TEKNOLOGISK
INSTITUT

89076: ”Gang of Four Design Patterns in C#”

Lab Manual

Wincubate ApS

08-09-2019

V2.2

Table of Contents

Exercise types

The exercises in the present lab manual differs in type and difficulty. Most exercises can be solved by applying the techniques from the presentations in the slides in a more or less direct manner. Such exercises are not categorized further.

However, the remaining exercises differs slightly in the sense that they are not necessarily easily solvable. These are categorized as follows:

Labs marked with a single star denote that the corresponding exercises are a bit more loosely specified.

Labs marked with two stars denote that the corresponding exercises contain only a few hints (or none at all!) or might be a bit more difficult or nonessential. They might even require additional searches for information elsewhere than in the slide presentations.

Labs marked with three stars denote that the corresponding exercises are not expected in any way to be solved. These are difficult, tricky, or mind-bending exercises for the interested participants – mostly for fun! ☺

Prerequisites

The present labs require the course files accompanying the course to be extracted in some directory path, e.g.

```
C:\Wincubate\89076
```

with Visual Studio 2017 (or later) installed on the PC.

We will henceforth refer to the chosen installation path containing the lab files as *PathToCourseFiles* .

Module 2: “Abstract Factory”

Lab 02.1: ”Tasty Factories and Products”

This exercise implements all aspects of the Abstract Factory Pattern in an example involving foreign cuisines. The overall structure of the solution will proceed in a manner similar to examples in the module presentation.

- Open the starter project in
  PathToCourseFiles`\Labs\02 - Abstract Factory\Lab 02.1\Starter` ,
  which contains a project called `Cuisines`.

Here you fill in all the additional code needed for implementing Abstract Factory.

Throughout this exercise a “foreign cuisine” (such as Italian or Indian) is an abstract factory interface letting the client create

1. A main course (e.g. pizza)
2. A dessert (e.g. tiramisu)

Consequently, there are two kinds of abstract products in the cuisine abstract factory: MainCourse objects and Dessert objects. These are already defined in the existing projects via the following two definitions:

```
interface IMainCourse
{
    void Consume();
}

interface IDessert
{
    void Enjoy();
}
```

Main courses should have a `void Consume()` method. The intention here is that concrete products should print to the console what is being consumed by the client.

Desserts should have a `void Enjoy()` method. When invoked it should print to the console reflect what is being enjoyed by the client.

You will start by implementing an Italian cuisine using the Abstract Factory Pattern

- Implement a concrete main course product called `Pizza`
  - Its constructor should accept a sequence of topping strings.
- Implement a concrete dessert product called `Tiramisu` (without additional members)
- Create the appropriate abstract factory interface for cuisines called `IMealFactory`.
- Create a concrete factory class for the Italian cuisine, where
  - the main course being created is a pizza with “Tomato Sauce”, “Pepperoni”, “Pineapple”, and “Cheese”

- the dessert is a tiramisu,

- Test your implementation by adding the appropriate client code in Program.cs.
  - Invoke `IMainCourse.Consume()` on the created main course object.
  - Invoke `IDessert.Enjoy()` on the created dessert object.

When you run the program, the output should be the following (or equivalent):

```
C:\WINDOWS\system32\cmd.exe
Delicately consuming Tomato Sauce, Pepperoni, Pineapple, Cheese
Mmmm... Enjoying the coffee-flavoured cake with mascarpone
Press any key to continue . . .
```

You have now implemented the Italian cuisine. You will then proceed to implementing the Indian cuisine as follows.

- Implement a concrete main course product called `ChickenCurry`
  - Its constructor should accept an integer indicating spicyness.
- Implement a concrete dessert product called `Kheer` (without additional members)
- Create the corresponding concrete factory class for the Indian cuisine, where
  - the main course being created is with a spicyness of 5.
- Test your implementation by changing only the Italian cuisine to the Indian cuisine in Program.cs.

When you run the program, the output should now be the following (or equivalent):

```
C:\WINDOWS\system32\cmd.exe
Eating delicious and hoooooot dish with naan and rice
Aaaah... Enjoying nice and soothing sugary rice pudding
Press any key to continue . . .
```

Module 03: “Builder”

Lab 03.1: ”Creating Very Simple Fluent APIs” ()

This exercise illustrates how to create a Fluent API for building pizza products using a variation of the Builder Pattern. Fluent APIs are quite popular in .NET for configuring the Builder instances in a “fluent” fashion, which is reminiscent of the flow in natural, spoken languages.

Consider the `Pizza` class defined as

```
class Pizza
{
    public CrustKind Crust { get; set; }
    public bool HasSauce { get; set; }
    public IEnumerable<ToppingKind> Toppings { get; set; }
    public CheeseKind? Cheese { get; set; }
    public bool Oregano { get; set; }
}
```

Then the well-known Hawaii pizza manually constructed in the following manner:

```
Pizza hawaii = new Pizza
{
    Crust = CrustKind.Classic,
    HasSauce = true,
    Cheese = CheeseKind.Regular,
    Toppings = new List<ToppingKind>
    {
        ToppingKind.Ham,
        ToppingKind.Pineapple
    },
    Oregano = true
};
```

could be built using an appropriate fluent API Builder as follows:

```
FluentPizzaBuilder builder = new FluentPizzaBuilder();
Pizza hawaii = builder
    .Begin()
    .WithCrust(CrustKind.Classic)
    .Sauce
    .AddCheese()
    .AddTopping(ToppingKind.Ham)
    .AddTopping(ToppingKind.Pineapple)
    .Oregano
    .Build();
```

Your task is now to create this `FluentPizzaBuilder` class.

- Open the starter project in
  PathToCourseFiles`\Labs\03 - Builder\Lab 03.1\Starter` ,

which contains a project with the `Pizza` class and related types.

- Create the `FluentPizzaBuilder` class.
- Test that your class in the Fluent API definition correctly build a Hawaii pizza instance equivalent to the manually created instance above.

Lab 03.2: "A Much Better Fluent API" (★★★)

This exercise examines how to create a better Fluent API for building pizza products.

The Fluent API solution to Lab 03.1 is simple and not too difficult to create with a little bit of practice. But it is too simplistic for professional purposes due to a number of problems:

1. Any order of invoking the fluent methods is allowed
2. Repetitions of the fluent methods are allowed
3. All methods are essentially optional (as well as repeatable)
4. It uses properties containing getters with side effects.

Your task is now to remedy all these deficiencies.

- Open the starter project in
  PathToCourseFiles`\Labs\03 - Builder\Lab 03.2\Starter` ,
  which contains a project with the solution to Lab 03.1.
- You should create a better fluent API solution, which is statically safe in the sense that the compiler will only allow fluent method sequences which are legal.

More specifically, this sequence should be allowed by the compiler:

```
Pizza hawaii = new FluentPizzaBuilder()
    .Begin()
    .WithCrust(CrustKind.Classic)
    .WithSauce()
    .AddCheese()
    .AddTopping(ToppingKind.Ham)
    .AddTopping(ToppingKind.Pineapple)
    .WithOregano()
    .Build();
```

This sequence should also be allowed by the compiler:

```
Pizza hawaii = new FluentPizzaBuilder()
    .Begin()
    .WithCrust()
    .WithNoSauce()
    .AddCheese()
    .AddTopping(ToppingKind.Ham)
    .AddTopping(ToppingKind.Pineapple)
    .Build();
```

However, this sequence should **not** be allowed by the compiler:

```
Pizza hawaii = new FluentPizzaBuilder()
    .Begin()
    .WithOregano()
    .WithCrust(CrustKind.Classic)
    .WithSauce()
    .AddCheese()
```

```
        .AddTopping(ToppingKind.Ham)
        .AddTopping(ToppingKind.Pineapple)
        .Build();
```

Make sure that:

- Building always begins with `Begin()`
- Building always completes with `Build()`
- Proper defaults are chosen, e.g for `WithCrust()`
- Some methods are optional choices, e.g. `WithOregano()`
- Some choices are mandatory, e.g. `WithSauce()` vs. `WithNoSauce()`
- The compiler allows only correct sequences, which are in the “usual” order:
    - Crust,
    - Sauce,
    - Cheese,
    - Any sequence of toppings, and finally
    - Oregano

Module 04: “Factory Method”

Lab 04.1: ”Factory Method for Pizza Creation”

In order to illustrate how Builder and Factory Method compares, we will in this exercise implement construction of concrete instances of the `Pizza` example of Module 03 – but this time using the Factory Method pattern.

