expressions
    return (default, new InvalidCastException());
}
```

```
var alphabetStart = (Alpha: "a", Beta: "b");
(string First, string Second) fstSnd = (Alpha: "a", Beta: "b");
```

Inferred tuple element names (C# 7.1)

```
int count = 5;
string label = "Colors used in the map";

// element names are "count" and "label"
var pair = (count, label);
```

Deconstruct

```
public class Point
{
    public Point(double x, double y) { X = x; Y = y; }

    public double X { get; }
    public double Y { get; }

    public void Deconstruct(out double x, out double y)
    {
        x = X;
        y = Y;
    }
}
```

```
Point p = new Point(3.14, 2.71);
```

```
(double x, double y) = new Point(3.14, 2.71);
```

Local functions

```
public void OnMouseMove(object sender, MouseMoveEventArgs e)
{
    Task fireFirget = OnMouseMoveAsync();

    async Task OnMouseMoveAsync()
    {
        var map = await GetMapAsync(e.X, e.Y);
        await QueueMapAsync(map);
    }
}
```

Use local parameters

Generalized async return types

Struct

```
private async ValueTask<int> LoadAsync(string key)
{
    if (_cache.TryGet(key, out int value))
        return value;
    int item = await _loader.LoadAsync(key);
    _cache.TryAdd(key, item);
    return item;
}
```

Private protected access modifier (C# 7.2)

```
public class MagicBase
{
    private protected const int InterbalNumber = 10;
}
```

```
public class Magic : MagicBase
{
    public Magic()
    {
        Trace.WriteLine($"The number is {InterbalNumber}");
    }
}
```

Not accessible form external assemblies

Span (.NET Core 2.1)

```
int[] span =  { 1, 2, 3 };
Span<int> spanTail = span.Slice(1);
```

https://medium.com/@bnayae/net-span-t-better-performance-29b182b19dce

Stackalloc (C# 7.3)

```
Span<int> span = stackalloc[] { 1, 2, 3 };// stack allocation
```

https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7-3#ref-local-variables-may-be-reassigned

Stackalloc (C# 8)

```
Span<int> span = stackalloc[] { 1, 2, 3 };// stack allocation
```

https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7-3#ref-local-variables-may-be-reassigned

Asynchronous Stream (C# 8)

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator(
          CancellationToken cancellationToken = default);
}

public interface IAsyncEnumerator<out T> : IAsyncDisposable
{
    T Current { get; }
    ValueTask<bool> MoveNextAsync();
}
```

https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/generate-consume-asynchronous-stream#async-streams-provide-a-better-way

Asynchronous Stream (C# 8)

```
async IAsyncEnumerable<int> GenerateSequence()
{
    for (int i = 0; i < 20; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

```
await foreach (var number in GenerateSequence())
{
        Console.WriteLine(number);
}
```

https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/generate-consume-asynchronous-stream#async-streams-provide-a-better-way

Asynchronous Disposable (C# 8)

```
public interface IAsyncDisposable
{
    ValueTask DisposeAsync();
}
```

https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/generate-consume-asynchronous-stream#async-streams-provide-a-better-way

Asynchronous Disposable (C# 8)

```
class MyResource : IAsyncDisposable
{
    public async ValueTask DisposeAsync()
    {
        await Task.Delay(200).ConfigureAwait(false);
        Console.WriteLine("Released");
    }
}
```

```
await using (var r = new MyResource())
{
    Console.WriteLine("Executing");
}
Console.WriteLine("Disposed");
```

https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/generate-consume-asynchronous-stream#async-streams-provide-a-better-way

Nullable Reference Types

- Turning On

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <LangVersion>8.0</LangVersion>
    <NullableContextOptions>enable</NullableContextOptions>
    <!--<TreatWarningsAsErrors>true</TreatWarningsAsErrors>-->
  </PropertyGroup>

</Project>
```

https://www.youtube.com/watch?v=HW_FnmnDlGQ

Nullable Reference Types

```
public class Person
{
    // either should initialized or use string?
    public string FirstName { get; set; }
    public string[] MiddleNames { get; set; }
    public string LastName { get; set; }
}
```

```
var p = new Person { FirstName = "Hana", LastName = "Green"};
WriteLine($"{p.FirstName}, {p.MiddleNames[0]}, {p.LastName} ");
```

https://www.youtube.com/watch?v=HW_FnmnDlGQ

Indices and Ranges

- Provide syntax for **subranges** in an **array**, **Span**<T>, or **ReadOnlySpan**<T>.

```
var words = new string[]
{
    // index from start    index from end
    "A",       // 0                   ^9
    "B",       // 1                   ^8
    "C",       // 2                   ^7
    "D",       // 3                   ^6
};
```

```
words[^1]        // D
words[1..3]      // B, C
words[1..^1]     // B, C
```

https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8#indices-and-ranges

SELA

Module 03 - Fundamental Design Patterns

.NET Design Patterns

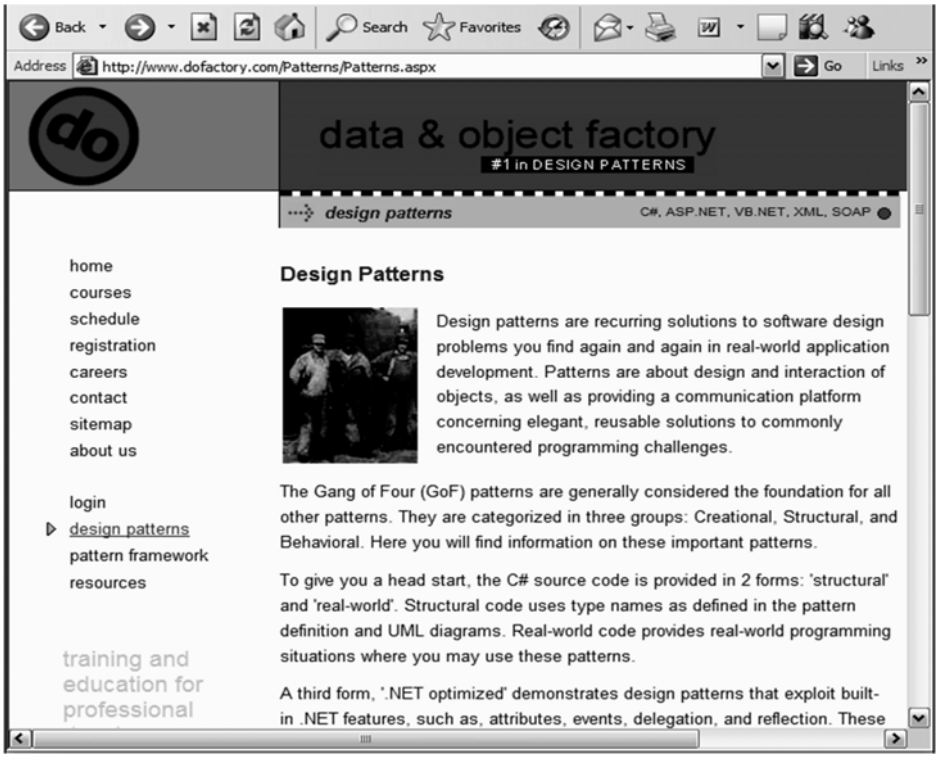
In This Chapter

- We will discover some fundamental ideas on which many design pattern are based
- For example:

  The abstraction that an interface or an abstract class provides

  The simplicity that an immutable objects provide when an object is may be shared among many other objects etc

GoF - http://www.dofactory.com/Patterns/Patterns.aspx



Fundamental Design Patterns

- Delegation – Delegates data to another object. Use another object instead of inheriting from it
- Interface – Reference to functionality
- Marker Interface – Enables asking is on an object
- Immutable – Every attempt to change object state will create a new instance with the change

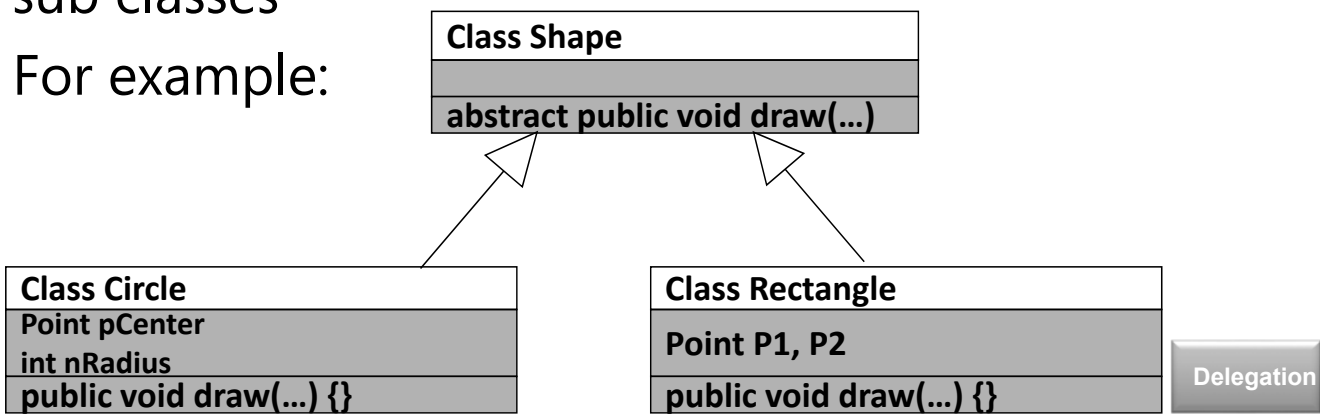
SELA

Delegation Pattern

Or…When not to use inheritance

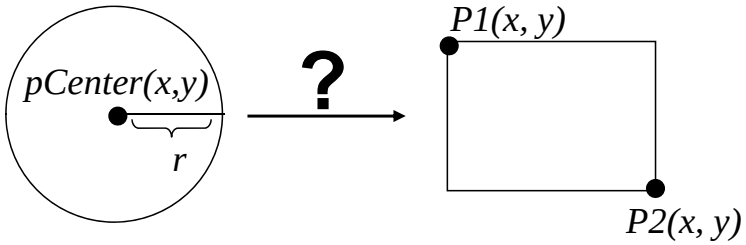
Delegation vs. Inheritance

- **Inheritance** is a useful way to **extend** a **functionality** of a class
- Using inheritance, we can create a general, base type and extend, or specialize its behaviors in sub classes
- For example:



Delegation vs. Inheritance – The Problem

- What about **converting** a Circle to a Rectangle?
- Can a shape be a **circle and a rectangle** at the **same time**?



- Do we have to think about such cases?

Delegation

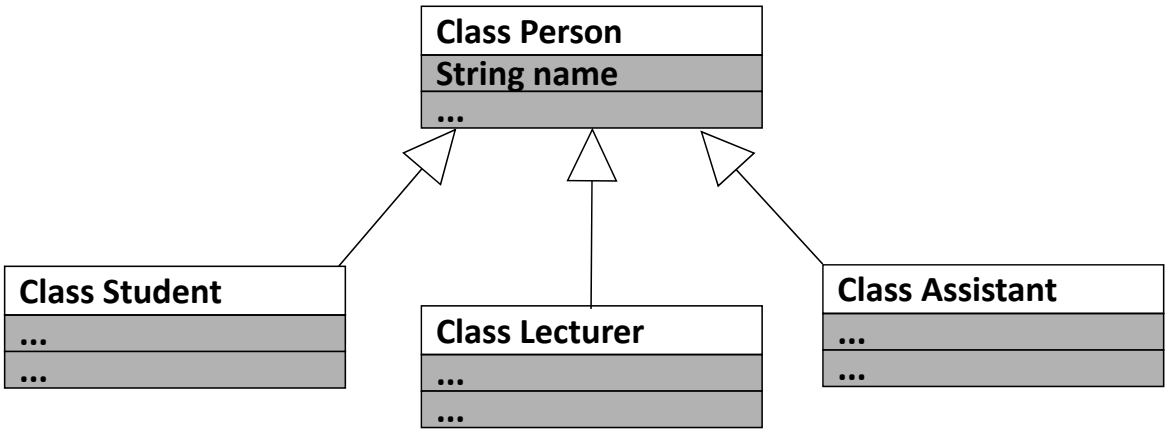
OO College

- College Management
  - Student
  - Lecturer
  - Assistant
- All of them are hopefully…human beings

Delegation

OO College – Diagram #1

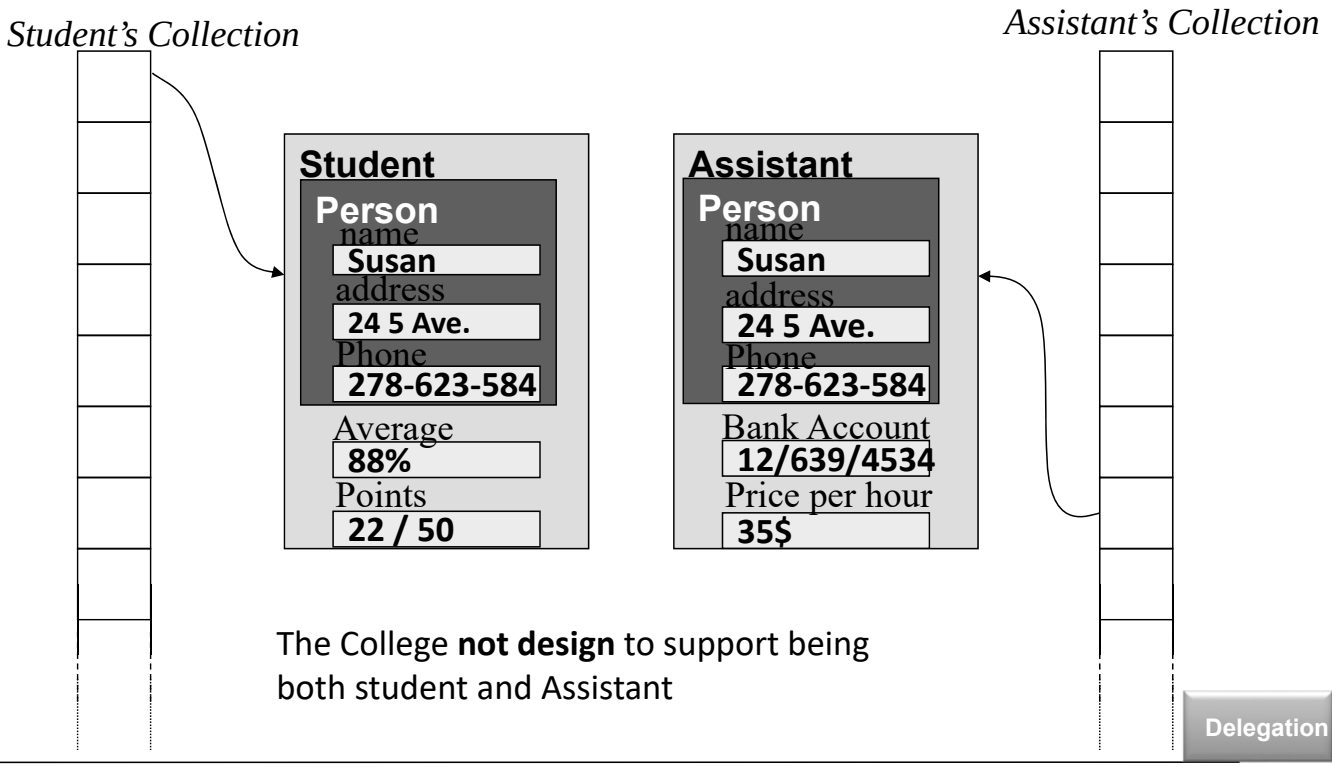
- One may think about the following diagram:



OO College

- New requirements:
  - BA **graduates can serve as assistants**
- For Example:
  - **Student "Susan"** is a BA **graduate**.
  - …working on her Master's degree
  - …and **assisting** Dr. Cohen in the course

OO College – cont'd

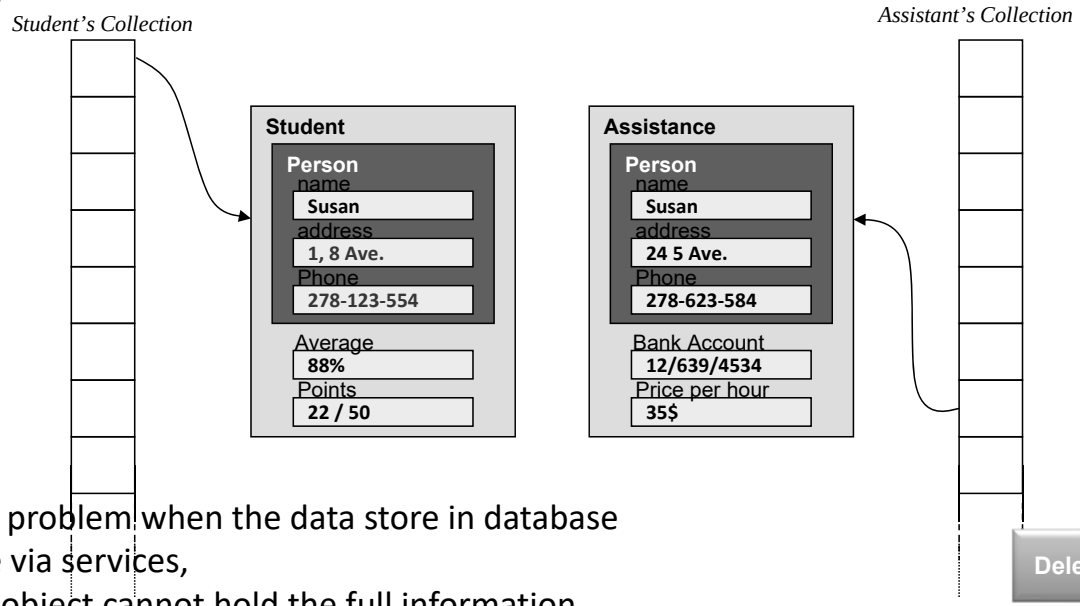


The College **not design** to support being both student and Assistant

Delegation

OO Collage – cont'd

- What will happen if Susan changes her address and updates only the secretary responsible for student affairs?



