Lecture 6. OOP extras

Programming II

School of Business Informatics
Autumn 2016

(: The 21st century: deleting history is more important than making it :)

Lifetime of a managed object

- After creation the object is placed on the heap
- .NET keeps track of all references to every object. While there is at least 1 reference to an object, it is kept in memory
- When the last reference goes out of scope, an object is marked for deletion
- At a certain point garbage collection takes place, all objects that were marked for deletion are released

Manually disposing objects

- It is not possible to predict in advance when garbage collection will happen
- Sometimes a class holds large blocks of data (e.g. images) or important resources (e.g. database connection) that should be released as soon as possible

```
1  class Data : IDisposable
2  {
3      SomeLargeResource r;
4
5      // Implement dispose logic
6      // ...
7  }
```

IDisposable pattern

See project in the handout:

- Reference **IDisposable** in the class declaration
- Add a "protected virtual Dispose(bool disposing)" method and call it from parameterless Dispose with disposing = true
- Call **GC.SuppressFinalize** to inform the garbage collector that the object was manually cleaned

Best practices with IDisposable

- Free disposable objects as soon as possible
- If a class references disposable objects through its instance fields, implement IDisposable in the class
- Allow Dispose() to be called multiple times and don't throw exceptions

The “using” block


```
1  using (var fs = new FileStream(...))
2  {
3
4  } // fs.Dispose() is called automatically on
      exiting the block
```

Unmanaged resources

- Most classes in the .NET framework allocate memory in the managed area which is controlled by the runtime and the garbage collector
- There are certain cases (e.g. interconnecting with native Windows libraries) when unmanaged resources are required
- The user program is then fully responsible for manually releasing these resources

Finalizers

Finalizers are special methods that are called when the garbage collector cleans the object. They are used to make sure unmanaged resources are released in case Dispose is not called.

```
1   class Data : IDisposable
2   {
3       // Finalizer
4       ~Data()
5       {
6           Dispose(false);
7       }
8
9       protected virtual void Dispose(bool disposing)
10      {
11          // Implement dispose logic
12      }
13  }
```

- If a class has only private constructor(s), it is not possible to instantiate its objects from outside
- To get a class instance a static property is declared

Singleton example


```
1   public class Settings
2   {
3       private Settings() {}
4
5       private static Settings _default;
6
7       public static Settings Default
8       {
9           get
10          {
11              if (_default == null)
12                  _default = new Settings();
13              return _default;
14          }
15      }
16  }
```

Factory pattern

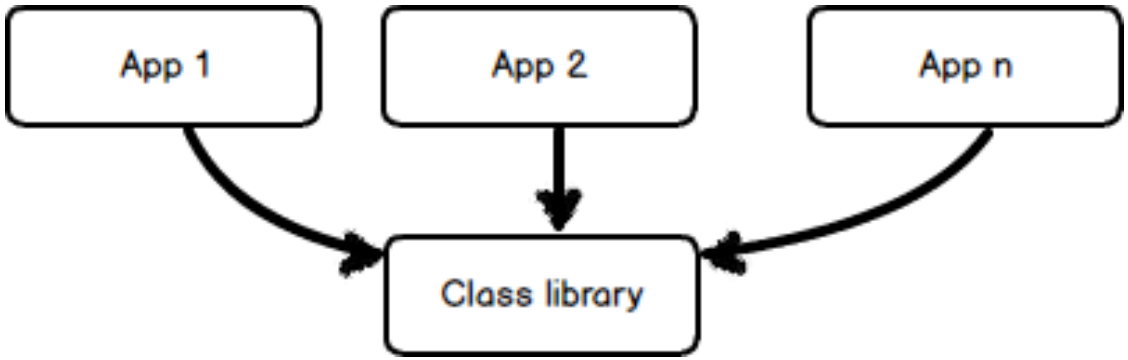
- The goal of the factory pattern is to provide a single place inside the application where concrete classes are instantiated
- Factory methods should return abstractions (interfaces or base classes)
- All application components should depend on these abstractions