- Open the starter project in
  PathToCourseFiles`\Labs\04 – Factory Method\Lab 04.1\Starter` ,
  which contains a project with the wellknown `Pizza` class from Module 03.

Moreover, the Program.cs file contains two concrete instances of `Pizza`; a Pepperoni `Pizza` and a Hawaii `Pizza`.

- Refactor the existing `Pizza` class to implement the Factory Method pattern instead
  - Add all the classes necessary.

Lab 04.2: "Combining Factory Patterns" (⭐⭐)

One often encounters various combinations of the Factory Method, Builder, and Abstract Factory patterns in everyday programming. Such hybrids can be quite helpful in creating "intelligent" factories.

In this exercise, we will investigate a mix between Factory Method and Abstract Factory which is sometimes employed in code to avoid compile-time dependencies to concrete classes. Below we will use the `System.Reflection` API to instantiate objects from a string description of the concrete object type.

- Open the starter project in
  PathToCourseFiles`\Labs\04 – Factory Method\Lab 04.2\Starter` ,
  which contains a project with several predefined `Pizza` classes.

The main `Pizza` class and related types are identical their definitions introduced in Lab 03.1. However, several additions have been made:

- An interface `IPizza` has been added to describe an abstract `Pizza` product
  - `Pizza` now implements this interface in order to be a concrete product.
- A number of concrete `Pizza` product classes have been added:
  - `ElDiabloPizza`
  - `HawaiiPizza`
  - `MargheritaPizza`
  - `MeatLoverPizza`

  These are all ready to be instantiated by an Abstract Factory.

The following interface is already defined:

```
interface IPizzaFactory
{
    IPizza Create( string description );
}
```

Your task is now to create a concrete class implementing the Abstract Factory interface with a single Factory Method accepting a string parameter which describes the concrete IPizza object to create.

- Create a class `ReflectionPizzaFactory` implementing `IPizzaFactory` such that it processes the incoming description string and instantiates the corresponding concrete class, if the following statements are true
  - The "cleaned up" description string is the name of a type existing in the currently executing assembly.
  - The existing type implements the `IPizza` interface.

  To be more precise,
  - `"margherita pizza"` will instantiate the type `MargheritaPizza`
  - `"meat lover pizza"` will instantiate the type `MeatLoverPizza`
  - `"El Diablo pizza"` will instantiate the type `ElDiabloPizza`
  - `"Hawaii Pizza"` will instantiate the type `HawaiiPizza`

Note: Google is probably your friend when it comes to Reflection Kung-fu…!

As you complete your implementation;

- Test that your implementation works correctly by modifying the code in Program.cs accordingly.

When you run the program, the output should be the following:

C:\WINDOWS\system32\cmd.exe

```
Classic pizza with sauce with regular cheese. Toppings:

Classic pizza with sauce with extra cheese. Toppings:
        Bacon
        Pepperoni
        Sausage
with oregano on top

Classic pizza with sauce with regular cheese. Toppings:
        Mushrooms
        Jalapenos
        Tomatoes
        Onions
        Sausage
        BlackOlives
with oregano on top

Classic pizza with sauce with regular cheese. Toppings:
        Ham
        Pineapple
with oregano on top

Press any key to continue . . .
```

Module 05: “Prototype”

Lab 05.1: ”Implementing ICloneable”

This exercise implements the Prototype pattern for resource-intensive objects by implementing `ICloneable`.

- Open the starter project in
  PathToCourseFiles`\Labs\05 - Prototype\Lab 05.1\Starter` ,
  which contains a project with the `LinkScraper` and `LinkInfo` classes.

The `LinkScraper` class is created by supplying a URL pointing to an HTML resource to be analyzed for references. It is used as follows:

```
LinkScraper original = new LinkScraper("http://www.jp.dk");
await original.ScrapeAsync();

foreach (LinkInfo info in original.AnalysisResult)
{
    Console.WriteLine(
       $"Reference to {info.Href} located at index {info.Index}"
    );
}
```

Upon invoking the `LinkScraper.ScrapeAsync()` the HTML page at the specified URL is downloaded and a detailed link analysis report is computed. In this sense, it is quite resource-intensive to retrieve these analysis results. Consequently, it makes sense to use the Prototype pattern to be able to create an efficient copy of `LinkScraper` master objects without revisiting the webpage and recomputing the result.

- Implement the Prototype pattern for `LinkScraper` class.
- Create a unit test (or integration test) proving that your implementation is correct.

Module 06: "Singleton"

Lab 06.1: "Making Classes Thread-safe Singletons"

This exercise implements the Singleton pattern in a thread-safe manner for an existing class.

- Open the starter project in
  PathToCourseFiles`\Labs\06 - Singleton\Lab 06.1\Starter` ,
  which contains a project with the `ConsoleLogger` class.

The `ConsoleLogger` class is essentially a very simple logger, which writes single lines to the console a color dependent upon the log line's severity status. It is used as follows:

```
ConsoleLogger logger1 = new ConsoleLogger();
logger1.Log( "Hey! We're running fine" );

ConsoleLogger logger2 = new ConsoleLogger();
logger2.Log( "Houston, we have a problem...!" + Environment.NewLine,
        true);

Console.WriteLine( $"Lines logged: {logger1.LinesLogged +
        logger2.LinesLogged}");
```

Your task is to make the `ConsoleLogger` class a thread-safe singleton, thus making the above code nicer.

- Change the `ConsoleLogger` class to make it a <u>thread-safe</u> singleton.
  - Choose whichever Singleton implementation you prefer as long as it meets the specification.
  - Mind the `LinesLogged` property!
- Rewrite the code in Program.cs accordingly to make use of the modified `ConsoleLogger` implementation.
- Test that your implementation works correctly.

Module 07: “Adapter”

Lab 07.1: ”Adapting to a Simple Web Shop”

This exercise implements an Adapter for a pre-specified API definition for a very simple web shop.

- Open the starter project in

  PathToCourseFiles`\Labs\ 07 - Adapter\Lab 07.1\Starter` ,

  which contains a project with the `InventoryClient` class.

The `InventoryClient` class makes use of an `IInventoryRepository` instance in order to retrieve and display inventory information from a web shop back-end. Unfortunately, the web shop back-end is based on a commercial-off-the-shelves product with an API that cannot be changed.

The back-end supplies inventory information in the shape of a `ProductRepository` class which must be instantiated. This is the only way to retrieve information about the current line of products, unfortunately.