Not necessary problem when the data store in database and accessible via services,
but still single object cannot hold the full information

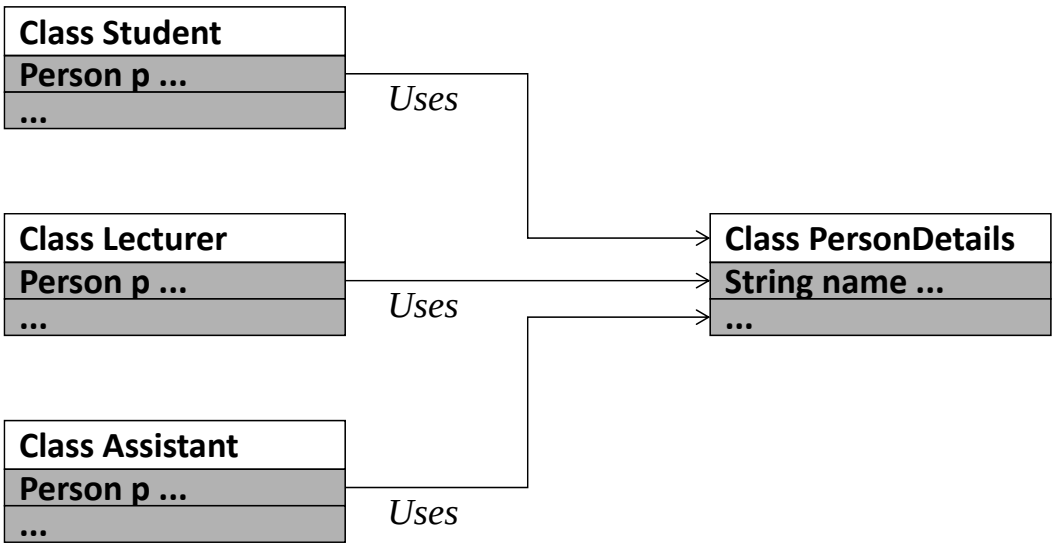
Delegation

Designing an Inheritance Tree

- You should design the **inheritance** tree around the **most stable relationship** in your domain
- Inheritance is the **solution for** an **IS-A** relationship, although it **doesn't fit** into ~~**IS-A-ROLE-PLAYED-BY**~~ relationship
- In our case, it is preferable to find a better solution

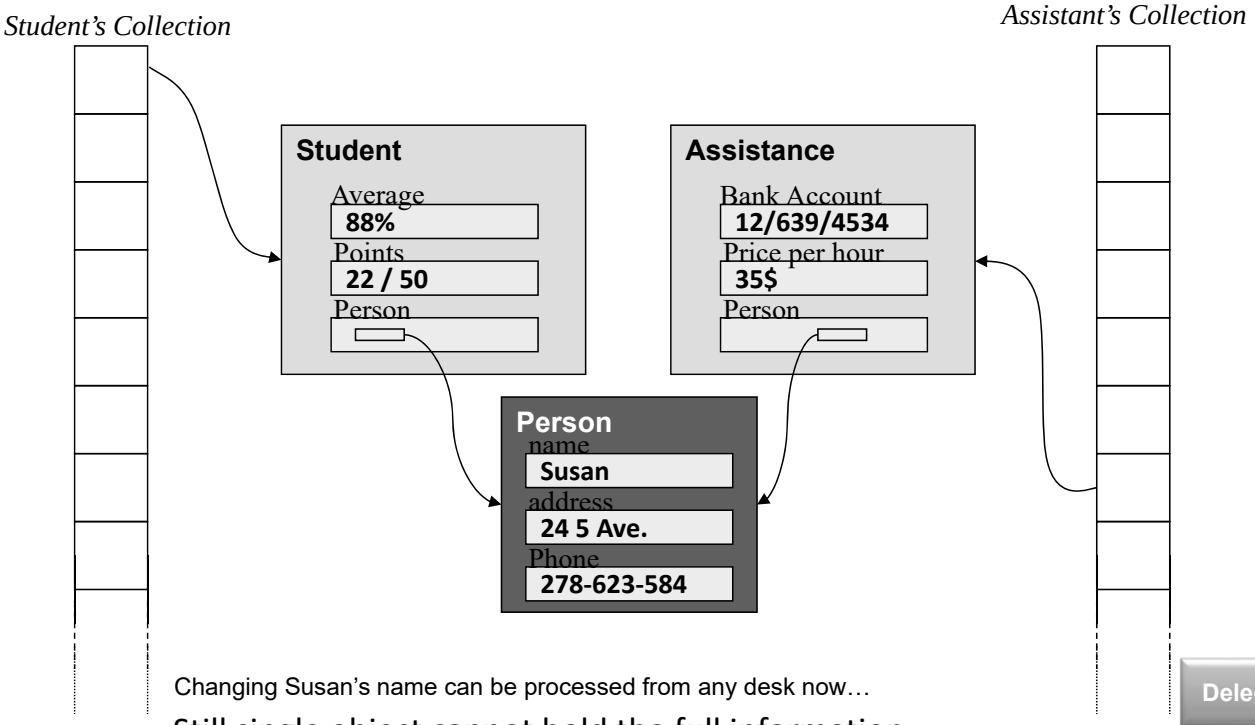
Delegation

OO College – Diagram #2



Delegation

OO College – Using Delegation



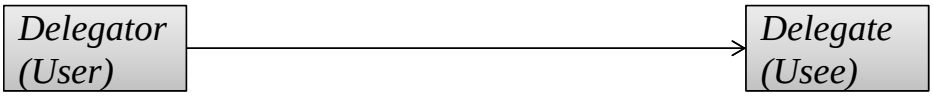
Changing Susan's name can be processed from any desk now…

Still single object cannot hold the full information

Delegation

Consequences

- Advantages:
  - Delegation enables objects to be shared
  - Behavior is composed and modified at runtime
- Disadvantages:
  - Less structured than inheritance
  - Less apparent than inheritance



Delegation

3rd option

Person + Person's Roles.

```
class PersonBase
{
    public string Name { get; }
    public string Address { get; }
    public string Phone { get; }
}

class Person : PersonBase
{
    IStudentRole Student { get; } // can be null
    IAssistanceRole Assistanc { get; } // can be null
}
```

Delegation

4th option

Person + Person's Roles

Pattern Matching could help.

```
class PersonBase
{
    public string Name { get; }
    public string Address { get; }
    public string Phone { get; }
}

class Person : PersonBase
{
    IRole[] Roles { get; }
}
```

Delegation

5th option

Pattern Matching could help.

```
class Student : PersonBase, IStudentRole
{
}

class Assistance : PersonBase, IAssistanceRole
{
}

class AssistStudent : PersonBase, IStudentRole, IAssistanceRole
{
}
```

Delegation

Interface Pattern

The Interface pattern

- Using **well defined contract** (instead of well-known implementation) in order to communicate across boundary.
- Can be implement either by .NET Delegate or .NET Interfaces
- Filter (LINQ in general) are good example.
- The pattern can be apply to **services** (with known signature) on **micro-services** scenario

Interface

SELA

Immutable

Immutable

- Before diving into this pattern, observe the following:

```
public static void Main (string[] args)
{
    string Original = "Hello All";
    Original.Trim();
    Original.Substring(6);
    Original.ToUpper();
    Console.WriteLine(Original);
}
```

Q What will be the output? **The answer is: "Hello All"**

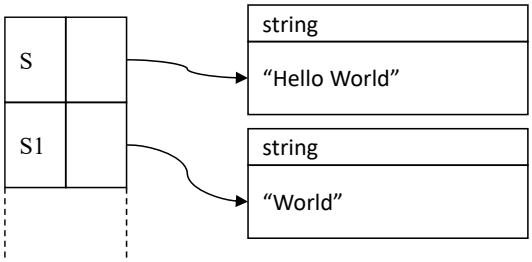
Q How many instances of class string where created? **4**

Immutable

Immutable - String

- An instance of class `String` **cannot be changed**.
- All String **methods** return **new memory allocation**.

```
string s = "Hello world"
string s1 = s.Substring(5);
```



Why immutability?

- Implicit **thread safety** (synchronization-free)
- **Snapshot**
- **Read-only** semantics
- **Functional-programming** friendly
- Easy **undo** or **re-play**

Immutable

Create Immutable Class: Traditional

```
public class ImmutablePerson
{
    public ImmutablePerson(string name, DateTime dateOfBirth)
    {
        Dame = name;
        DateOfBirth = dateOfBirth;
    }

    public DateTimeOffset CreatedOn { get; } = DateTimeOffset.UtcNow;
    public string Name { get; }
    public DateTime DateOfBirth { get; }
}
```

Override GetHashCode and equitablity is good idea

Immutable

Immutable Collections

- Immutable**Stack**<T>
- Immutable**Queue**<T>
- Immutable**List**<T>
- Immutable**Array**<T>
- Immutable**HashSet**<T>
- Immutable**SortedSet**<T>
- Immutable**Dictionary**<K, V>
- Immutable**SortedDictionary**<K, V>

Immutable

Fluent API

- Most of the frequently used APIs are fluent

```
// immutable collection starts with Empty (not ctor)
var colFluent = ImmutableList<int>.Empty
    .Add(1)
    .Add(2);

// immutable collection starts with Empty (not ctor)
var col = ImmutableList<int>.Empty;
var col1 = col.Add(1);
var col2 = col1.Add(2);
```

Immutable

Immutable.Add? Is it possible?

- How can an immutable object use the Add API?
- Well, it's not much different with **strings**

```
string s = " Hello "
    .Trim()
    .Replace("l", "*")
    .Remove(0, 1);
```

```
// immutable collection starts with Empty (not ctor)
var colFluent = ImmutableList<int>.Empty
    .Add(1)
    .Add(2);
```

Immutable

Immutability and memory allocation

- Naïve implementation of immutable collections can become a memory problem.
- Similar to **string**, immutable collections may produce a lot of garbage

```
var colA = ImmutableList<int>.Empty;
var colB = colA.Add(1); // new allocation
var colC = colB.Add(2); // new allocation

var colD = ImmutableList<int>.Empty;
// single allocation
var colE = colD.AddRange(new[] { 11, 12, 13, 14, 15 });
```

Immutable

Immutable Objects and Threads

- In a **multithreaded** environment, there is a need to **synchronize** access to **shared memory**.
- Due to the fact that **immutable** objects' data **cannot be changed** after construction, the usage of immutable object can **avoid the overhead of synchronizing** access.

Immutable

Marker Interface

Or…Does a type support a specific behavior?

What is a Marker Interface?

- **Tagging** object without enforcing functionality
- Can be done:
  - Empty Interface
  - Attribute
- May benefit from Pattern Matching
- Main **disadvantage**: relay on **convention rather than contract**

Marker Interface

Using Interface

```
interface ILoadable { }

public class MainEntry : ILoadable
{
    public void Main()
    {
        // by convention
    }
}
```

Using Attribute

```
class LoadableAttribute: Attribute { }

[Loadable]
public class MainEntry
{
    public void Main()
    {
        // by convention
    }
}
```

Questions

SELA

Module 04 - Creational Patterns

.NET Design Patterns

In this Chapter

- Deal with different pattern of **creation of instances**

Patterns That Will Be Discussed in This Chapter

- Singleton – Only one instance of the class can be created
- Factory Method – Creates an instance of several derived classes
- Abstract Factory – Helps you to define a family of objects without specifying their concert classes
- Builder – Separates the construction of complex object from its representation so that the same construction process can create different representations
- Prototype – Creates new objects by copying existing instances from a catalogue
- IoC – Inversion of control

Singleton

Or…I am unique!

The Singleton Pattern

- Ensure that **only single instance** of a type **is created**

Singleton

Singleton – A Simple Example

```
public sealed class Singleton
{
    private Singleton() { }

    public static readonly Singleton Instance = new Singleton();

    // TODO: add the class code
}
```

Singleton concider a bad pattern: not testable friendly.
IoC frameworks / Dependencies Injection will address it better.

Singleton

Singleton – Disadvantage

- Hard to **test** (hard to mock)
- Hard to trace and monitor **exception** during the creation of the single instance.

Singleton

Lazy Initialization (.net 4.0)

- **Lazy initialization** is primary used to **improve performance** (faster start time) and **reduce** program **memory** requirements

```
public sealed class Singleton
{
    private Singleton() { }

    private readonly static Lazy<Singleton> _instance =
                        new Lazy<Singleton>(new Singleton());
    public static Singleton Instance => _instance.Value;

    // TODO: add the class code
}
```

SELA

Factory Method

What is a Factory Method?

- Indirect creation of object
- Motivation
  - Tune the object creation to
    - Environment (prod, qa, dev)
    - Installed components (driver, db, etc.)

What is a Factory Method? (Cont.)

- Examples
  - Database connection:
    - According to the environment:
      - Dev
      - QA
      - Prod
  - Menu options
    - According to the user status:
      - Admin
      - Contributer
      - Reader

Factory Method

SELA

Abstract Factory

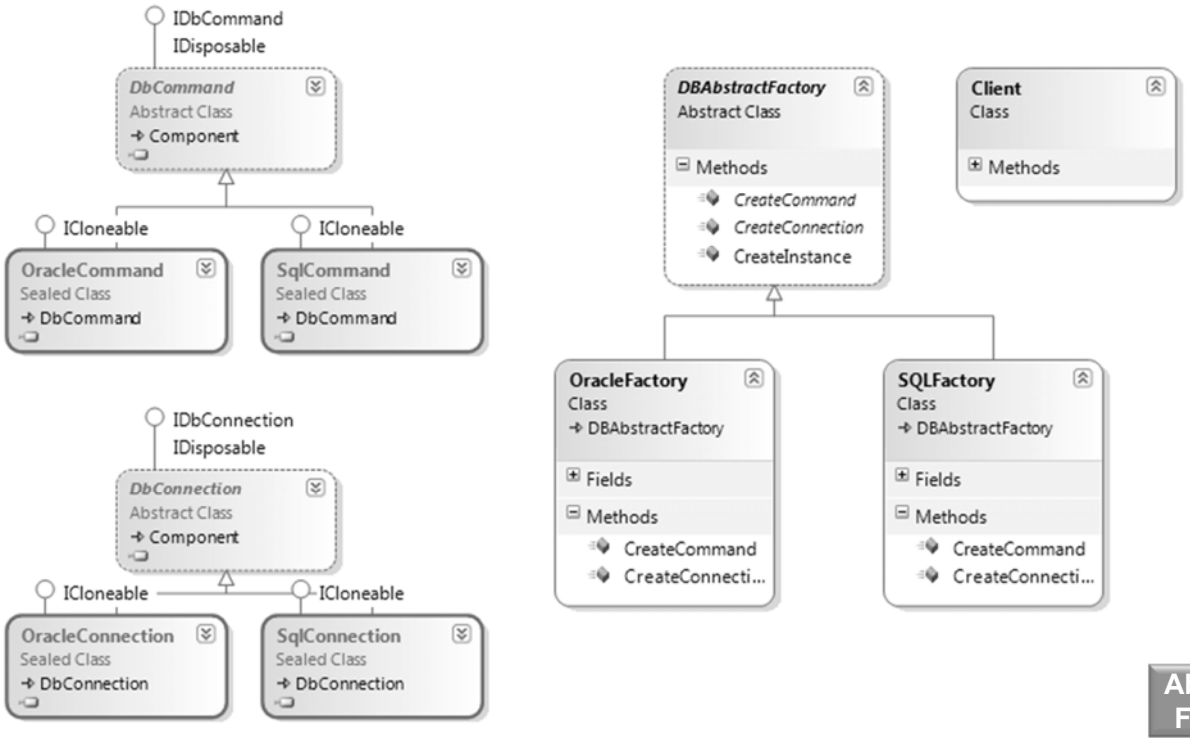
Encapsulate a group of individual factories

Abstract Factory

- Provides a way to **encapsulate a group of individual factories** that have a common theme.