Factory pattern example

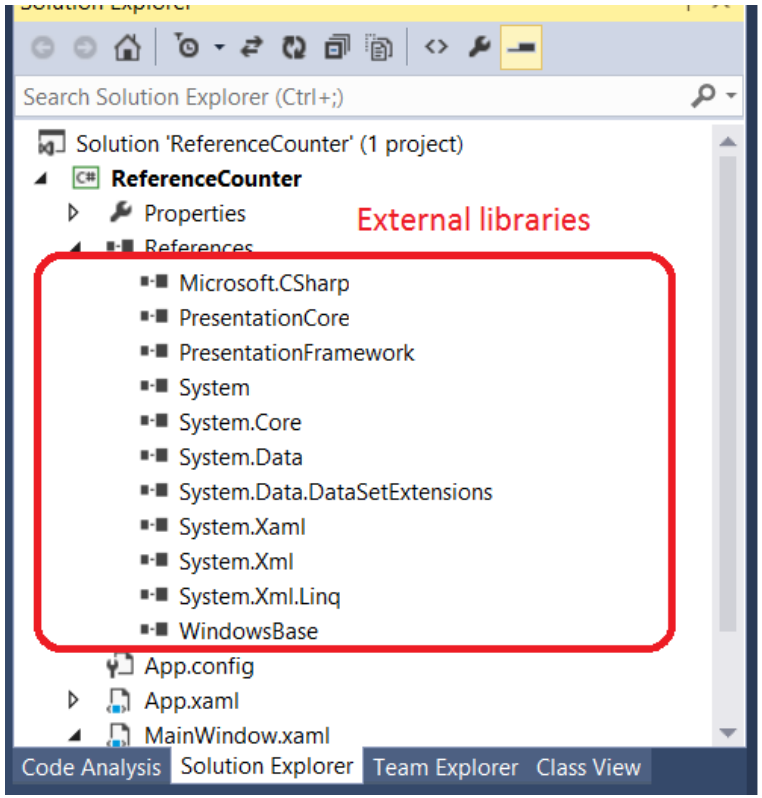

```
1
2   public class Factory
3   {
4       // Factory is usually made as a singleton
5
6       public IRepository GetRepository()
7       {
8           return new DbRepository();
9       }
10
11      public ILogger GetLogger()
12      {
13          return new FileLogger();
14      }
15  }
```

Managing project dependencies

- Each project has dependencies on external libraries (e.g. standard .NET libraries)
- Additional references can be added through the "Project - Add reference" menu
- You can develop your own libraries to accommodate code that will be reused across multiple projects



Libraries and namespaces



Namespace references

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
```

- Don't confuse library and namespace references
- A library can contain multiple namespaces and a single namespace can be split across multiple libraries

NuGet package manager

- Managing external dependencies can be a diffucult task (versioning, inter-dependencies and other common problems)
- To simplify dependency control package managers are used. The basic idea is very similar to mobile app stores - a centralized location for all available packages
- Visual Studio provides an integrated package manager - NuGet (Menu - Project - Manage NuGet packages)

SOLID principles of object-oriented design

- Single responsibility
- Open-closed principle
- Liskov substitution principle
- Interface segregation
- Dependency inversion

Articles by Robert Martin (examples in C++)

Single responsibility principle


```
1   public class Cart
2   {
3       List<OrderItem> _items;
4
5       public void Checkout(PaymentDetails details)
6       {
7           if (!ProcessPayment(details))
8               throw new PaymentException();
9
10          ReserveInventory();
11          NotifyUser();
12      }
13  }
```

The Cart class is responsible for too many things

Providing dependencies

The following example uses a technique known as dependency injection:

```
1 public Cart(IPaymentGateway paymentGateway,
      IInventorySystem inventory, INotificationService
      notificationService)
2 {
3
4 }
```