- You need to adapt the `ProductRepository` back-end to the `InventoryClient` front-end use without changing either class.
- Modify the code in Program.cs appropriately;

```
InventoryClient client = new InventoryClient( ... );
client.DisplayInventory();
```

When you run the program, the output should be the following:

```
C:\WINDOWS\system32\cmd.exe
Category: Hardware
        "HoloLens" by Microsoft
        "Switch" by Nintendo
        "Switch Controller" by Nintendo

Category: Software
        "Business Suit Boba Goes Looking for Love in Aldaraan Places" by Wincubate Games

Category: Book
        "Design Patterns in C#" by Gang of One
        "How to survive without Internet" by Chris MacDonald

Press any key to continue . . .
```

Module 08: “Bridge”

Lab 08.1: ”Refactoring Shapes and Visualizations to Bridge” (⭐⭐)

This exercise improves an existing solution by introducing the Bridge Pattern for an added level of abstraction.

Note: The exercise is probably mostly for developers with a basic knowledge of Windows Forms!

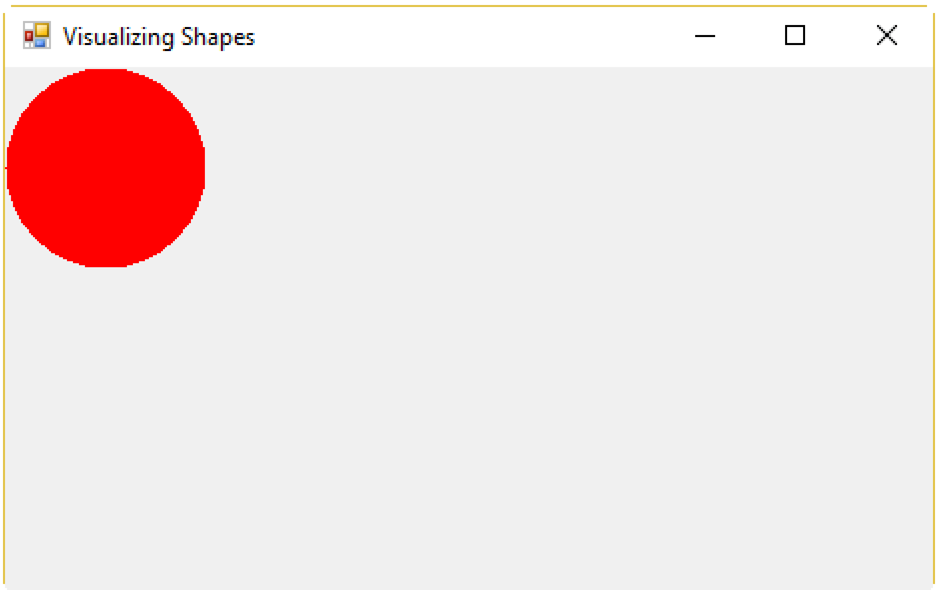
- Open the starter project in
  PathToCourseFiles`\Labs\ 08 - Bridge\Lab 08.1\Starter` ,
  which contains a project with the four concrete classes with a common abstract base class called `Shape`.

Unfortunately, you will soon discover that these classes are a complete mess! There are four concrete classes which you can instantiate inside of the `ShapeForm` constructor.

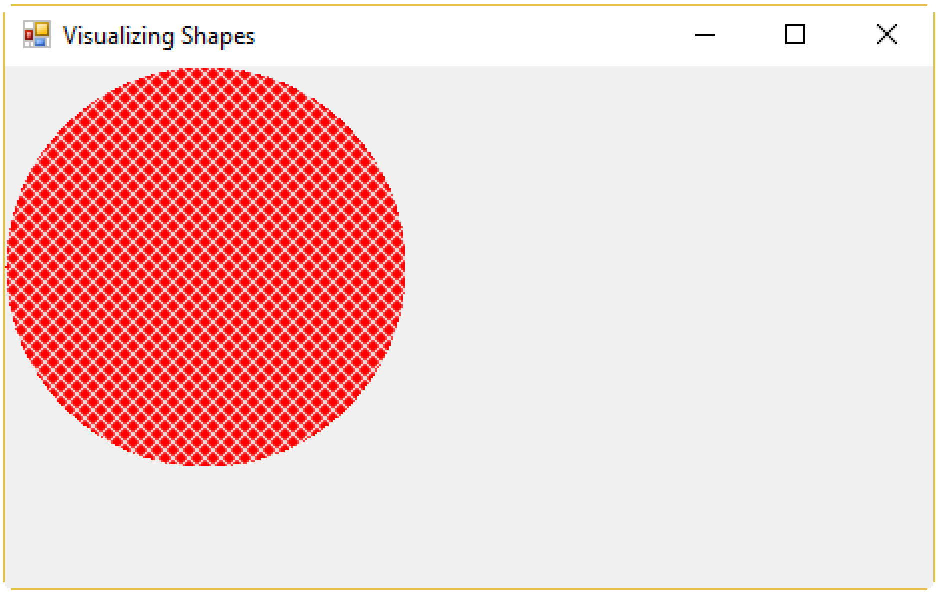
- Locate the following line of the `ShapeForm` constructor:

```
// TODO: Try different versions to see result
```

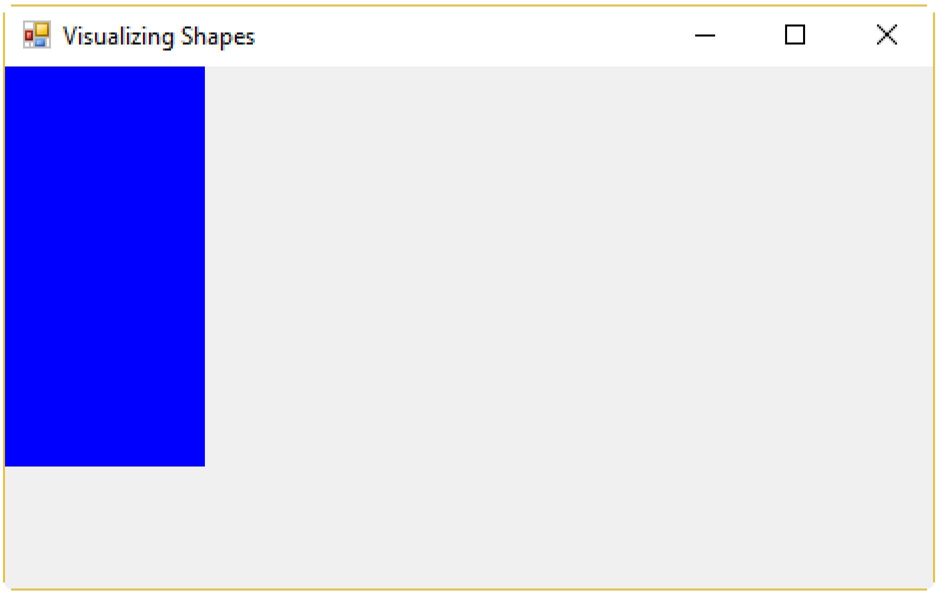
  In turn, run each of the four concrete shape examples. They will produce the following results:
  - `new SolidColorCircle( Color.Red, 100 )`:



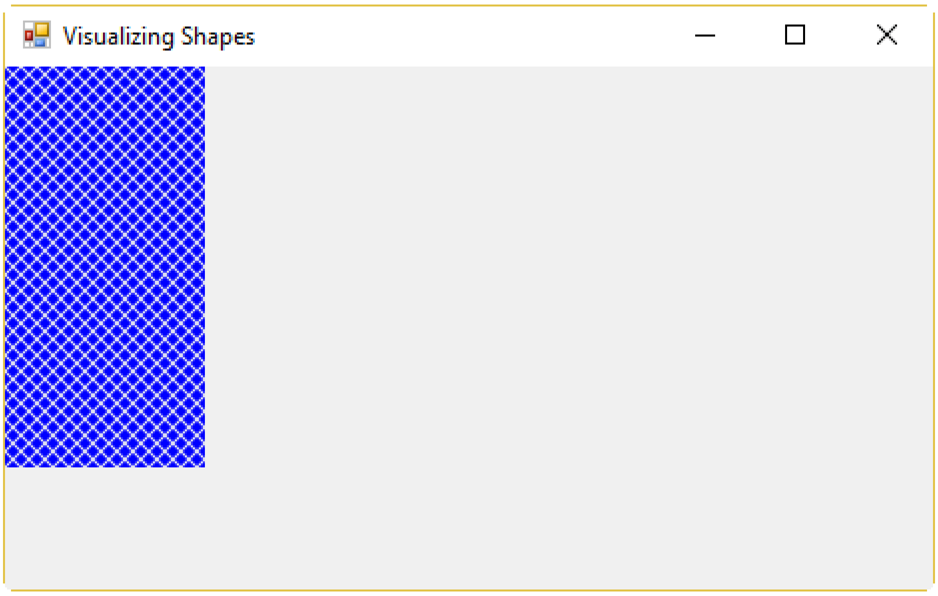
- `new HatchedCircle( Color.Red, 200 ):`



- `new SolidColorRectangle( Color.Blue, 100, 200 ):`



- `new HatchedRectangle(Color.Blue, 100, 200)`:

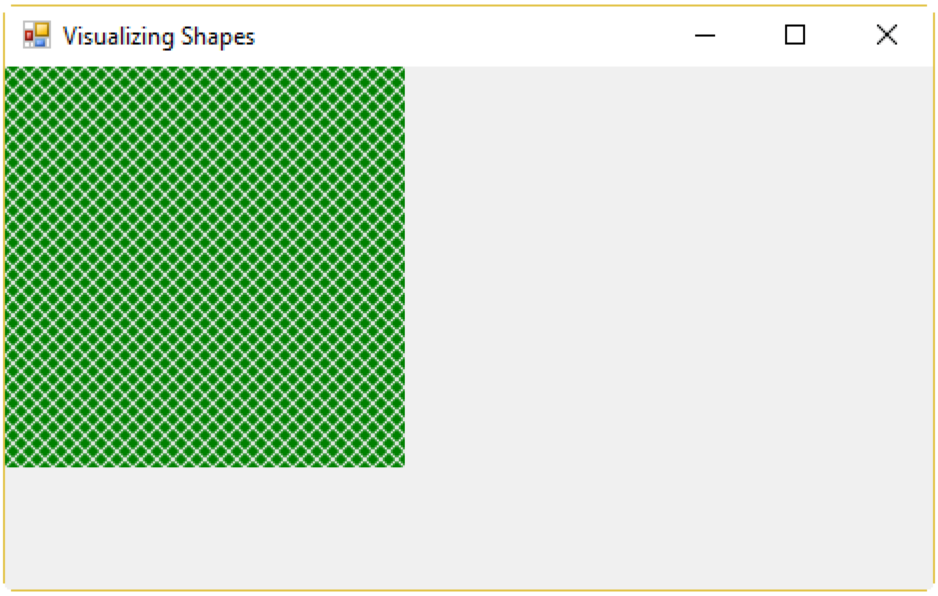


There is no need for four concrete classes…! Moreover, the way these classes are constructed makes it hopelessly cumbersome to a new shape , e.g. `Square`, or a new way of visualizing the shapes, e.g. `GradientXxx`.

- Fix the depressing design by refactoring the existing code to the Bridge Pattern. Your completed solution should have the following characteristics:
  - The functionality is preserved
  - The `Shape` hierarchy should consist of only the logical shapes with two concrete classes:
    - `Shape`
      - `Circle`
      - `Rectangle`
  - The visualization aspects such as `Color`, `SolidColor`*`Xxx`*, and `Hatched`*`Xxx`* are moved to a new hierarchy.
    - Note: `Color` should no longer be part of the `Shape` hierarchy.
- Run your refactored program and check that with the new (and vastly better!) structure you still get the same visualizations of the shapes as specified above.

Your design now has two levels of abstraction. Celebrate this by showcasing just how elegant it now is to add more shapes.

- Add a new shape called `Square` at the appropriate place in the `Shape` hierarchy.
  - A `Square` of 200 visualized in green, hatched style should look like as follows:

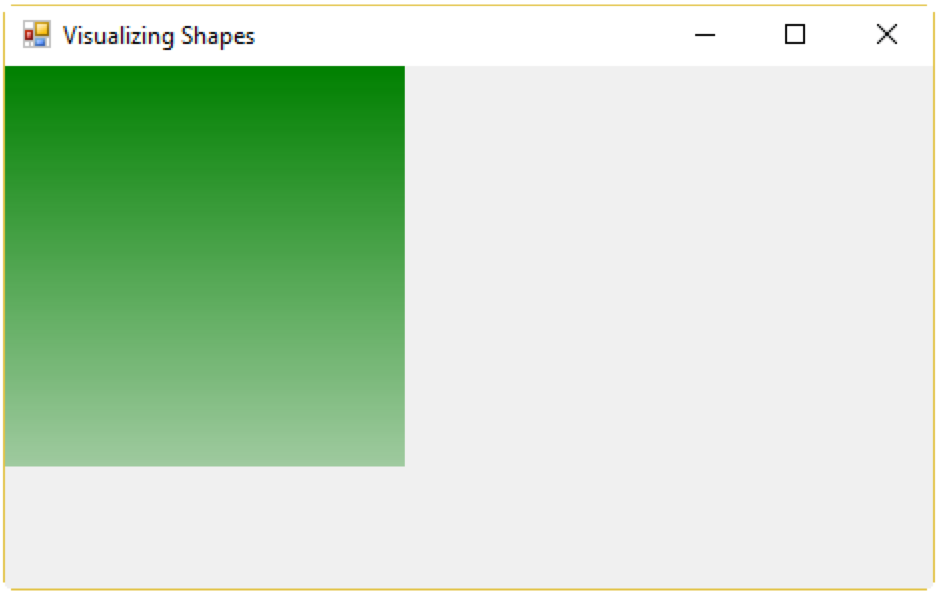


That was perhaps surprisingly simple…! Now add a new style of visualization, which is equally beautiful.

- Add a new style of visualization `Gradient` where the shape is painted with a gradient brush
    - Note: You can obtain use a `LinearGradientBrush` specified as follows:

```
new LinearGradientBrush(
    new PointF( 0, 0 ),
    new PointF( 0, 300 ),
    color,
    Color.Transparent
)
```

    - A `Square` of 200 visualized in green, gradient style should look like as follows:



Module 09: “Composite”

Lab 09.1: ”Wedding Gift Sharing with Composite Pattern” (★)

This exercise uses the Composite Pattern for group-based sharing of wedding gift expenses.

- Open the starter project in
  PathToCourseFiles\Labs\ 09 - Composite\Lab 09.1\Starter ,
  which contains a project with the Person class.