Abstract Factory

DBAbsFactory Class Diagram



Abstract Factory

Factory Client

```
public static void Main()
        {
            DBAbstractFactory factory =
   DBAbstractFactory.CreateInstance();

            DbConnection connection = factory.CreateConnection();
            DbCommand command = factory.CreateCommand();
            command.CommandType = System.Data.CommandType.Text;
            command.CommandText = "Delete from Entity e where
   e.ID = 1";

            connection.Open();
            command.ExecuteNonQuery();
            connection.Close();
        }
```

Abstract Factory

Builder

The Builder Pattern

- Simplify the process of building complex objects
  - Chose **how** to build an object
  - Potentially by **multi-step** building.
  - Can build complex object with different **tuning options**.

Builder

String Builder

- Building string

```
var builder = new StringBuilder(origin);
builder.Replace(“a”, *);
builder.Remove(“x”)

var data = builder.ToString(); // immutable
```

Mutable builder

- Similar to **StringBulder**, immutable collections can have a **builder**

```
var col = ImmutableList<int>.Empty;

// mutable (produce less non garbage)
var builder = col.ToBuilder();
for (int i = 0; i < 100; i++)
{
    builder.Add(i);
}

var col1 = builder.ToImmutable(); // immutable
```

Fluent API

- Used to **chain operation** usually by returning **interface** for **next possible operations**

```
var builder = new StringBuilder(origin);
var data = builder.Replace("a", *)
                  .Remove("x")
                  .ToString();
```

Adaptive builder

- Can you create a **builder which don't repeat itself?**
- how to avoid building same part twice?
- Vehicle Builder
  - Frame
  - Engine
  - Wheels
  - Doors

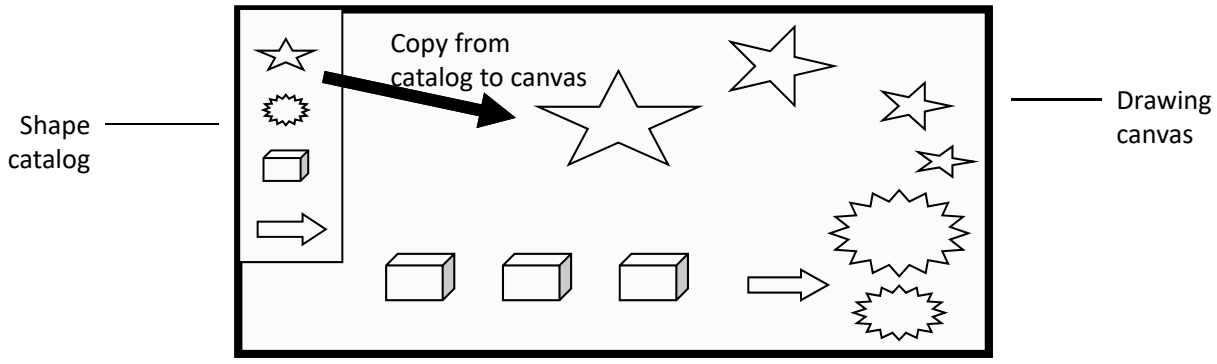
The Prototype Pattern

Prototype = Clone

- Create object based on existing instance
- Consider the problem of **shallow** vs **deep cloning**
- .NET deep cloning is done via Serialization (inefficient)
- The **motivation** may be **avoid heavy construction**

Prototype

Prototype Factory - Motivation



- Consider to hide mutability of the object by creation method (instead of simple clone)
- For example:
  - Star cloned = star.Resize(100,100)

Prototype

SELA

Module 05 - **IoC (Inversion of Control)**

.NET Design Patterns

IoC (Inversion of Control)

also refer as

DI (Dependency Injection) or

Service Locator

https://www.youtube.com/watch?v=QtDTfn8YxXg

IoC: Factory on steroids

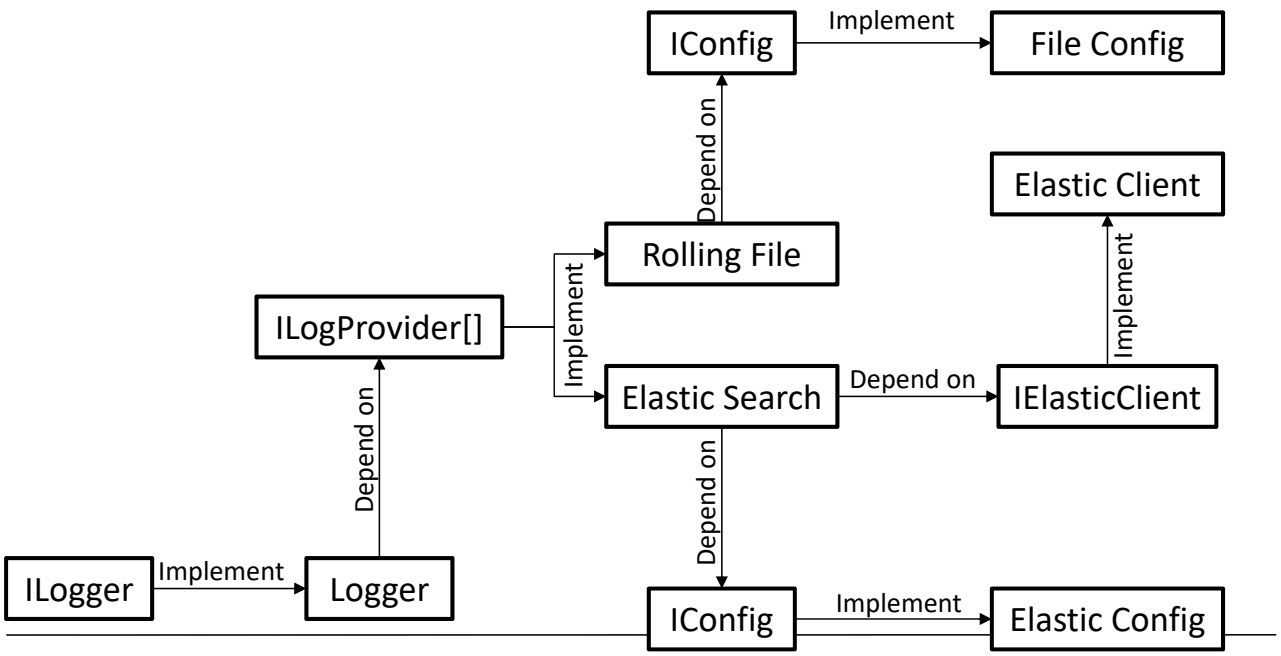
- **IoC** is a design principle which **separate** the **creation** of component **from** it **consumption**.
- **Unlike** the **Factory** pattern, IoC should **resolve nested** component **dependencies**.

- **Increase modularity** and **Extensibility** by **separating** the **component** from its **dependencies**.

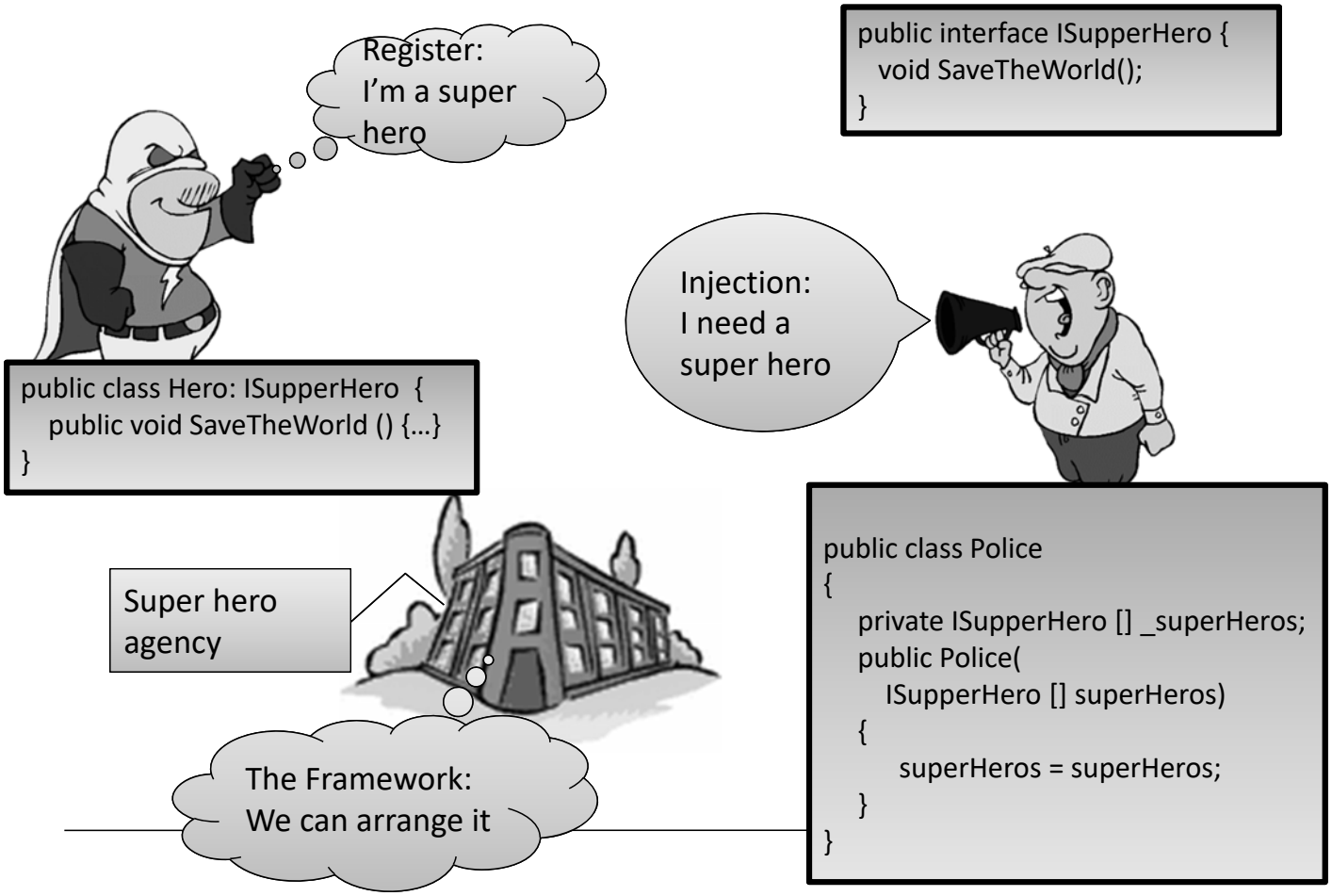
- This is **great** pattern for **testability**.
- Test can **isolate** the **component** under test, from its **dependencies**.

The Problem (Why is it different from Factory?)

- **How to create Logger**
  - When each dependency is having it's own dependencies



DI Concept



Which concerns involved?

- **IoC** can break out into 3 concerns:
- IoC **Framework**: orchestration the composition.
  - Responsible for the dependency matching.
- **Component Code** which can use:
  - **DI** (dependencies Injection): unaware of the IoC framework
    - Normally use Constructor Injection
    - This course will **focus** on **DI**
  - **Service Locator**: partly aware for the IoC framework
    - Resolve dependencies
- **Deployment**: concern about the **registration**

Choosing the IoC framework

- Choosing IoC framework **can be tricky**
- There're lot of them: http://www.palmmedia.de/blog/2011/8/30/ioc-container-benchmark-performance-comparison
  When you choose a framework you should consider
  what's important to you.
  - **Not necessarily** which one have **more feature** or **execute faster**.

Autofac

- On this course we will use Autofac as our IoC of choice

- Modern Open Source framework
- Great Documentation
- Consume via NuGet

Home: https://autofac.org/
GitHub: https://github.com/autofac/Autofac
Documentation: http://autofac.readthedocs.io/en/latest/getting-started/index.html

Constructor Injection

```
public class TheLogic : ILogic
{
    private readonly IConfiguration _config;
    private readonly ILogger _logger;
    public TheLogic(IConfiguration config, ILogger logger)
    {
        _config = config;
        _logger = logger;
    }
    public double Calc(double value)
    {
        var factor = _config.Get<FactorSetting>("Fact");
        _logger.Log(SeverityLevel.Info,
                            value * factor.Value);
        return result;
    }
}
```

Dependency

https://github.com/bnayae/Samples.IoCOverAutofuc

Dependency Definitions

```
public interface IConfiguration
{
        T Get<T>(string key);
}
```

```
public interface ILogger
{
        void Log(SeverityLevel level, object text);
}
```

Test + Mocking technique

- How to **inject dependencies**?
- How to set **dependencies behavior**?
- How to test the **dependencies usage**?

- The **hard way** is to do **custom implementation**.
- **Better way** is using existing **Mocking Framework**.

Mocking frameworks

- Most popular mocking framework will do the job.

- List of popular frameworks:
  - FakeItEasy
  - Moq
  - Rhino Mocks
  - NSubstitute
  - NMock3
  - And other…

Choosing the testing framework

- All the popular testing framework will do the job
  - MsTest
  - NUnit
  - XUnit
  - MbUnit
  - And other

Test: using Moq framework

```
[TestMethod] public void Logic_With30_Test()
{
   var cfg = new Mock<IConfiguration>();
   var log = new Mock<ILogger>();
   cfg.Setup(m => m.Get<FactorSetting>("X"))
       .Returns(new FactorSetting { Value = 2 });

   var logic = new TheLogic(cfg.Object, log.Object);
   double result = logic.Calc(30);

   Assert.AreEqual(60, result);
   log.Verify(m => m.Log(SeverityLevel.Info,
                         It.IsAny<string>()),
                         Times.AtLeast(1));
}
```

Test: using FackItEasy framework

```
[TestMethod] public void Logic_With30_Test()
{
   var cfg = A.Fake<IConfiguration>();
   var log = A.Fake<ILogger>();
   A.CallTo(() => cfg.Get<FactorSetting>("X"))
       .Returns(new FactorSetting { Value = 2 });

   var logic = new TheLogic(cfg, log);
   double result = logic.Calc(30);

   Assert.AreEqual(60, result);
   A.CallTo(() => log.Log(SeverityLevel.Info,
                          A<string>.Ignored))
          .MustHaveHappened(Repeated.AtLeast.Times(1));
}
```

Module (Registration pattern)

- It is **not easy** to **figure** out all registration **at the host**.
- If each **component** will be **responsible** for it **registration** the **host registration** will only be **responsible for** which **component** it want to register.

Module Registration Pattern

```
public class InfraModule: Module
{
	protected override void Load(
		ContainerBuilder builder
	{
		builder.RegisterInstance(new ConfigMode())
				.As<IConfigMode>();

		builder.RegisterType<Logger>()
				.As<ILogger>()
				.As<ILoggerEvents>()
				.SingleInstance();
	}
}
```

https://github.com/bnayae/Samples.IoCOverAutofuc

Module Registration Pattern: Host

```
private static ILogic InitializeIoC()
{
	var builder = new ContainerBuilder();
	builder.RegisterModule<FileConfigProviderModule>();
	builder.RegisterModule<TraceLogProviderModule>();
	builder.RegisterModule<WinEventLogProviderModule>();
	builder.RegisterModule<ComponentsModule>();
	builder.RegisterModule<InfraModule>();
	var container = builder.Build();
	var logic = container.Resolve<ILogic>();
	return logic;
}
```

Specialized (Registration pattern)

- Sometimes you may have multiple implementation for same contract and you may want to get specific kind
- The specialize decision can be done at **Compile-time** or **dynamically** at **Run-time**.

Compile-Time

```
public class Provider : IConfiguration
{
    private readonly ILoggerProvider _logger;
    public FileSettingProvider(
              // Avoid circular dependencies
              [KeyFilter("Primal")]ILoggerProvider logger)
    {
       _logger = logger;
    }
}
```

Compile-Time registration

```
builder.Register(C =>
{
       var log = C.ResolveKeyed<ILoggerProvider>("Primal");
       var instance = new Provider(log);
       return instance;
})
.As<IConfiguration>()
.SingleInstance();
```

Run-Time

When the decision should take at run-time, use **IIndex** with selection aspect and contract.

```
public class Provider : IConfiguration
{
    private readonly string _configFolder;
    public FileSettingProvider(
             IConfigMode cfgMode,
             IIndex<ConfigModes, IFolderSelector> selector)
    {
        var selection = configFolderSelector[cfgMode.Mode];
       _configFolder = selection.Folder;
    }
}
```

Compile-Time registration

```
builder.Register(C =>
{
       var cnfMode = C.Resolve<IConfigMode>();
       var selector = C.Resolve<IIndex<ConfigModes,
                                       IFolderSelector>>();
       var instance = new Provider(cnfMode, selector);
       return instance;
})
.As<IConfiguration>()
.SingleInstance();
```

Best Practice

- **Use module** in order to **break** the **registration** into smaller concern.
- Consider to **Discover** the module dynamically?
- Use **singleton** lifetime
- **Avoid injection** of **to many parameters** (break it, use encapsulation, review the code, ext.)
- Prefer DI (Dependencies Injection) rather Service Locator
- If using **Service locator** expose the **most frequent resolved** component **via extension method**.

Pitfalls

- **Don’t overuse** IoC
  - When a **few class** create **atomic behavior** together one class can create the other using the **new operator**.
  - Don’t use xml file for registration (it’s not maintainable)
  - **Never inject properties** via **xml** (use configuration framework for that kind of injection)
  - **Never use transient lifetime** (use factory in those case).

SELA

Module 06 - Structural Patterns

.NET Design Patterns

In This Chapter

- Focused on object **creation and initialization patterns**
- Patterns of **objects collaboration**

- **Patterns**
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Proxy
  - Flyweight

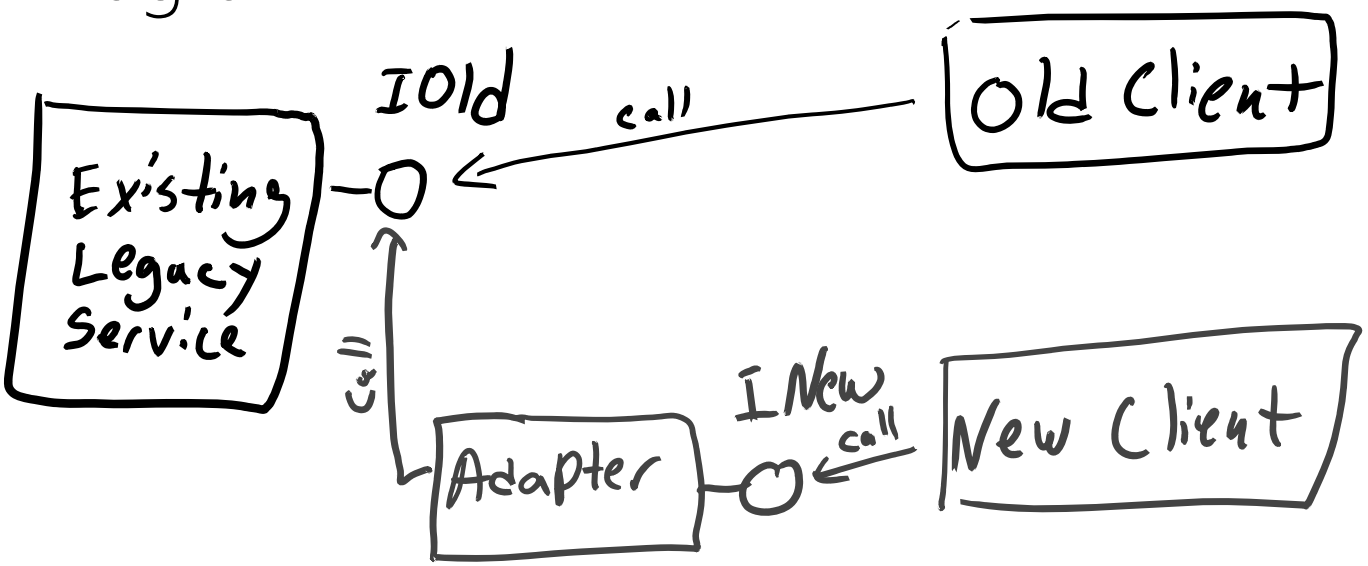
Adapter

https://en.wikipedia.org/wiki/Adapter_pattern

Adapter Definition

- **Abstract** class **interface by another interface** (which will be expected by the client)
- **Solve incompatible interfaces** (for the interface consumer) :
  - **Target**
    - Defines the domain-specific interface that Client uses
  - **Adaptee**
    - Defines an existing interface (which need to be consume)
  - **Adapter**
    - Adapts the Adaptee's interface to the Target interface
  - **Client**
    - Use the Target interface

Diagram



Example: COM Interop

- .NET platform enable you to use existing COM object using COM interop
- Interop assemblies mapping COM object members to equivalent .NET managed members
- Interop assemblies created by visual studio .net
  - COM Interop handle many of the details of working with COM object, such as Interoperability marshaling

Adapter

SELA

Bridge

https://en.wikipedia.org/wiki/Bridge_pattern#C#

Bridge Pattern

- Definition
  - Decouple an abstraction from its implementation so that the two can vary independently
  - Bridge vs. Adapter
    - **Adapter** abstract **existing** code
    - **Bridge** design **upfront**
      - **Dependency Injection** can provide this abstraction

Bridge

Composite

Composite - Definition

- **Partitioning** pattern (of object tree)
- **Unify Group / Single item** operations
- Compose objects into tree structures to represent part-whole hierarchies

https://en.wikipedia.org/wiki/Composite_pattern

Composite

Example

- Authorization
  - Allow / Deny
    can be apply to ‘single user’ or ‘group of users’
- UI Render
  - Page / View / Control: can all be rendered

Composite

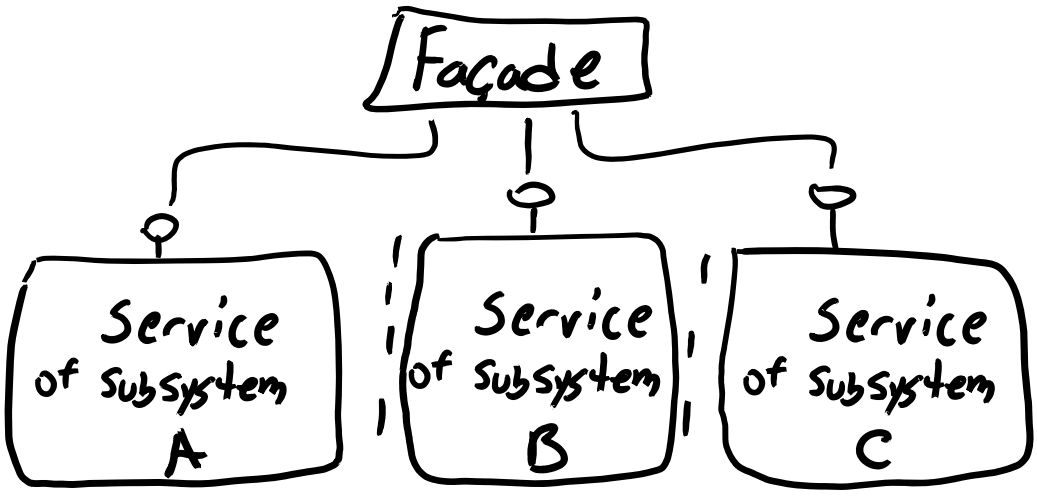
SELA

Façade

Definition & Participants

- **Unified interface** to a set of interfaces in a subsystem
- Makes subsystem **easier to use**

Façade



https://en.wikipedia.org/wiki/Facade_pattern

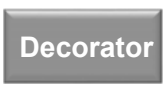
SELA

Decorator

Definition & Participants

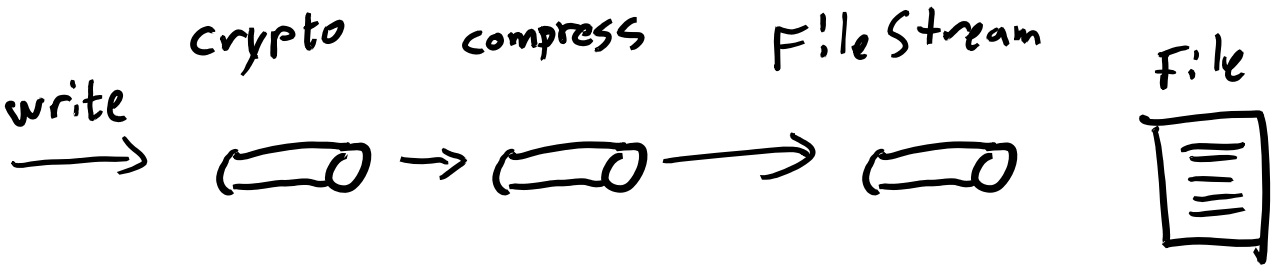
- **Extend object's behavior**
  without changing it interface.
- Useful for applying cross concerns responsibilities
  - Error Handling
  - Metrics
- Enable interception of calls (before and after)
- *Supported by some of the **IoC** framework*

https://en.wikipedia.org/wiki/Decorator_pattern



Example

- .NET Stream
  - CryptoStream
  - GZipStream



AutofacAOP: https://github.com/bnayae/AOPs

.NET - Stream

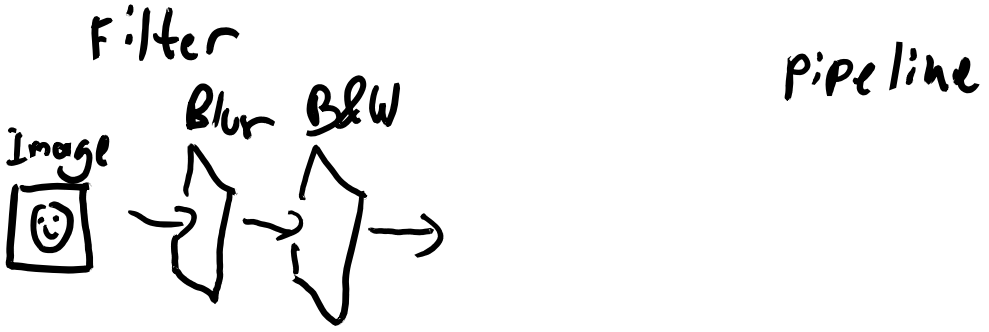
```
FileStream fileStream = new FileStream("temp.txt", FileMode.OpenOrCreate);
GZipStream zipStream = new GZipStream(fileStream, CompressionMode.Compress);
BufferedStream buffeedStream = new BufferedStream(zipStream);

byte[] buffer = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
buffeedStream.Write(buffer, 0, buffer.Length);
buffeedStream.Close();

fileStream = new FileStream("temp.txt", FileMode.Open);
zipStream = new GZipStream(fileStream, CompressionMode.Decompress);
buffeedStream = new BufferedStream(zipStream);

byte[] result = new byte[buffer.Length];
buffeedStream.Read(result, 0, result.Length);

foreach (byte b in result)
    Console.Write(b);
```

Pipeline

SELA

Decorator my refer as Filters / Pipeline

(Non GOF)



Pipeline / Filter Pattern

- The output of each step use as the input for the next step
-
- Could be implement with Orchestrator or Decorator pattern

Filter

Filter Sample

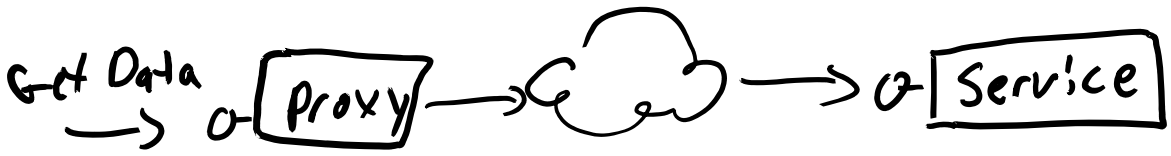
```
public class Slicer
{
    private readonly Predicate<int>[] _filters;

    public Slicer(params Predicate<int>[] filters)
    {
        _filters = filters;
    }

    public IEnumerable<int> Slice(IEnumerable<int> data)
    {
        foreach (var item in data)
        {
            if (_filters.All(f => f(item)))
                yield return item;
        }
    }
}
```

Proxy

Or…Who am I talking to?

What is a Proxy?

- Service's surrogate
- Abstract the real target
- Delegate calls to the real target



Proxy

Example

- HttpClient
  - Abstract REST calls

Proxy

SELA

Flyweight

Participants

- **Reuse** of **immutable** object in order to **reduce memory** allocation or **cost of creation**
- Often **combined with Composite Pattern** in order to **reuse parts** of the object

Example:

- Each of the collection can be **used** simultaneously in **multiple places**.

```
var colA = ImmutableList<int>.Empty;
var colB = colA.Add(1).Add(2)
var colC = colB.Add(3);
var colD = colC.AddRange(new[] { 11, 12, 13, 14, 15 });
```

Summary

- Adapter
- Bridge
- Composite
- Decorator / Pipe Line
- Façade
- Proxy
- Flywight

Questions

SELA

Module 07 - Behavioral Patterns

.NET Design Patterns

In this Chapter…

- Command
- Chain of Resp
- Template Method
- Null Object
- Iterator

Command

Participants

Command

- **Abstract execution**
  - **Encapsulate all information** needed perform action
  - **Expose execution** method (command interface)
- The command interface **can get parameters** or be **parameter-less**.
- The command is **self-contain** and don't rely on external resources for it execution
  - Many time the command is Serializable

SELA

Chain of Responsibility

If I can not handle the job maybe my neighbour can …

https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern

Chain of Responsibility

- **Break complex process** by **multiple simple processing units**
- **Each unit** in the chain **linked to other** units on the chain
- **Each unit** may **do part** of the work (or nothing) **before forward** the call though the chain

Chin of responsibility

Template method

Define skeleton

https://en.wikipedia.org/wiki/Template_method_pattern

Participants

- **Abstract Class**
  - Defines algorithm's abstract skeleton
- **Concrete Class**
  - Implements the skeleton (subclassing)

Template Method

Mediator

https://en.wikipedia.org/wiki/Mediator_pattern

Mediator Pattern (Hub)

Mediator

- Encapsulates **objects interactions**
- Object can "**Talk**" to each other **without "knowing"** about each other
- **Central** communication **control**
- Single point of **failure**
- Create Loosely coupled interaction
- Be carful of highly coupled with the mediator (everything is connected to the mediator)
- Sometimes hard to follow the communication flow

Platforms

Mediator

- All kind of Hub / Bus
- Orchestrators like K8s

Example

- Chat Room

Mediator

Iterator

- Foreach

https://en.wikipedia.org/wiki/Iterator_pattern

Iterator

- Define a consistent way to **iterate over collection**, independently from the collection implementation

Iterator

Iterator Design Pattern

- Different data structures expose different interaction API
  - Stack - `Push/Pop`
  - Array - `[]`
  - Liked list - `AddHead()`, `AddTail()`
  - Queue - `Enqueue()`, `Dequeue()`
- Client code is coupled to particular API
  - Difficult to change data structure

Iterator

yield return & yield break

- The yield return and yield break keywords signal to the compiler that the method in which they appear is an iterator block
- The compiler generates a class to implement the behavior that is expressed in the iterator block
  - yield return provide a value to the enumerator object. This is the value that is returned
  - yield break signal the end of iteration

```
public class List
{
   internal object[] elements;
   internal int count;

   public IEnumerator GetEnumerator() {
         for (int i = 0; i < count; i++) {
         yield return elements[i];
      }
   }
}
```

Iterator

Null Object

(Non GOF)

https://en.wikipedia.org/wiki/Null_object_pattern

The Null Object Pattern

- Representation of non null empty object
- Avoid null checking

Null Object

Example

- String.Empty
- Array.Empty<T>() // .NET 4.6

```
int? i1 = 4;
int? i2 = 4;
i2 = null;

int temp1 = i1 == null ? 0 : i1.Value;
int temp2 = i2 == null ? 0 : i2.Value;

Console.WriteLine(temp1 + temp2); // result: 4
```

Null Object

Questions

SELA

Module 08 - Behavioral Patterns

.NET Design Patterns

In this Chapter…

- State
- Strategy
- Observer
- MVC
- Mediator
- Visitor
- Memento

State

No more huge switch case structures to implement a state machine !!!