The Person class contains information about what the specified person must pay to contribute to the wedding gift. It is specified as follows:

```
class Person
{
    public string Name { get; set; }
    public decimal MustPay { get; set; }

    public override string ToString() => $"{Name} pays {MustPay:c}";

    public Person( string name ) => Name = name;
}
```

Program.cs defines 9 persons and contains a brutally simple sharing algorithm for sharing the wedding gift expenses: The expenses are shared equally among all contributing persons!

```
List<Person> participants = new List<Person>
{
    noah,
    frederikke,
    ane,
    jesper,

    peter,
    malene,

    thomas,
    rasmus,
    mads
};

decimal giftPrice = 2500;

// Equal sharing among all participants
foreach (Person person in participants)
{
    person.MustPay = giftPrice / participants.Count;
}
```

For a gift of DKK 2.500 the algorithm produces the following output:

C:\WINDOWS\system32\cmd.exe

```
Noah pays 277,78 kr.
Frederikke pays 277,78 kr.
Ane pays 277,78 kr.
Jesper pays 277,78 kr.
Peter pays 277,78 kr.
Malene pays 277,78 kr.
Thomas pays 277,78 kr.
Rasmus pays 277,78 kr.
Mads pays 277,78 kr.
Press any key to continue . . .
```

- Use the Composite Pattern to modify the algorithm such all families can take part in sharing the expenses in such a way that they <u>together</u> contribute as a single individual.
    - Add the appropriate interfaces and class to implement Composite
    - Change Program.cs accordingly to test your implementation.
- Specifically, when splitting the 9 pre-existing persons into 2 groups and 3 individuals as follows:
    - Group 1 consists of Noah, Frederikke, Ane, and Jesper
    - Group 2 consists of Peter and Malene
    - Thomas participates as an individual
    - Rasmus participates as an individual
    - Mads participates as an individual,

  the results of the modified sharing algorithm should be:

C:\WINDOWS\system32\cmd.exe

```
Noah pays 125,00 kr.
Frederikke pays 125,00 kr.
Ane pays 125,00 kr.
Jesper pays 125,00 kr.
Peter pays 250,00 kr.
Malene pays 250,00 kr.
Thomas pays 500,00 kr.
Rasmus pays 500,00 kr.
Mads pays 500,00 kr.
Press any key to continue . . .
```

Module 10: “Decorator”

Lab 10.1: ”Profiling with Decorators”

This exercise uses the Decorator Pattern for adding a timing aspect to an existing component whose source code we have no access to.

- Open the starter project in
  PathToCourseFiles`\Labs\ 10 - Decorator\Lab 10.1\Starter` ,
  which contains two projects:
  - An external library with the interface `IComputeOperation` and concrete class `ComputeOperation`
  - A console application invoking the library.

It seems that the library computes rather slowly when invoked as in Program.cs. We would very much like to instrument the `ComputeOperation` class with code for starting and stopping a `Stopwatch` to be able to measure the execution time of the `Compute()` method.

Unfortunately, this is an external library for which we cannot modify the source code, so we need to figure out an alternate way of decorating the existing library with `Stopwatch` measurements.

- Define an appropriate abstract `ComputeOperationDecorator` class for `IComputeOperation` .
- Write a concrete `Timing` decorator deriving from `ComputeOperationDecorator`.
- Test your implementation by modifying Program.cs accordingly.

When you run your complete solution the last part of the output produced should be similar to the following:

```
86
88
90
92
94
96
98
Execution time was 00:00:05.5876512
Press any key to continue . . .
```

Module 11: "Façade"

Lab 11.1: "Provide a Price Search Façade" ( )

This exercise uses the Façade Pattern for providing a simple price lookup service for a very simple web shop.

- Open the starter project in
  PathToCourseFiles`\Labs\ 11 - Facade\Lab 11.1\Starter` ,
  which is essentially based on the simple web shop of Lab 07.1 augmented with pricing and currency conversion abilities. The contains three projects:
  - A `Financial` library providing currency conversion among the DKK, USD, and GBP currencies through the `CurrencyConversionService` class.
  - A WebShop library providing
    - a `ProductRepository` class providing information about products in the web shop
    - a `PriceInfoRepository` class providing pricing information (<u>in USD</u>) on products in the web shop given a specified `ProductId`
  - An empty `WebShopPriceSearch` console application.

Your task is to create a `PriceSearch` facade class allowing clients simple access to search for a simple substring to access pricing information <u>in DKK</u> for the product names matching the specified search string.

<u>Note</u>: You're not allowed to change any existing classes in `Financial` or `WebShop`.

- Provide a class `PriceSearch` which exposes the following method for clients to call

```
class PriceSearch
{
    ...
    public IEnumerable<PriceSearchInfo> Lookup( string searchName );
    ...
}
```

  The `PriceSearchInfo` structure is given as

```
struct PriceSearchInfo
{
    public string ProductName { get; set; }
    public decimal PriceDkk { get; set; }
}
```

- Test your implementation by commenting in the lines in the predefined Program.cs accordingly. The following code fragment:

```
PriceSearch ps = new PriceSearch();
IEnumerable<PriceSearchInfo> searchInfos = ps.Lookup("Switch");

foreach (PriceSearchInfo si in searchInfos)
{
```

```
            Console.WriteLine( $"{si.ProductName} costs DKK {si.PriceDkk:f2}" );
        }
```

should produce the following output:

Module 12: “Flyweight”

Lab 12.1: ”Finish the Coffee Brewing Flyweight Example” (⭐⭐)

This exercise completes the Coffee example used as the running examples in the presentation.

- Open the starter project in
  PathToCourseFiles`\Labs\ 12 - Flyweight\Lab 12.1\Starter` ,
  which is essentially the Coffee Brewing example from the presentation.

Make the Coffee objects proper value objects

Each class in the Coffee class hierarchy is already immutable to allow sharing of the intrinsic objects among several clients. However, .NET guidelines suggest that when implementing such immutable “value objects” representing the same values, you should ensure such coffee objects are always deemed equal.

- Override the equality operators == and `!=` as well as the the `Equals()` method to ensure two identical `Coffee` instances are compared correctly when compared using the operators or `Equals() method`.
- Override the `GetHashCode()` method to reflect that two identical `Coffee` instances are hashed correctly in dictionaries.

Make the CoffeeFactory thread-safe

The individual `Coffee` instances are already immutable, and hence thread-safe. However, the object instantiation inside the `CoffeeFactory` is not.

- Make the flyweight object instantiation in `CoffeeFactory` thread-safe by whichever means you prefer.

Module 13: “Proxy”

Lab 13.1: ”Implementing Caching as a Proxy” (★★/★★★)

This exercise extends the Web Shop example from the presentation by implementing product caching using the Proxy pattern.

- Open the starter project in
  PathToCourseFiles`\Labs\13 - Proxy\Lab 13.1\Starter`,
  which essentially contains the project of the presentation containing the `Product`, `ProductRepository`, and related classes and interfaces.

The starter project above contains some initial code in Program.cs, which retrieves the product objects with ID 1 and 3 a number of times. Unfortunately, the existing `ProductRepository` is rather slow at retrieving single `Product` instances by ID in the `GetById()` method.

A test execution of the existing implementation reveals the following depressing output:

```
C:\WINDOWS\system32\cmd.exe
Fetching Product ID 1 from ProductRepository in 78 ms
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 1 from ProductRepository in 321 ms
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 3 from ProductRepository in 615 ms
[3] Book: "Design Patterns in C#" by Gang of One
Fetching Product ID 1 from ProductRepository in 375 ms
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 1 from ProductRepository in 355 ms
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 1 from ProductRepository in 468 ms
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 3 from ProductRepository in 252 ms
[3] Book: "Design Patterns in C#" by Gang of One
Press any key to continue . . .
```

Your task is to improve upon these execution times by creating a caching Proxy called `CachingProductRepository` which caches the retrieved products instances such that multiple lookups of the same ID result in only a single lookup in the original `ProductRepository`.