Let us leverage OOP…

The State Pattern

- **Change** Object's **behavior according to a State delegation**
- State **delegator implement the behavior** and **control** the **transition of state**

- The **state-full** object **don't define** the **transition** (in a switch like statement)

- The state delegator should define the transition
- Each state delegation aware of it direct transition option (ignore indirect transition)

Strategy

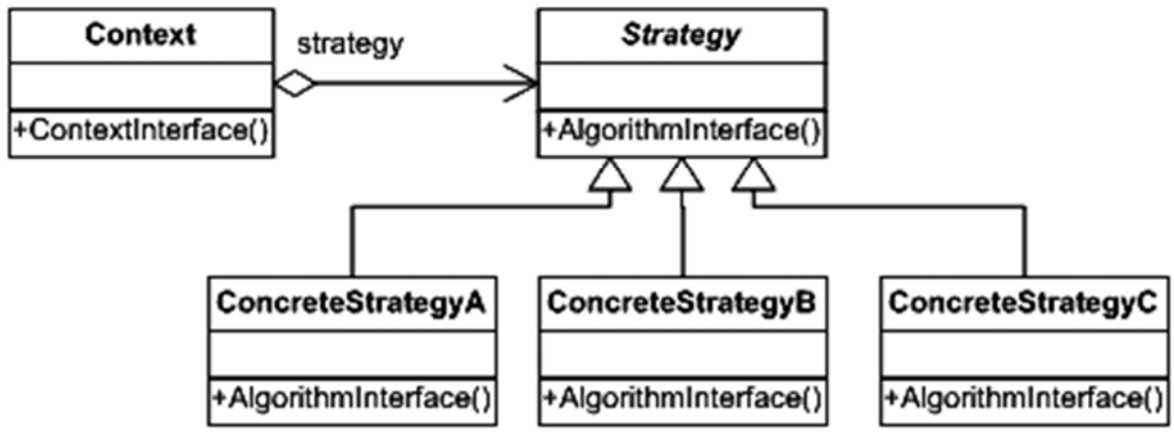
https://en.wikipedia.org/wiki/Strategy_pattern

The Strategy Pattern

- Choose algorithm at runtime

Strategy UML

UML class diagram



Examples

- Choose **price algorithm** according to the
  - user segment
  - time
  - Etc.

Strategy

Tariffs

Tariff

```
public interface Tariff
{
    int calcCall(int nTime);
}
```

NightTariff

```
public  class NightTariff : Tariff
{
           public int calcCall(int nTime)
           {
                      Console.WriteLine("Calculating Night Call");
                      return nTime;
           }
}
```

DayTariff

```
public  class DayTariff : Tariff
{
      public int calcCall(int nTime)
      {
            Console.WriteLine("Calculating Day Call");
            return nTime * 2;
      }
}
```

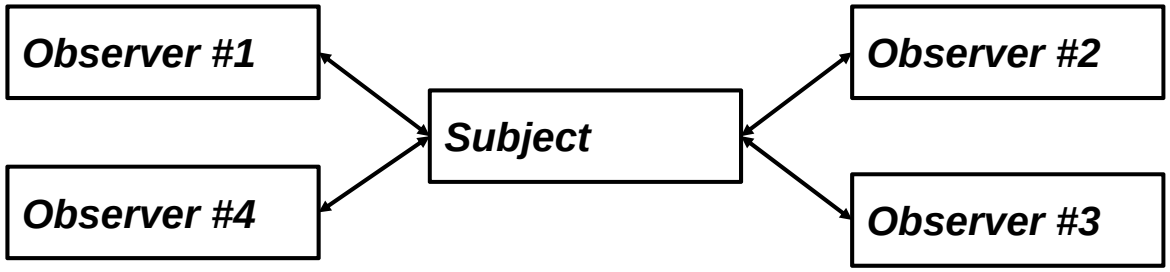
Strategy

SELA

Observer

The Observer Pattern

- The Observer pattern allows you to separate a source of data from its users
- Push model



.NET Delegate

- .NET helps implementing the Observer pattern with much less code using delegates and events
- Rather than implementing the Observer interface and registering itself with the subject, an observer create a delegate instance and register this delegate with the subject event

Observer

Observer .NET 4.0

- The IObserver<T> and IObservable<T> (.net 4.0), interfaces provide a generalized mechanism for the Observer design
- The IObservable<T> interface represent the class that sends notifications (Subject), The IObserver<T> interface represents the class that receives them

```
Namespace System
{
      public interface Iobservable<out T>
      {
                Idisposable Subscribe(Iobserver<T> observer);
      }

      public interface Iobserver<in T>
      {
                void OnCompleted();
                void OnError(Exception error);
                void OnNext(T value);
      }
}
```

Observer

Rx Extensions for .NET

- A library for composing asynchronous and event-based programs using observable collections
- Superset of the standard LINQ sequence operators that exposes asynchronous and event-based computations as push-based, observable collections.
- Using .NET 4.0 interfaces IObservable<T> and IObserver<T>
- Using PFX(Parallel Extensions for .NET)

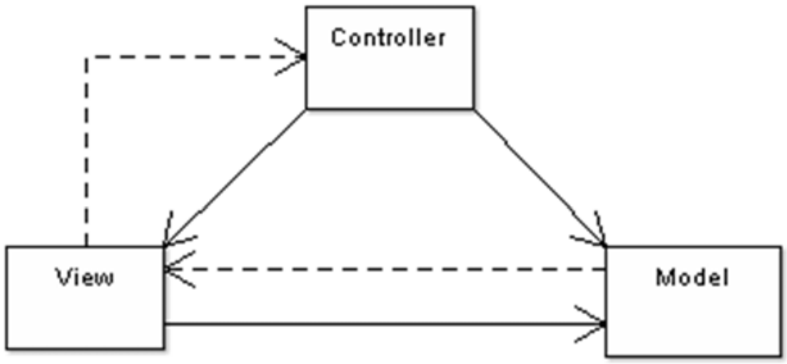
MVC

Three tiers computing

What is it?

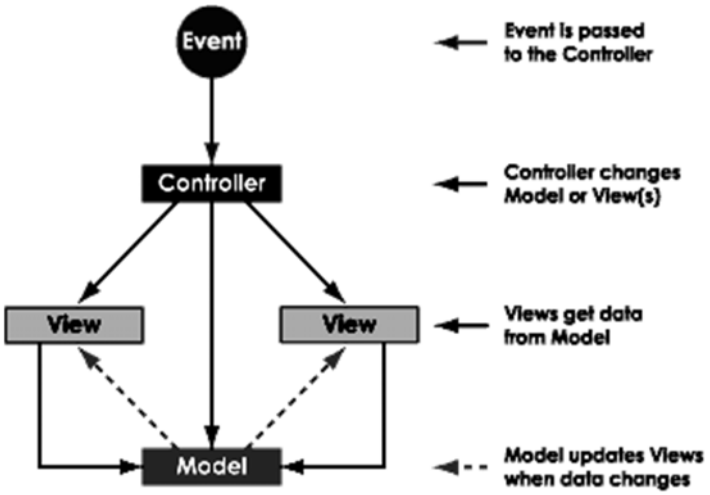
- **Model-view-controller (MVC)** is a classic design pattern often used by application that need the ability to maintain multiple view at the same time
- MVC design separates an application's into one of three categories **Data model**, **User interface**, or **Control logic**
  - **Model** for maintaining **data**
  - **View** for **displaying** all or a portion of data
  - **Controller** in charge of the logic of the application, and **handle events that effect the model or view**(s)
- With MVC modifications to one component can be made with minimal impact to the others.

MVC



How it works

- Events typically cause a controller to change a model or view or both.
- Whenever a controller changes a model's data all dependent views area automatically updated
- Whenever a controller changes a view, the view gets data from the underlying model to refresh itselt



MVC

SELA

Visitor

https://en.wikipedia.org/wiki/Visitor_pattern

Visitor

- **Separating** from Object Structure
  (which the algorithm operate on)
- Gain ability to add operation without changing the object structure

- Fit well on Tree or Graph structure

- Follow open/closed principle

Visitor

Visitor cons

- May be hard to debug
  Pattern matching may help

Visitor

.NET Examples

- Expression Tree operations
- Roslyn Compiler

Visitor

Memento (undo / rollback)

Or .Net Serialization

Memento definition

- Capture and externalize an **object's internal state** so that the object can be **restored** to this **state later**

Memento

Sample

```
public class OriginalObject
{
    private Memento _mememto;

    public string Name { get; set; }
    public int Value { get; set; }
    public OriginalObject(string name, int value)
    {
        Name = name;
        Value = value;
        this._mememto = new Memento(name, value);
    }

    public void Revert()
    {
        Name = this._mememto.Name;
        Value = this._mememto.Value;
    }
}
```

SELA

Broker
(Non GOF)

How to distribute a system

Before you distribute it…

- Introduce significant complexity into the software system.
  - communications, encoding, and security …

Server LookUp – The Broker

- The client use a **broker** to **dynamically find the server**
- Client should **know the broker location** and the **broker will know everything else**
- The broker component is **responsible for locating the server** for the client

Broker

Communication pattern

- Broker may support one of the following pattern
  - The **client obtains** the **location** of the server from the broker and **then communicates with the server directly**
  - The broker will forward all calls on behalf of the client (firefall, Security, tractability, etc.).

Broker

Summary

- Command
- Chain of Resp.
- Template Method
- Null Object
- Memento
- Iterator
- State
- Strategy
- Observer
- MVC
- Mediator
- Visitor

Questions

SELA

Module 09 - Using UML to Describe Design Patterns

.NET Design Patterns
Apendix

Unified Modeling Language (UML) Table of Contents

Unified Modeling Language (UML) Table of Contents

What is a Model?

- A **model** is a representation of entities
- A model is expressed in a way that is convenient to work with
- A model emphasizes important aspects of an entity from a specific point of view
  - "Unimportant" aspects are simplified or omitted
- A model enables:
  - Control of complicated structures
  - Communication with different project teams
  - Dealing efficiently with several solutions
  - Exposing and then focusing on weak elements within the system

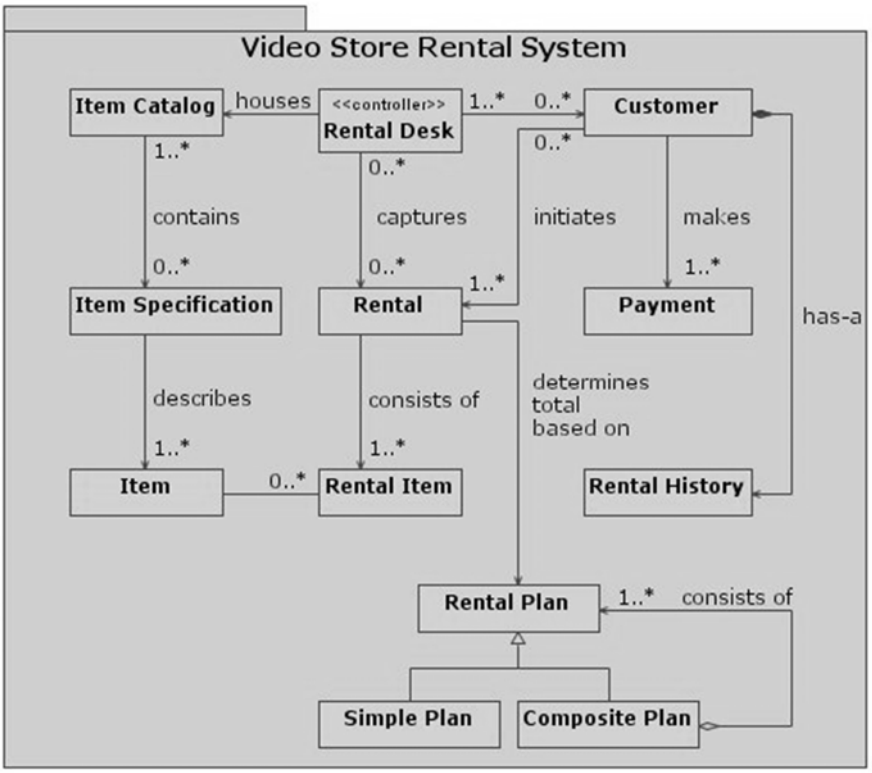
What is UML?

- UML (Unified Modeling Language) is a standard modeling language providing tools to create models of object-oriented software systems
- UML defines lots of graphical notations that enables modeling different aspects of software systems
- UML was designed to be a general-purpose modeling language
  - Programming language independent
  - Development process independent
- A software system model may be expressed at different levels of resolution, representing its different development stages:
  - Conceptual level - Domain analysis
  - Specification level - High-level design / Interfaces
  - Implementation level - Skeleton definition for code generators

UML Graphical Diagrams

- UML defines several graphical diagrams that enable modeling different views of the system:
  - Use case diagram
  - Class diagram
  - Behavior diagrams
  - Implementation diagrams
- Note:
  Complex systems are best approached via a set of nearly independent views of a model
  - No single view is sufficient
  - Not all systems require all views
- This chapter provides a brief overview of a single UML diagram:
  - The Class diagram

Conceptual Class Diagram

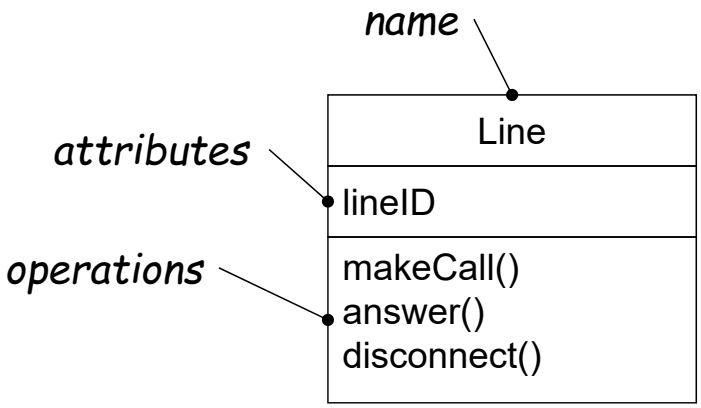


The Role of a Class Diagram

- Defines the system's elements
- Describes relations between system elements
  - The dynamic behavior of the diagram's elements are expressed in other diagrams

Classes

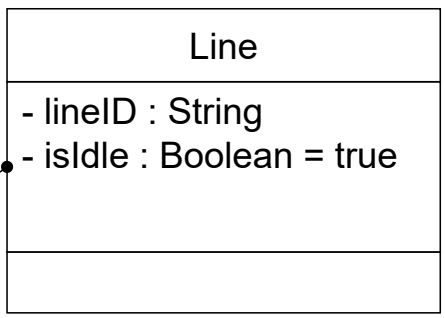
- They are the most important element of any object-oriented system
  - Associate operations with data – encapsulation



- Every class must have a name
  - A class name must be unique within its package

Attributes and Visibility

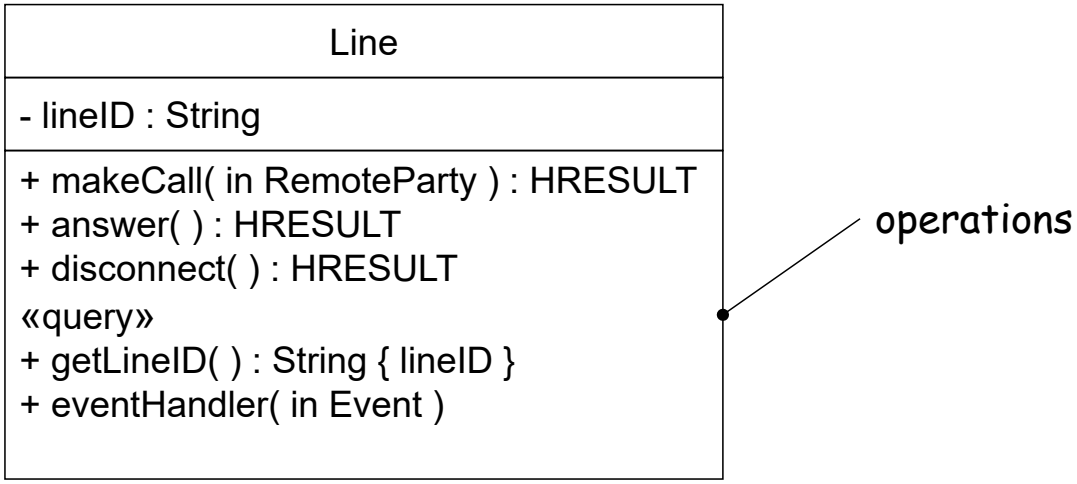
- An attribute is a named property of a class
- Attribute syntax:
  - visibility name : type = default-value
- Where visibility is one of:
  - + public visibility
  - # protected visibility
  - - private visibility



Operations

- An **operation** is a service that an instance of a class may be requested to perform
  - It has a name and a list of parameters
- Operation syntax:
  - visibility name (parameter-list): return-type
  - parameter-list syntax:
    - kind name : type = default-value
    - Where kind (direction) is:
      - in, out, or inout, with the default in

Operations – cont'd

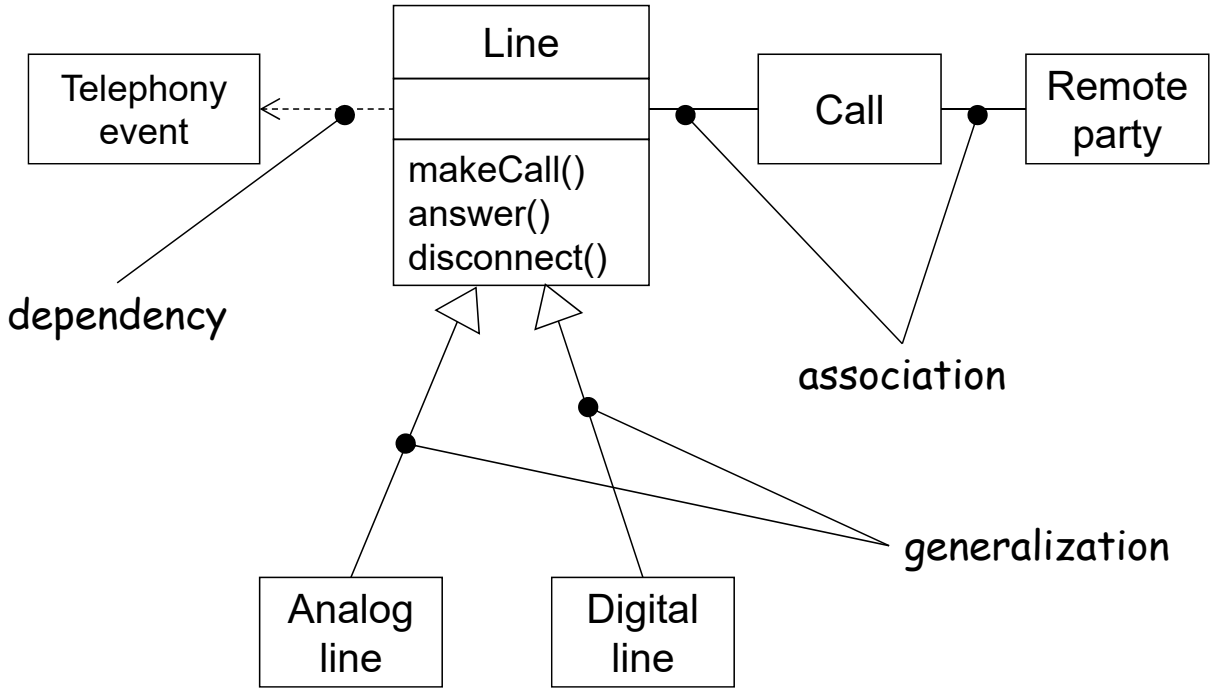


- An operation that does not modify the system state (one that has no side effects) is specified by the property "query"

Relationships

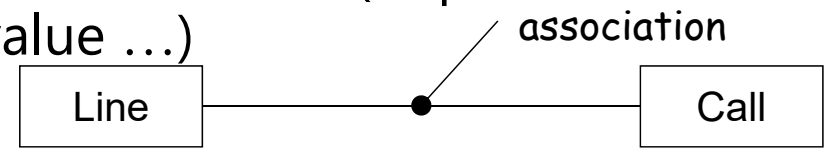
- A relationship describes a connection between model elements.
  - The description characterizes the type of the connection.
- Types of relationships:
  - Association
  - Generalization
  - Realization
  - Dependency
    - Parameter
    - Instantiation
    - Call

The Representation of a Relationship

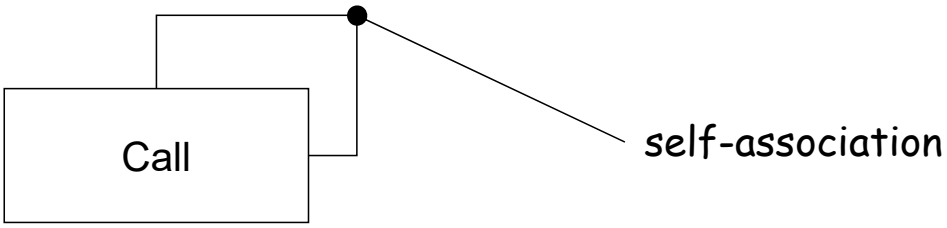


Association

- **Association** means that objects of one class are connected to objects of another class
- Object can invoke objects of the class on the other association end (as parameters or as return value …)

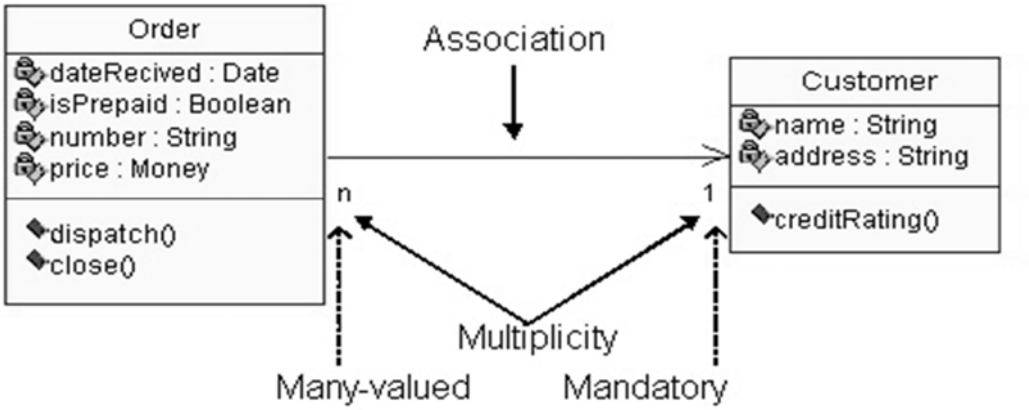


  - **Self-association** means that objects of one class are connected to other objects of the same class



Association

- The **association** shows the **relationship** between instances of classes
- The multiplicity of the association denotes the number of objects
  that can participate in then relationship



Association and Navigability

- Association has a direction
  - The direction is represented by an arrow
- The default association is bi-directional (no arrows)
  - Each object can invoke objects of the class on the other association end

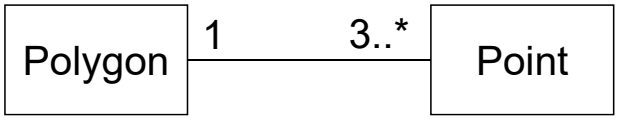
Association and Multiplicities

- **Multiplicity** on an association end indicates how many objects may participate in the given relationship
- Multiplicities syntax examples:
  - 4..* Four or more
  - 0..* 0 or more, no limit

  equivalent to:
  - * Many
  - 1..3,7,9 1, 2, 3, 7,9

Association andm Multiplicities Examples

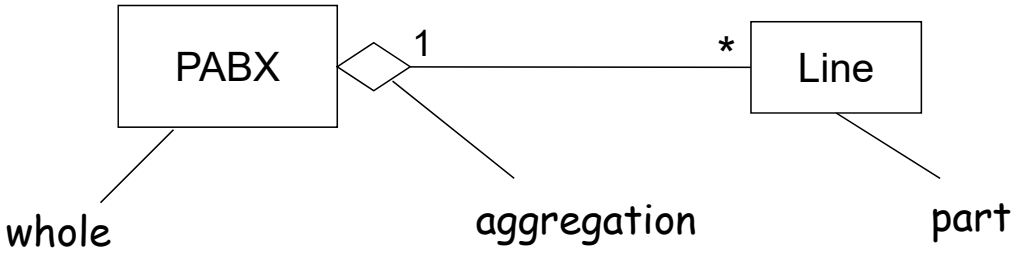
Line has 0 to 2 calls, Call has one and only one Line,
Call has one or two remote parties



Polygon has at least 3 Points

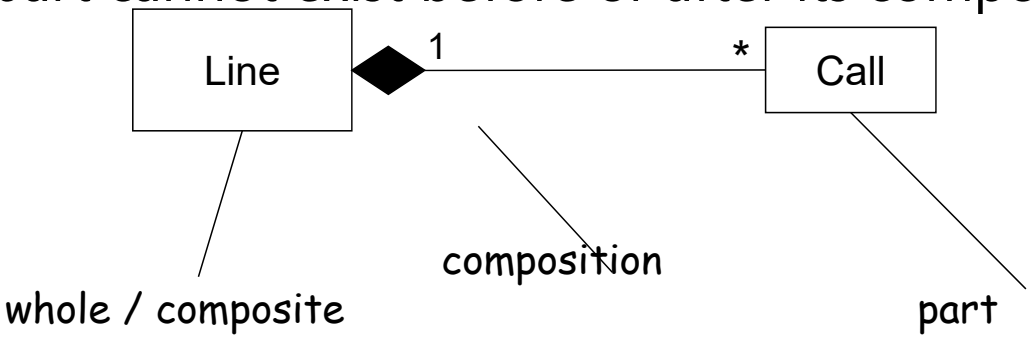
Aggregation

- Aggregation is an association
- Models a relationship known as:
  - whole / part
  - “has-a” relationship
- The whole / part lifetimes are not linked



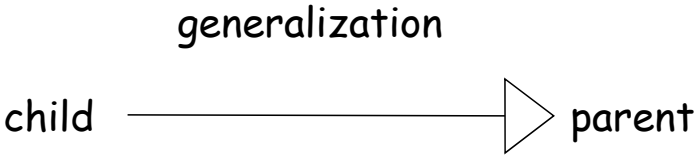
Composition

- **Composition** is aggregation with stronger ownership
- A part may belong to only one composite (whole)
- A part cannot exist before or after its composite

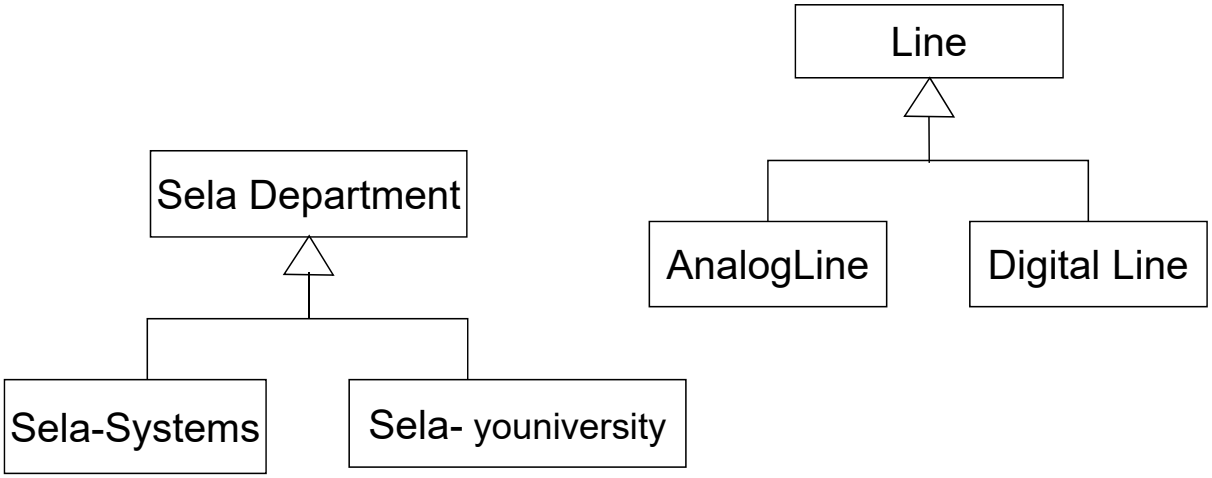


Generalization

- **Generalization** is a relationship between a general element (the parent), and a more specific element (the child)
- Models a relationship known as: "is a" relationship
- A child inherits the properties of its parent
  - Attributes
  - Operations
  - Relationships

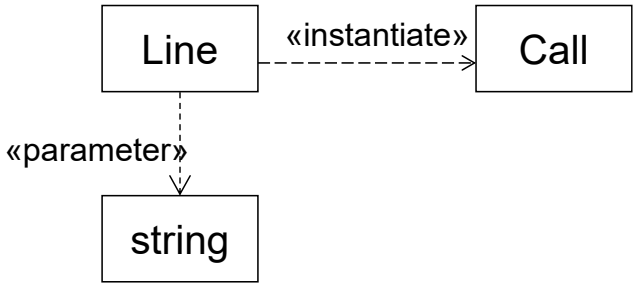


Generalization Notation



Dependency

- **Dependency** means that one element requires the presence of another element for its correct implementation or functioning
- A dependency may be stereotyped* further to indicate the exact nature of the dependency
  - «call»
  - «instantiate»
  - «parameter»
  - «friend»
  - …



** A stereotype represents a variation of an existing model element*

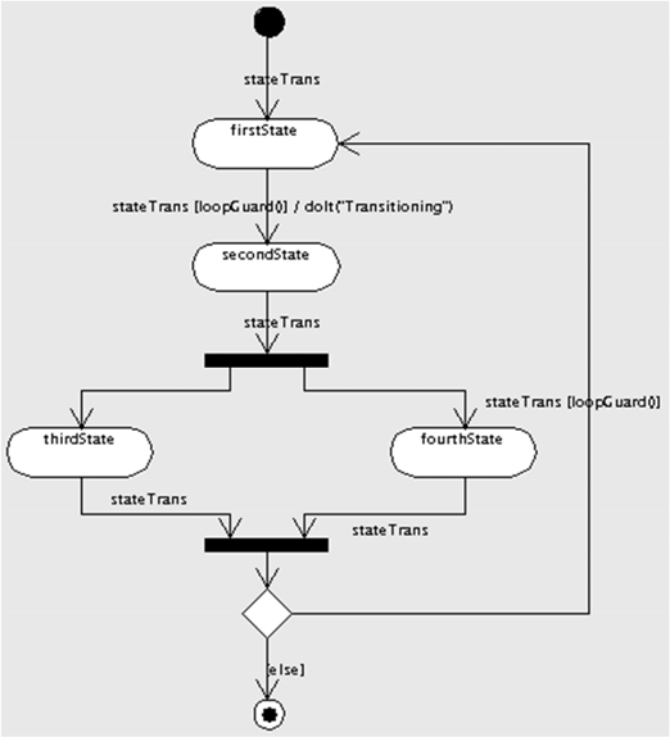
Realization (Implementation)

- Realization relationship exists between two classes when **one of them must realize, (implement), the behavior specified by the other**
- In a realization relationship, one entity (normally an interface) defines a set of functionalities as a **contract** and the other entity (normally a class) "realizes" the contract by implementing the functionality defined in the contract
- The class that specifies the behavior is called the **supplier**, and the class that implements the behavior is called the **consumer**