- Create a Proxy class `CachingProductRepository` such that the `GetById()` method caches duplicate lookups
    - Focus on only the `GetById()` method
    - **You can disregard caching in the `GetAll()` method for now**.
- Activate the proxy class in Program.cs and run your application.
- You should see a result along the lines of the following:

```
Fetching Product ID 1 from ProductRepository in 642 ms
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 3 from ProductRepository in 301 ms
[3] Book: "Design Patterns in C#" by Gang of One
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
[3] Book: "Design Patterns in C#" by Gang of One
Press any key to continue . . .
```

Note:

- You're not allowed to change any existing classes in the `WebShop` class library, of course.
- Don't worry about thread-safety.
- It might be helpful to use the `MemoryCache` class which is referenced in .NET Core 2.0 by adding the nuget package `Microsoft.Extensions.Caching.Memory` to the existing client project
  - You can assume that the underlying `ProductRepository` is only accessed through the caching Proxy so that cache never needs to be invalidated.
  - You don't need to implement `IDisposable`. This will be the topic of a later module!

If time permits…

- How would the `CachingProductRepository` proxy work in conjunction with the `AdministratorsOnlyProxyRepository`?
- How would you implement caching in the `GetAll()` method?

If this is implemented correctly then adding the following lines

```
IEnumerable<Product> all = repository.GetAll();
IEnumerable<Product> all2 = repository.GetAll();
```

before all the `GetById()` invocations in Program.cs would produce a result similar to

```
Fetching all products from ProductRepository in 863 ms
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
[3] Book: "Design Patterns in C#" by Gang of One
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
[3] Book: "Design Patterns in C#" by Gang of One
Press any key to continue . . .
```

Module 14: “Iterator”

Lab 14.1: ”Making an Existing Linked List implement the Iterator Pattern” ()

This exercise will implement the Iterator pattern on a pre-existing, but non-generic, linked list.

- Open the starter project in
  PathToCourseFiles`\Labs\14 – Iterator\Lab 14.1\Starter` ,
  which contains the non-generic, `LinkedList` class.

Take a bit of time to familiarize yourself with the pre-existing code in the project code above. Now,

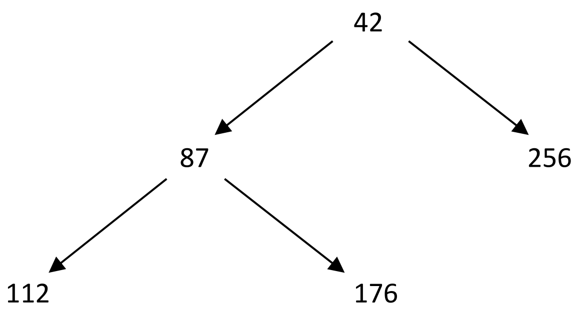
- Make the `LinkedList` class generic such that it stores elements of type T in `Node`.
- Make your newly created `LinkedList`<T> class implement `IEnumerable`<T>.
- Create a list of the elements 42, 87, 112, 176, and 255 and use `foreach` to print all the elements to the console.

Lab 14.2: "Iterator Pattern for Composites" (★★★)

In this exercise we will implement the Iterator pattern on a binary tree with a "Composite" structure.

- Open the starter project in
  PathToCourseFiles`\Labs\ 14 - Iterator\Lab 14.2\Starter` ,
  which contains a Node class.

Instances of Node are binary trees such as the following instance:



Every node in the tree has an integer value. In this manner, the tree above essentially represents the sequence of elements specified by

42, 87, 112, 176, 256.

Your task is now to implement the Iterator pattern on the Node class by creating a "recursive" enumerator.

- In Program.cs create an expression of type Node reflecting the binary tree depicted above.
- Implement the generic `IEnumerable` cleverly such that the enumeration produced on the tree above is exactly the specified sequence 42, 87, 112, 176, 256.
- Test that your implementation in Program.cs to ensure that it is correct.

Module 15: “Chain of Responsibility”

Lab 15.1: ”Evaluating Poker Hands” ()

This exercise will implement the Chain of Reponsibility pattern to evaluate the scoring of a 5-card poker hand.

- Open the starter project in
  PathToCourseFiles`\Labs\ 15 - Chain of Responsibility\Lab 15.1\Starter` ,
  which contains the well-known types, which you have seen before:
  - `Suit`
  - `Rank`
  - `Card`
  - `Deck`

Moreover, it contains a new class Hand which corresponds to a poker hand of up to 5 cards built successively by picking up dealt cards repeatedly such as e.g.

```
Deck deck = new Deck();
deck.Shuffle();

Hand hand = new Hand();
for (int i = 0; i < 5; i++)
{
    Card card = deck.Deal();
    hand.PickUp(card);
}

Console.WriteLine(hand);
```

Running the above program might for instance produce the sequence

```
C:\WINDOWS\system32\cmd.exe
/Jh Js Ah As Ac/
Press any key to continue . . .
```

- Take a careful look at the source code for the Hand class.
  - Note: The cards of the Hand are always stored in a fashion sorted by card Rank.

Your task to provide the correct poker scoring evaluation of the specified Hand instance using a chain of `IHandEvaluator` instances as prescribed by the Chain of Responsibility pattern.

The overall result of the hand evaluation is a value of type `HandEvaluation` specified as follows:

```
enum HandEvaluation
{
    HighCard,
    Pair,
    TwoPairs,
    ThreeOfAKind,
```

```
        Straight,
        Flush,
        FullHouse,
        FourOfAKind,
        StraightFlush,
        RoyalFlush
    }
```

The scoring of Poker is as follows:

> *The hand with the highest card wins unless one hand has a pair. The highest pair wins unless one hand has two pair. The hand with the highest "higher pair" wins unless one hand has three of a kind. Three of a kind is beaten by a straight (5 cards in sequence such as e.g. 5,6,7,8,9). A straight is beaten by a flush (5 cards in the same suit). A flush is beaten by a full house (three of one kind, a pair of another). A full house is beaten by four of a kind. Four of a kind is beaten by a straight-flush (5 cards in sequence and all of the same suit), and the highest hand is a royal flush (the ace-high version of the straight flush).*

A skeleton implementation of the `IHandEvaluator` interface already exists, but is not complete as it lacks a second method.

- Complete the `IHandEvaluator` interface by adding a method signature corresponding to a single hand evaluation being computed.

A skeleton implementation of an abstract `HandEvaluatorBase` class implementing the `IHandEvaluator` interface already exists. This is also not complete as it lacks the abstract method signature for the method you just added to `IHandEvaluator`

- Complete the `HandEvaluatorBase` interface by adding an abstract method signature corresponding to a single hand evaluation being computed.
    - Note: There are a number of helper methods available for use in the concrete subclasses.

For instance, the code to decide if a `Hand` instance is to be scored as a `RoyalFlush` could be

```
HasStraight(hand) && HasFlush(hand) && hand.HighestCardRank == Rank.Ace
```

You will need to implement a concrete `IHandEvaluator` class for each of the `HandEvaluation` enumeration members by deriving each such class from the abstract `HandEvaluatorBase`.

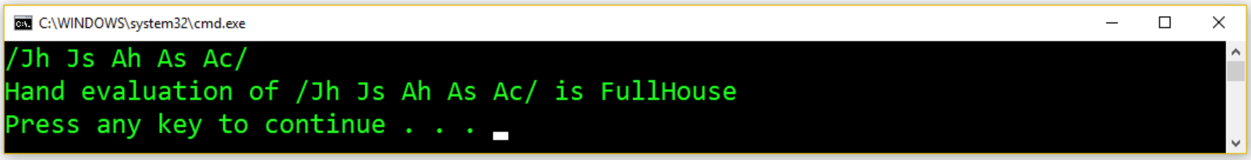
- Complete the `RoyalFlushEvaluator` class.
- Complete the `StraightFlushEvaluator` class.
- Complete the `FourOfAKindEvaluator` class.
- Complete the `FullHouseEvaluator` class.
- Complete the `FlushEvaluator` class.
- Complete the `StraightEvaluator` class.
- Complete the `ThreeOfAKindEvaluator` class.
- Complete the `TwoPairsEvaluator` class.