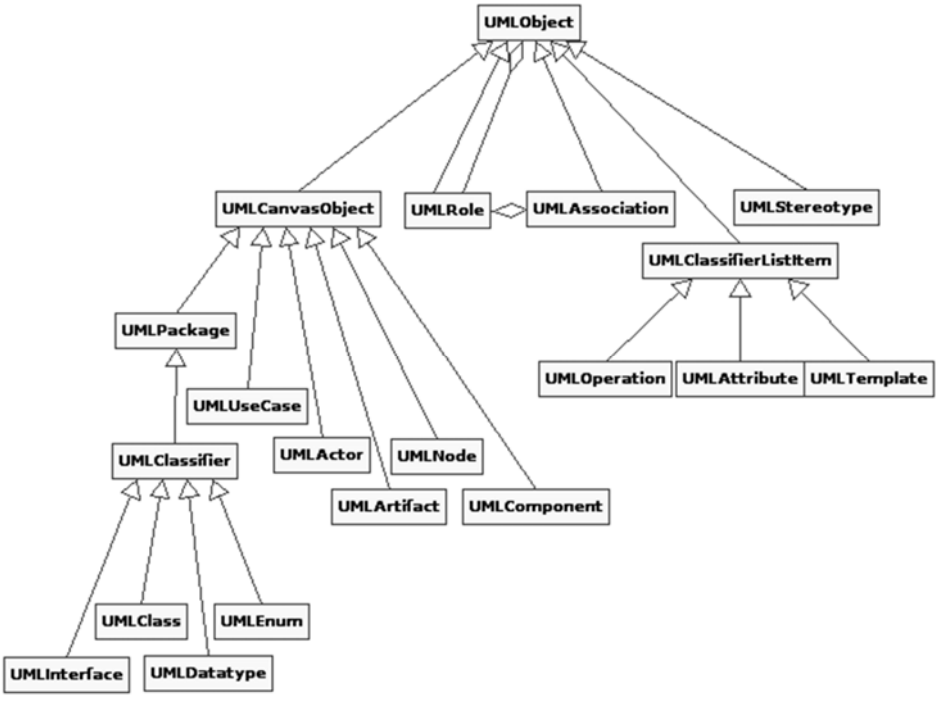
UML is a language

- In UML there are 9 different types of diagrams
  - Activity Diagram – *Popular*
  - Class Diagram – *Popular*
  - Communication Diagram
  - Component Diagram
  - Composite Structure Diagram
  - Deployment Diagram
  - Interaction Overview Diagram
  - Object Diagram
  - Package Diagram
  - Sequence Diagram – *Popular*
  - State Machine Diagram
  - Timing Diagram
  - Use Case Diagram – *Popular*

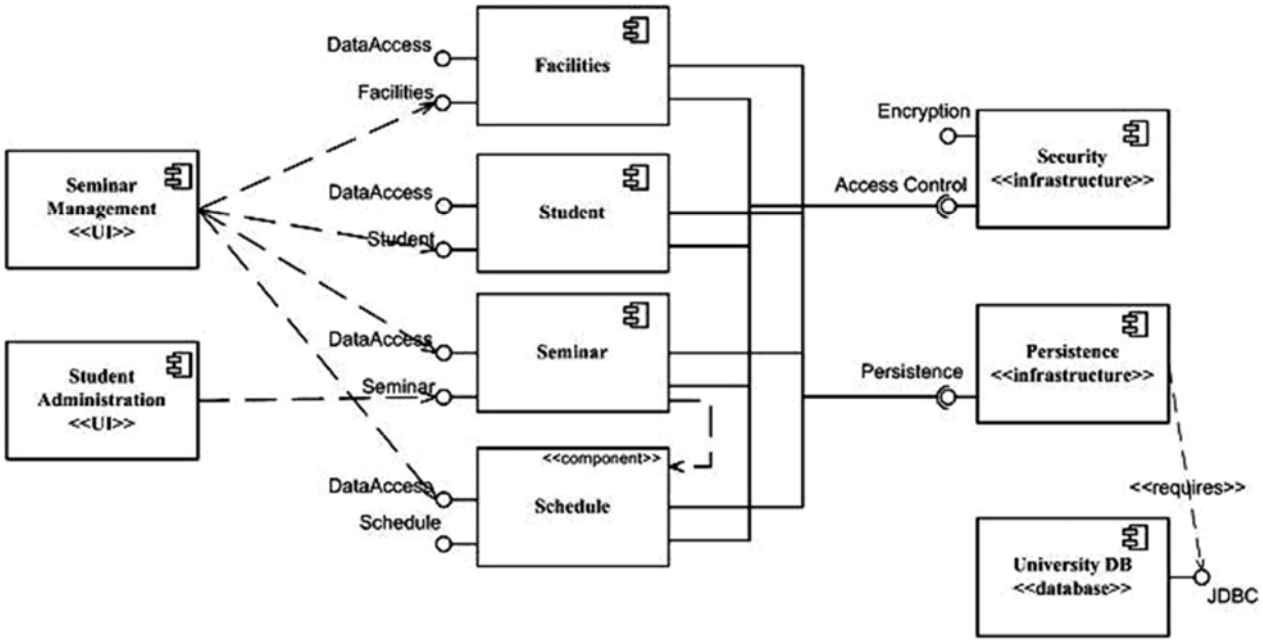
Activity Diagram



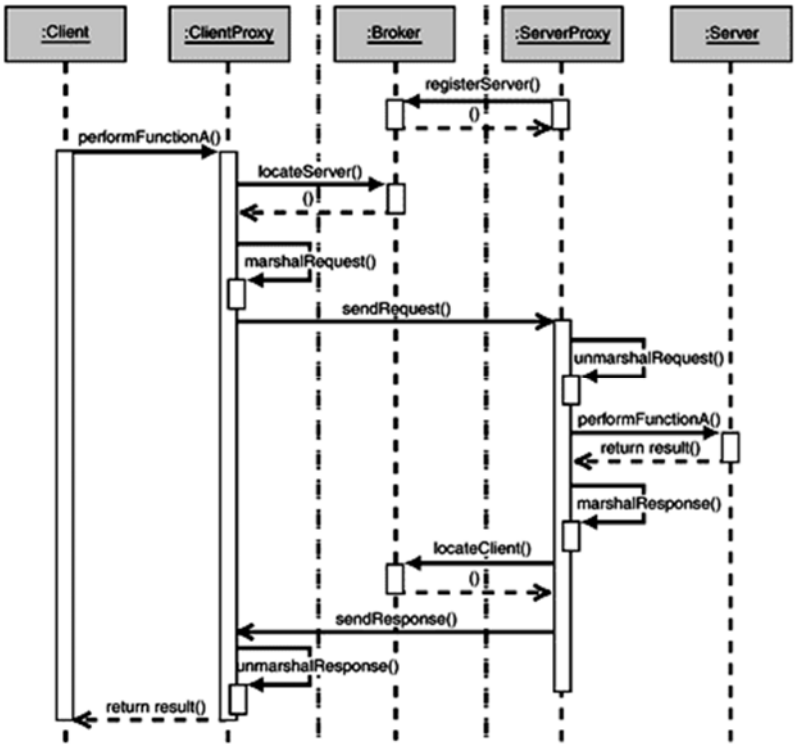
Class Diagram



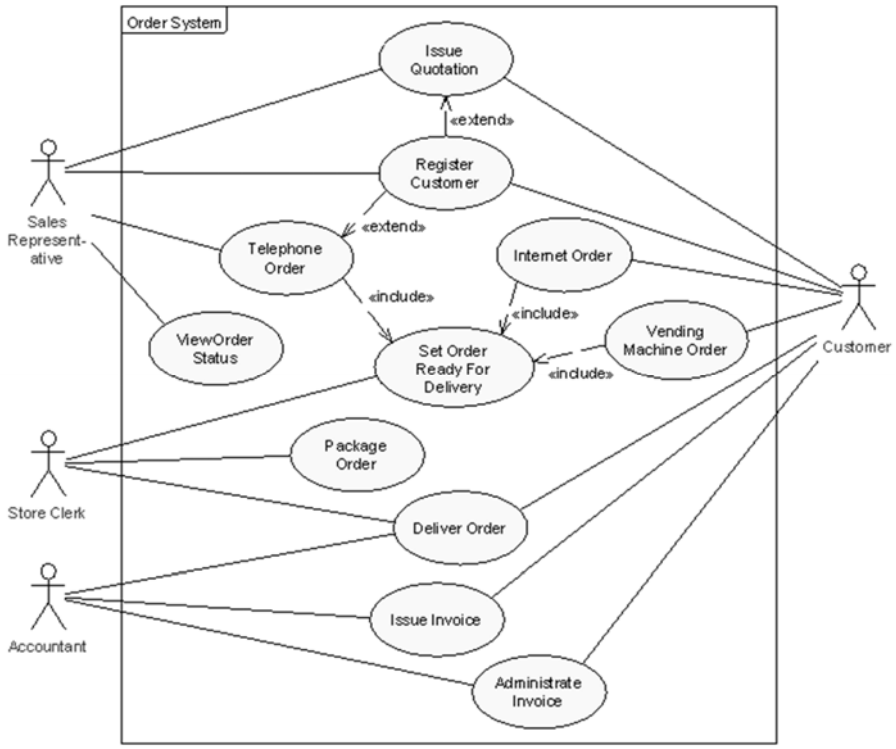
Component Diagram



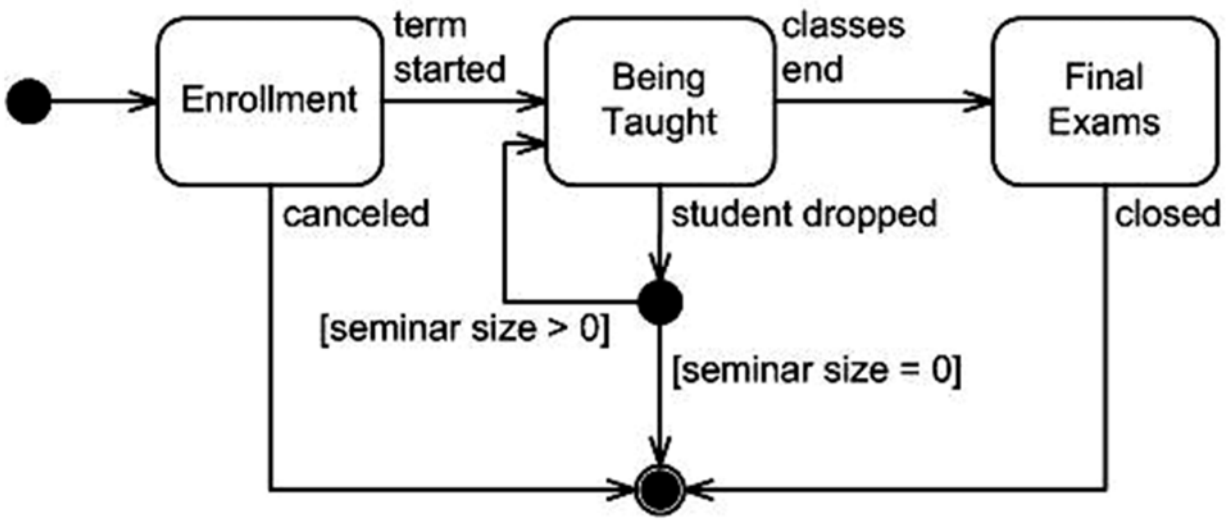
Sequence Diagram



Use Case



State machine diagram



Summary

- UML defines a set of graphical notations that enable modeling different aspects of software systems.
- Class diagram describes system element and relations
- Different types of UML diagrams

Questions

Module 10 - .Net & Concurrency Patterns (Non GOF)

.NET Design Patterns
Appendix

IDisposable

How to clean resources ?

Motivation

- Any class that holds resources must be cleaned.
- We want to control **when** it is cleaned.
- The effect of a finalizer is huge.
- There must be a separation in the cleaning process between managed and un-managed resources.
- We want to write the cleaning code once.
- We want to implement the IDisposable interface so we can use simple sintax like "**`using(){…}`**"

```
protected bool disposed = false;
  // disposing is true when we call Dispose
  // Note that this is not thread safe.
protected virtual void Dispose(bool disposing)
  {
      if (!this.disposed)
      {
          if(disposing)
          {
              // Add code here to release the managed resource.
              //managedResource.Dispose();
          }
          // Add code here to release the unmanaged resource.
          // unmanagedResource = IntPtr.Zero;
      }
      this.disposed = true;
  }
  ======= IDisposable Support ========================================
  // Put cleanup code in Dispose(bool disposing)
public void Dispose()
  {
      Dispose(true);
      GC.SuppressFinalize(this);
  }
  ------------------------------------------------------------------
  ~ResourceClass()
  {
      Dispose(false);
  }
```

Asynchronous Programming in .NET

- Support for Asynchronous Programming in .NET
- Design Pattern for Asynchronous Programming
- Asynchronous File Stream Read Example
- Asynchronous Delegates

Support for Asynchronous Programming i.NET

- Supported in Many Areas of .NET
  - I/O, sockets, networking, remoting, ASP.NET and Web Services, messaging (MSMQ), delegates
- Offers a design pattern for asynchronous programming
  - Consistent and type-safe programming model
  - User-created classes should conform to this design pattern

Design Pattern for Asynchronous Programming

- Caller decides whether to make a synchronous or asynchronous call
  - Server may explicitly implement asynchronous methods, or client can use delegate object's asynchronous support
- Caller does asynchronous operation in two parts
  - Call begin method to supply parameters and start operation
    - an object implementing **IAsyncResult** is returned
  - When operation is done, call end method to obtain the results
- Caller has several options to know when operation is done
  - **Callback** method - if specified in the begin method call, callback is invoked
  - **Poll** – check **IAsyncResult.IsCompleted** property
  - **Call** end method – if called before operation is done, automatically waits
  - **Wait** – wait on **IAsyncResult.WaitHandle** property, can use timeouts

Asynchronous File Stream Read Example

- Asynchronous read with callback
  - Create callback method delegate

```
AsyncCallback myCallback = new
      AsyncCallback(this.OnReadDone);
```

  - Begin the operation

```
IAsyncResult ar = aStream.BeginRead(buffer,0,
      buffer.Length, myCallback,(object)myState);
```

  - In the callback method, complete the operations

```
int byteCount = aStream.EndRead(ar); //data in buffer
```

Asynchronous File Stream Read Example (continued)

- Asynchonous read with polling
  - Begin the operation

```
IAsyncResult ar = aStream.BeginRead(buffer,0,
   buffer.Length, null,(object)myState);
```

  - Poll and complete

```
while (!ar.IsCompleted){ // do whatever
}
int byteCount = aStream.EndRead(ar); // data in buffer
```

Asynchronous Delegates

- Delegate object provides the ability to call a synchronous method in an asynchronous manner
  - Compiler generates synchronous invoke method and asynchronous methods: **BeginInvoke** and **EndInvoke**
- Asynchronous calls follow the design pattern
  - Start operation with **BeginInvoke**
  - Use callback, polling, call to **EndInvoke**, or wait to determine completion
  - Call **EndInvoke** to get results

SELA

Balking

If an object's method is called when the object is not in an appropriate state to execute that method, have the method return without doing anything.

The Balking Pattern

- This pattern allows objects discarding a methods call when an object is not in an appropriate state.
- Let's say you were asked to implement a non interactive banking application that displays stock exchange values.
- This application works 24/7 and updates its data automatically every 15 min.
- The only interactive feature this application supports is a user request to refresh the application's data.
- User may click the refresh button at any time.

The Balking Pattern: Motivation

- What should happen when a user presses the refresh button while an automatic refresh process takes place?
- What will happen when an automatic refresh signal arrives during the time that a manual refresh request is being processed?
- The simplest solution is to perform two refresh actions in a roll, though this will waste server and network resources.
- The best solution in such a case is to discard the second refresh request.

The Balking Pattern – cont'd

- When an application receives a refresh request, it should check if it is currently in the process of refreshing its data.
- If not, the application marks its state as "refreshing data".
- If true, nothing should happen and the refresh method should return immediately.

Loader Version 1.0

```
public class Loader
{
  private Viewer m_Viewer;
  public Loader(Viewer v)
  {
    m_Viewer = v;
  }
  public void reload()
  {
     Thread t = new Thread(new ThreadStart(run);
     t.Start();
  }
  public void run()
  {
        m_Viewer.showStartLoading();
        for (int i = 0 ; i < 10 ; ++i)
        {
          m_Viewer.setLoadingStatus(i * 10);
          try {Thread.Sleep(1000); }
          catch (InterruptedException e) {}
        }
        m_Viewer.showLoadingDone();
  }
}
```

Q **What will happen if the Loader will be asked to reload during a reload process?**

The Balking Pattern: Viewer

```
public class Viewer
{
   public void showStartLoading()
   {
        Console.WriteLine("Start Loading...");
   }
   public void showLoadingDone()
   {
        Console.WriteLine("Loading done.");
   }
   public void setLoadingStatus(int i)
   {
        Console.WriteLine("Loading "+ i+"%...");
   }
}
```

Loader

- Two (or more) concurrent loading processes.
- Will it be helpful to make the reload(…) method synchronized?