- Complete the `PairEvaluator` class.
- Complete the `HighCardEvaluator` class.

You’re almost there! Now,

- Complete the `HandEvaluatorChainFactory` class by setting up the chain of responsibility using all the created `IHandEvaluator` implementations inside the `Create()` method.
- Finish your implementation by printing the hand evaluation result of hand to the console.

When you run your completed program the output should be reminiscent of



Module 16: "Template Method"

Lab 16.1: "More Pretty-printing of Persons"

This exercise will implement another two concrete algorithms in the Template Method patterns setup from the module presentation.

- Open the starter project in
  PathToCourseFiles`\Labs\16 - Template Method\Lab 16.1\Starter` ,
  where the `Library` project contains the well-known, reusable types similar to what you have seen before:
    - `IPrettyPrinter`
    - `PrettyPrinterBase`
    - `XmlPrettyPrinter`
    - `JsonPrettyPrinter`

Firstly;

- Implement a new class `IniFilePrettyPrinter` deriving from `PrettyPrinter` printing the data in the style of the "good", old Windows .ini-files as follows:

```
[Person]
FirstName=Terry
LastName=Tate
Occupation=Office Linebacker
```

- Run your implementation and test that it prints in the format described above.

Secondly, you're not entirely happy with the `JsonPrettyPrinter` class as it prints too compactly, which is not-so-pretty:

```
{"FirstName":"Terry","LastName":"Tate","Occupation":"Office Linebacker"}
```

- In the console application project, create an appropriate new class `JsonPrettyPrettyPrinter` which instead prints the JSON objects as follows:

```
{ "FirstName": "Terry",
  "LastName": "Tate",
  "Occupation": "Office Linebacker" }
```

    - Which approach would you prefer? Deriving from the `PrettyPrinter` class or the `JsonPrettyPrinter` class? Why?
- Test your implementation accordingly.

Module 17: “Strategy”

Lab 17.1: ”Refactoring Deck Storage to Strategy” (★)

This exercise will refactor an existing implementation to the Strategy pattern.

- Open the starter project in
  PathToCourseFiles`\Labs\ 17 - Strategy\Lab 17.1\Starter` ,
  which contains a version of the `Deck` class of Lab 15.1.

The present version of `Deck` has been extended with methods for loading and saving the state of a `Deck` object by means of either binary or JSON files in the methods

```
class Deck : IEnumerable<Card>
{
    private List<Card> _cards;

    ...

   #region Load and Save

   public void Load( StorageFormat format )
   {
       ...
   }
   public void Save( StorageFormat format )
   {
       ...
   }

   #endregion
}
```

The `Deck` class now contains a multitude of bad aspects, e.g.

- The class has a more than one responsibility.
- The `Load()` and `Save()` methods both contain (de)serialization details which has no justification to be located within `Deck` itself.
- The `Deck` class now needs to change when a new storage format is introduced.
- There is a very high degree of coupling between unrelated parts.
- Also, it is possible to load a `Deck` object from a binary file and saving it as JSON (and vice versa).

Your task is to alleviate the above deficiencies by refactoring the existing code for storage mechanisms.

- Refactor the existing code to employ the Strategy pattern
  - Fix the storage strategy at construction time to prevent mixing storage strategies.
  - Don’t spend time on additional error handling..!
- Test your refactoring by modifying the setup code in Program.cs.

Rest assured that the world has now become a vastly better place… 😊

Module 18: “Memento”

Lab 18.1: ”Restore Deck State with Memento” (★)

This exercise will implement the Memento pattern of an existing set of types.

- Open the starter project in

  PathToCourseFiles`\Labs\ 18 - Memento\Lab 18.1\Starter` ,

  which contains a version of the `Deck` class of Labs 15.1 and 17.1.

The present version of `Deck` has no capabilities for saving and restore state, which you need to provide.

- Inspect the well-known types, which you have seen before:
  - `Suit`
  - `Rank`
  - `Card`
  - `Deck`
- Implement the Memento pattern on `Deck` such that after Shuffle() or Deal() the state of `Deck` can be restored
  - Make sure nobody outside of `Deck` can inspect the state inside the Memento object.
- Create a single unit test validating your implementation.

Lab 18.2: "Adding Redo Functionality using Memento" (⭐⭐)

This exercise improves on the Memento example of the presentation by adding Redo functionality to the solution already supporting Undo.

Note: The exercise is probably mostly for developers with a basic knowledge of Windows Presentation Foundation – or not afraid of figuring out how the existing command setup works in the supplied example code!

- Open the starter project in
  PathToCourseFiles`\Labs\18 - Memento\Lab 18.2\Starter` ,
  which contains the example from the presentation already supporting Undo functionality.

Your job is to add Redo functionality to the existing Undo functionality.

- Examine the XAML markup and add a "Redo" button.
- Construct the required addtional `ICommand` implementation in the view model and wire everything up accordingly.
- Implement Redo functionality using the newly added command and exising Memento pattern implementation.

Module 19: “Command”

Lab 19.1: ”Commands with Parameters” (⭐)

This exercise will illustrate a variation of the Command pattern, which extends Commands with parameters.

- Open the starter project in
  PathToCourseFiles`\Labs\ 19 - Command\Lab 19.1\Starter` ,
  which contains a class `LedLight` which has an interface allowing the client to set an intensity percentage between 0 and 100.

Your job is to implement the Command pattern which allows the commands to have an intensity parameter.

- Mimick the setup given in the examples of the presentation for this module and define classes – however with some variations!
  - Use the existing class `LedLight` as the Receiver.
    - Program.cs creates three instances of it to control through a `LoggingSwitch` defined below.
  - Create a suitable `SetCommand` class.
  - Create a `SetCommandFactory` implementing the existing `ISetCommandFactory` interface returning `SetCommand` instances.
    - <u>Note</u>: The factory needs to know about the three `LedLight` instances.
  - Implement a `LoggingSwitch` class which adds the command to be executed in a list before executing the command itself
    - <u>Note</u>: Since the commands are now parameterized, the `LoggingSwitch` cannot simply use a constant constructor-supplied command.
    - In this sense, the `LoggingSwitch` is merely a “log-and-execute” wrapper for commands generated via the command factory.

Module 20: “State”

Lab 20.1: ”Maintainability of State Implementations”

This exercise will add a new state in the State pattern setup from the module presentation.

- Open the starter project in

  PathToCourseFiles`\Labs\ 20 - State\Lab 20.1\Starter ,`

  where the project contains the well-known types that you have seen in the presentation:
  - `TimerSetup`
  - `ITimerSetupState`
  - `TimerSetupStateBase`
  - `NormalState`
  - `SetHoursState`
  - `SetMinutesState`
  - `CompletedState`

Your task is now to test the maintainability of the State pattern by allowing the user to also include seconds in the `CompletedState` being configured by means of the `TimerSetup` class.

- Add a new state `SetSecondsState` which allows the configuration of seconds.
- Modify the existing program accordingly to incorporate the new state

When you’re done, spend a few minutes considering the following questions

- Which classes were changed?
- Are there any kind of optimizations that you feel are compelling?
  - Consider which of those are in fact good ideas

Module 21: "Interpreter"

Lab 21.1: "A Console-based Drawing Interpreter" ()

This exercise will modify the classes of the Interpreter pattern from the module presentation.