The Balking Pattern: Loader Version 2.0

```
public class Loader
{
    private Viewer m_Viewer;
    private bool m_Loading;
    public  Loader(Viewer v)
    {
          m_Viewer = v;
    }
    public void reload()
    {
        lock(this)
      {
         if (m_Loading)
         {
          Console.WriteLine("Manual reload request discard...");
          return;
         }
          m_Loading = true;
      }
      Thread t = new Thread(new ThreadStart(run));
      t.Start();
    }
```

The Balking Pattern: Loader Version 2.0 – cont'd

```
   public void run()
     {
        m_Viewer.showStartLoading();
        for (int i = 0 ; i < 10 ; ++i)
        {
                m_Viewer.setLoadingStatus(i * 10);
                try {Thread.Sleep(1000); }
                catch (Exception e){}
        }
        m_Viewer.showLoadingDone();
        lock(this)
        {
                m_Loading = false;
        }
   }
}
```

Client

```
public class Client
{
   public static void Main()
   {
        Loader l= new Loader(new Viewer());
        //Automatic reload

        l.reload();
        Thread.Sleep(5000);
        //Manual load request
        l.reload();
        Thread.Sleep(7000);
        //Another manual load request
        l.reload();
        Console.Read();
        }
   }
```

SELA

Scheduler

The Scheduler Pattern

- Sometimes you want several threads to access the same resource and you want to verify that only one thread will use the resource at a time.
- The .NET language supports the lock keyword that can easily help you to accomplish this task.
- Using this keyword, you can be sure that only one thread will reside in a code block running on an object context.

```
public class SharedResource {
   public void doSomething() {
        lock(this)
        {
           //Access the shared resource
        }
   }
}
```

The lock Keyword

- Let's say you want to implement a login service.
- This service will support an unlimited number of threads.
- All threads will call a synchronized method of class login, which will return a boolean value specifying success or failure.

```
Public class Login
{
    public bool doLogin(Data data)
    {
          lock(this)
          {
             return (checkData(data));
          }
    }
}
```

Login Service

- The main question is: How will the queuing work?
- Who will get the service first?

Login Service

- How can you define your own queue?
- How can you control the order of the threads that will gain ownership of a shared resource?

The Scheduler Pattern

```
public class SharedResource
{
     Scheduler m_Scheduler = new Scheduler()
     public void doSomething(SomeData)
     {
        //The enter method will halt the caller
        //thread until the scheduler decides to
        //let it continue.
        m_Scheduler.enter(SomeData);
        //Do the job
        processData(SomeData);
        //Tell the scheduler that the current
        //thread is done processing data, so
        //the scheduler will allow other threads
        //to work.
        m_Scheduler.leave();
     }
}
```

The Scheduler Pattern

- The scheduler has to maintain the client's queue according to an application specific logic.
- A simple logic may be:
  - Let clients perform a task according to their thread's priority and time their request was arrived.
- How can we manage other, more unique priorities?

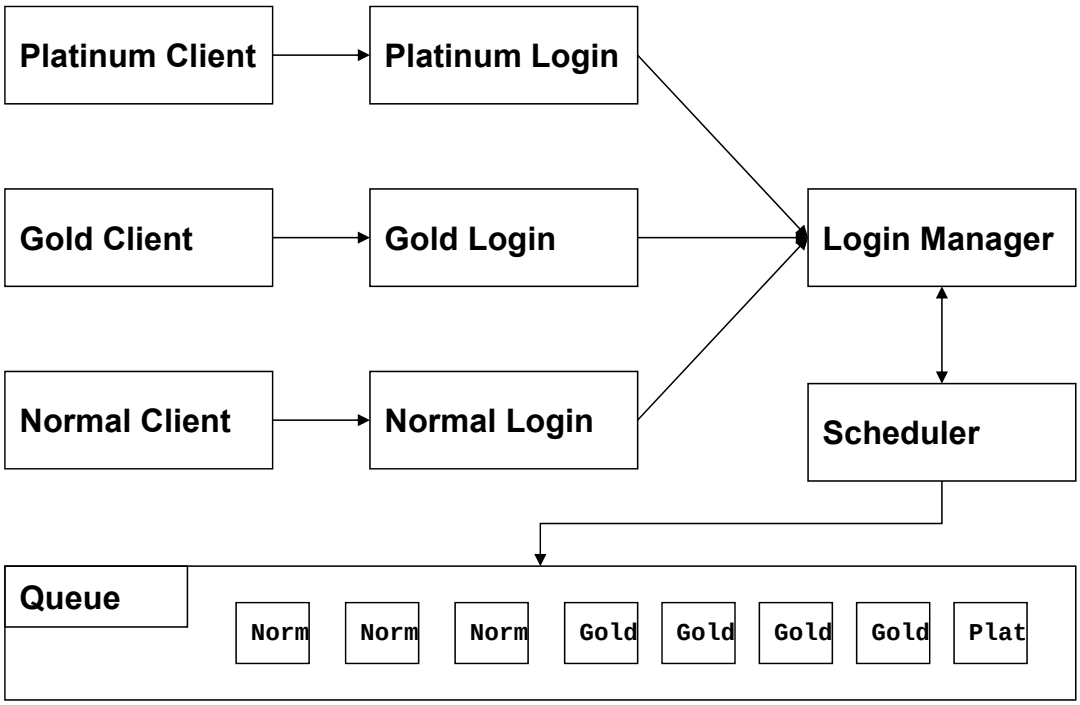
Controlling Priorities

- Each client that calls the shareResource' doSomething(…) method, passes some user data.
- This data may contain information about the priority of this task.
- The scheduler may analyze this data and decide when this task should be executed.
- Commonly, all these user data classes will implement an interface to allow their comparison.
  - For example, they may implement System.IComparable interface.

Scheduler - Example

- In this example, we will implement a login system.
- This login system will serve three different types of clients:
  - Platinum Clients – These are the most important clients. They should be served as soon as possible.
  - Gold Client – These clients are less important than the Platinum clients.
  - Normal Clients – These will be served last.
- Our login system will queue all client request, and execute them according to the time they have arrived and their user's type.

Scheduler Example



LoginInfo

```
public abstract class LoginInfo : IComparable
 {
    protected static  int PRIORITY_LOW = 0;
    protected static  int PRIORITY_MEDIUM = 5000;
     protected static  int PRIORITY_HIGH = 10000;
     private string m_sUserName;
     private string m_sPassword;
     private long m_LoginTime;
     public int CompareTo(object other)
     {
       if (!(other is LoginInfo))     throw new ClassCastException();
       int nPriorityDifference = getPriority() - ((LoginInfo)other).getPriority();
       int lTimeDifference = (int)(m_LoginTime - ((LoginInfo)other).m_LoginTime);
       return nPriorityDifference - (int)lTimeDifference;
     }
```

LoginInfo – Cont.

```
   public LoginInfo(string sUserName, string sPassword)
     {
       m_sUserName = sUserName;
       m_sPassword = sPassword;
       m_LoginTime = DateTime.Now.Second*1000+DateTime.Now.Millisecond;
     }
   public string ToString()
     {
       return "Login Info, Priority = " + getPriority() + " Name = " + m_sUserName;
     }

   public string getUserName() { return m_sUserName; }
   public string getPassword() { return m_sPassword; }
   public abstract int getPriority();
}
```

Login Data Types

```
public class SimpleClientLogin : LoginInfo
 {
    public SimpleClientLogin(string sName, string sPassword):
          base(sName, sPassword){}
    public override int  getPriority()
      { return PRIORITY_LOW;}
}


public class GoldClientLogin : LoginInfo
  {
    public GoldClientLogin(string sName, string sPassword):
          base(sName, sPassword){}
    public override int getPriority()
      { return PRIORITY_MEDIUM;}
 }

public class PlatinumClientLogin : LoginInfo
 {
    public PlatinumClientLogin(string sName, string sPassword):
          base(sName, sPassword){}
    public override int getPriority()
      { return PRIORITY_HIGH;}
}
```

Client

```
public abstract class Client
{
   private LoginInfo m_Info;
   private LoginManager m_lManager;
   public Client(LoginInfo info, LoginManager lManager)
     {
        Thread t = new Thread(new ThreadStart(run));
        m_Info = info;
        m_lManager = lManager;
        t.Start();
     }
   public void run()
     {
        try
        {
           m_lManager.login(m_Info);
        }
        catch (InterruptedException e)
        {
           Console.WriteLine("Internal Error in login mechanism");
        }
     }
}
```

Client Types

```
public class NormalClient : Client
{
    public NormalClient(string sName, string sPassword, LoginManager Manager)
        :base(new SimpleClientLogin(sName, sPassword), Manager){}
}

public class GoldClient : Client
{
    public GoldClient(string sName, string sPassword, LoginManager Manager)
        :base(new GoldClientLogin(sName, sPassword), Manager){}
}

public class PlatinumClient : Client
{
    public PlatinumClient(string sName, string sPassword, LoginManager Manager)
        :base(new PlatinumClientLogin(sName, sPassword), Manager){}
}
```

Login Manager

```
public class LoginManager
{
    private LoginScheduler m_Scheduler = new LoginScheduler();
    public bool login(LoginInfo login)
    {
        bool bRetVal;
        m_Scheduler.enter(login);
        Console.WriteLine("Start process: " + login);
        //Sleep, ilustrate processing time...
        try { Thread.Sleep(500); }
        catch (InterruptedException error) { }
        bRetVal = evalLogin(login);
        Console.WriteLine("Processe done: " + login);
        m_Scheduler.leave();
        return bRetVal;
    }
```

Login Manager – cont'd

```
    private bool evalLogin(LoginInfo login)
    {
        string sName = login.getUserName();
        string sPassword = login.getPassword();
        string sDesieredPassword = reverse(sName);
        return sDesieredPassword.Equals(sPassword);
    }

    private string reverse(string source)
    {
    string destination="";
    for(int i=source.Length-1;i>0;--i)
        destination+=source.Substring(i,1);
    return destination;
    }

}
```

The Scheduler

```
public class LoginScheduler
{
    private Hashtable m_LoginTasks = new Hashtable();
    private Thread  m_RunningThread = null;
    public void enter(LoginInfo login)
    {
        Thread CurrentThread = Thread.CurrentThread;
        lock(this)
        {
            if (m_RunningThread == null)
            {
                m_RunningThread = CurrentThread;
                return;
            }
            Console.WriteLine(login.ToString() + CurrentThread);
            m_LoginTasks.Add(login, CurrentThread);
        }
        lock (CurrentThread)
        {
            while (m_RunningThread != CurrentThread)
                Thread.CurrentThread.Suspend();
        }
        lock(this)
        {
            m_LoginTasks.Remove(login);
        }
    }
```

The Scheduler – cont'd

```
public void leave()
{
    lock (this)
    {
        if (m_RunningThread != Thread.CurrentThread)
            throw new IllegalStateException("Wrong Thread Running!");
        //No thread is waiting
        if (m_LoginTasks.Count == 0)
        {
            m_RunningThread = null;
            return;
        }
        //There is a thread waiting. Remove the highest priority job
        IDictionaryEnumerator iDicEnum = m_LoginTasks.GetEnumerator();
        iDicEnum.MoveNext();
        m_RunningThread = (Thread)m_LoginTasks[iDicEnum.Key];
        lock (m_RunningThread)
        {
            Monitor.PulseAll(this);
            //m_RunningThread.notifyAll();
        }
    }
}
```

Discussion

- When do you think should we use the pattern
  - Give me some examples
- What do you think are the Pros and Cons
  - Give some more examples

SELA

Read / Write Lock

Multiple readers one writer …

Read / Write Lock

- Let's say you were asked to implement a web site for a stock exchange.
- This web site should display the current values of any stock.
- In addition, the web site will be online updated from the stock exchange market.
- You have to make sure, that at any time, the values displayed in the site are concurrent with the real values.

Read / Write Lock – cont'd

- The simplest way to implement such a system is to let only one thread to read or write stock price at a time.
- For example:

```
1.  class Stock
2.  {
3.          private string name;
4.          private float price;
5.          Stock(string name)
6.          {
7.                          this.name = name;
8.          }
9.          public float Price
10.         { get { lock (this) {
11.                         return price;}}
12.           set { lock (this) {
13.                         price = value;}}
14.         }
15. }
```

Read / Write Lock – cont'd

- Statistically, we are about to receive much more read requests than write requests.
- There is no reason to prevent a multiple of clients from reading the data at the same time.
- Can we remove the lock keyword from the get method?

```
1.   class Stock
2.   {
3.        private string name;
4.        private float price;
5.        Stock(string name)
6.        { this.name = name;}
7.        public float Price
8.        { get { return price;}
9.          set { lock (this) {
10.                 price = value;}}
11.       }
12.  }
```

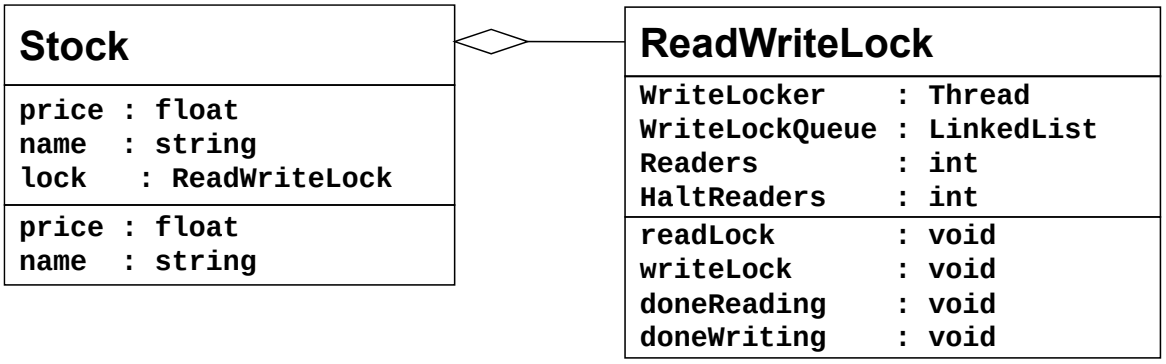
Read / Write Lock – cont'd

During the execution of a get method, no other thread can call the set method
Multiple threads can invoke the get(…) method at the same time

During the execution of a set(…) method, no other thread can call the set or get methods

- **Clients entering the get method should obtain a read lock**
- **Clients entering the set method should obtain a write lock**
- **Clients will not obtain a read lock while other are waiting to obtain a write lock**
- **Clients will not obtain a write lock while there is a client that own a read lock**

Object Model



- **WriteLocker** – a reference to the thread currently owning the write lock. If not null, it points to the fact that there is a thread owning a write lock.
- **WriteLockQueue** – hold reference to all threads waiting to obtain a write lock.
- **Readers** – hold the number of threads currently owning a read lock.
- **HaltReaders** – hold the number of threads currently waiting to obtain a read lock. If not zero, it points to the fact that there is a write locker and / or that the WriteLockQueue is not empty.

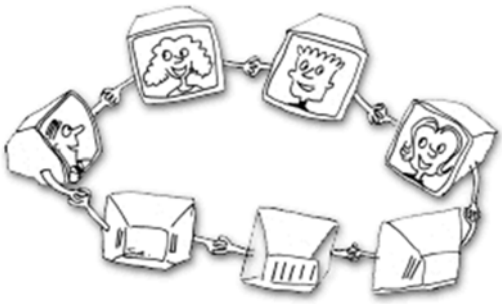
ReadWriteLock Example

```
ReaderWriterLock rwl = new ReaderWriterLock();

public override int Read(int threadNum)
{
    // ...
    // method code that doesn't require exclusive access
    rwl.AcquireReaderLock(Timeout.Infinite);
    try
    {
        Console.WriteLine(
         "Start Resource reading (Thread={0})count: {1}", threadNum, count);
        Thread.Sleep(250);
        Console.WriteLine(
         "Stop Resource reading (Thread={0}) count: {1}", threadNum, count);
        return count;
    }
    finally
    {
        rwl.ReleaseReaderLock();
    }
}
```

ReadWriteLock Example

```
public override void Increment(int threadNum)
{
    // ...
    // method code that doesn't require exclusive access
    rwl.AcquireWriterLock(Timeout.Infinite);
    try
    {
        Console.WriteLine(
         "Start Resource writing (Thread={0}) count: {1}", threadNum, count);
        int tempCount = count;
        Thread.Sleep(1000);
        tempCount++;
        count = tempCount;
        Console.WriteLine(
         "Stop  Resource writing (Thread={0}) count: {1}", threadNum, count);
    }
    finally
    {
        rwl.ReleaseWriterLock();
    }
    // rest of method code that doesn't require exclusive access
}
```

Discussion



- When do you think should we use the pattern
  - Give me some examples
- What do you think are the Pros and Cons
  - Give some more examples

Questions ?