- Open the starter project in
  PathToCourseFiles`\Labs\ 21 - Interpreter\Lab 21.1\Starter` ,
  where the project contains the expression structure that you have seen in the presentation:
  - `IDrawing`
  - `NextTo`
  - `Inside`
  - `IShape`
  - `Box`
  - `Ellipse`

Upon inspection of the existing project you will realize that these types all have had their `Interpret()` method removed. (You will fill these gaps later in this exercise.) Correspondingly, the `Context` class has now been changed to the following:

```
class Context
{
    public string Evaluation { get; set; }
}
```

We will start by adding a new construct to the existing grammar as follows:

<*drawing*> ::= <*shape*> **rotated** <*angle*>,

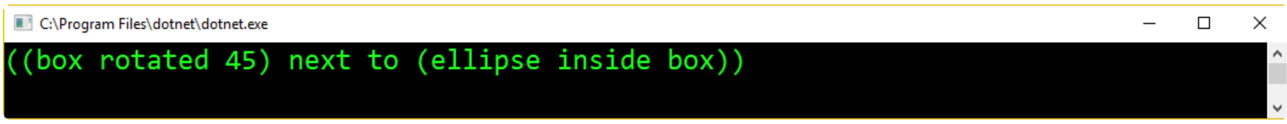
where <*angle*> is just an integer.

The remaining grammar rules are still present. In this exercise we will fortunately not be concerned with creating the graphics for the drawing – just the expression classes.

- Create a new class `Rotated`, defined corresponding to the new grammar rule above.
- In Program.cs define an `IDrawing` instance called `drawing` corresponding to the sentence " `box rotated 45 next to ellipse inside box` "
  - Leave the `Interpret()` method unimplemented for now.

Finally, we need to provide appropriate implementations for the `Interpret()` methods in each class.

- Implement all the `Interpret()` methods such that when you interpret the `IDrawing` instance called `drawing` created earlier, you obtain the following result:



Module 22: “Visitor”

Lab 22.1: ”Document Visitors” ( )

This exercise will implement the Visitor pattern for a document object structure already specified in the exercise project files.

- Open the starter project in
  PathToCourseFiles`\Labs\ 22 - Visitor\Lab 22.1\Starter` ,
  which contains a number of predefined document parts deriving from `DocumentElement`:
  - `RegularText`
  - `BoldText`
  - `Hyperlink`
  - `HeadingElement`.

The main document class in the project is specified as follows:

```
class Document : IEnumerable<DocumentElement>
{
    private readonly List<DocumentElement> _elements;

    public Document( params DocumentElement[] elements )
        : this(elements.AsEnumerable())
    {
    }

    public Document( IEnumerable<DocumentElement> elements )
    {
        _elements = elements.ToList();
    }

    ...
}
```

In this manner, an instance of Document can be created as specified below:

```
Document document = new Document(
    new HeadingElement( "Welcome to Document Fun", 1 ),
    new RegularText( "Here is some plain text." ),
    new BoldText( "Here is some bold text." ),
    new Hyperlink( "Useful information", "http://www.ubrugelig.dk" )
);
```

- Implement the Visitor pattern on `Document`.
- Create a class `HtmlVisitor` which produces the following output when visiting the `Document` instance specified above:

C:\WINDOWS\system32\cmd.exe

```
<h1>Welcome to Document Fun</h1>
Here is some plain text.
<b>Here is some bold text.</b>
<a href="http://www.ubrugelig.dk">Useful information</a>

Press any key to continue . . .
```

- To show the flexibility of the Visitor pattern already established, create a class `TextVisitor` which produces the following output when visiting the `Document` instance specified above:

C:\WINDOWS\system32\cmd.exe

```
WELCOME TO DOCUMENT FUN
Here is some plain text. --Here is some bold text.-- Useful information-->[http:
//www.ubrugelig.dk]

Press any key to continue . . .
```

Module 23: “Observer”

Lab 23.1: ”An Alternative Observer Pattern implementation in .NET 4.0” (★★★)

This exercise will provide an alternative to events when implementing the Observer pattern in C#.

- Open the starter project in
  PathToCourseFiles`\Labs\ 23 - Observer\Lab 23.1\Starter` ,
  which contains a fully functional event-based implementation developed during the module presentation.

In .NET 4.0 two new interfaces were – more or less unnoticed by the community – added to allow a different mechanism for implementing Observer. The new interfaces are defined as follows:

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

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

Your goal in this exercise is to refactor the existing solution to these new interfaces.

- Removed the event-specific aspects of the starter project and refactor the solution to use the two new interfaces for the existing `StockMarket` and `StockObserver` classes, respectively.
  - You will probably need to investigate the online documentation
  - Note: Don’t worry too much about race conditions and thread-safety issues.

Module 24: “Mediator”

Lab 24.1: ”Controlling Car Engines with Mediator” (⭐⭐)

This exercise will implement the Mediator pattern in a system of complicated interactions.

- Open the starter project in
  PathToCourseFiles`\Labs\ 24 - Mediator\Lab 24.1\Starter` ,
  which contains a number of classes and interfaces constituting parts of an interacting system.

The existing classes are composed of a futuristic car engine auto-driver system, which automatically operates the constituent engine components specified in the following classes:

- `Accelerator`
- `Brake`
- `GearBox`
- `Ignition`

Each of these derive from a common (but not yet completed) class `EngineComponent`.

The auto-driver system is subject to the following very complicated constraints and rules:

- *When the Ignition is off, the other components must also be off.*
- *When the Accelerator is on, the brakes must be disabled.*
  - *The Accelerator works by setting a desired target speed.*
    - *The auto-driver system should then increment the actual speed of the car until the designated target speed is reached*
    - *<u>Note</u>: For simplicity the Accelerator cannot decrease the actual speed, so these attempts can just be ignored.*
- *When Braking, the Accelerator must be disabled.*
  - *You can assume that when braking the car stops instantly (after all, it is a future car!), so the actual speed goes to 0*
    - *The gear should then be set to Neutral*
- *If the ignition is on, the Gear Box should be in the appropriate gear given the actual speed of the car*
  - *First Gear, if the actual speed < 25*
  - *Second Gear, if the actual speed < 50*
  - *Second Gear, if the actual speed < 70*
  - *Second Gear, if the actual speed < 90*
  - *Second Gear, if the actual speed < 110*
  - *Second Gear, if the actual speed < 130*
  - *Second Gear, if the actual speed >= 130*

Your task will be to complete the implementation of the auto-driver system by means of implementing the Mediator pattern for the pre-existing classes and interfaces of the Starter project.

In highlights your tasks are the following:

- Implement an appropriate `IEngineMediator` interface.
- Let the provided `EngineMediator` implement `IEngineMediator`.
  - Complete the `IEngineMediator` class by
    - providing “registration” methods for the components to call
    - providing “callback” (or “mediate”) methods for the components to call, e.g. `void OnIgnitionEnabledChanged( bool isOn )`
      - Hint: It makes very good sense to call the pre-existing `Display()` at the end of all of these methods.
- Store the components’ reference to the `IEngineMediator` in the `EngineComponent` class.
  - Update `Accelerator` to
    - store the mediator object
    - register with it, and
    - call it whenever state is changed
  - Update `Brake` to
    - store the mediator object
    - register with it, and
    - call it whenever state is changed
  - Update `GearBox` to
    - store the mediator object
    - register with it, and
    - call it whenever state is changed
  - Update `Ignition` to
    - store the mediator object
    - register with it, and
    - call it whenever state is changed.
- Set up the auto-driver system in Program.cs by performing the following actions:

```
ignition.Start();
accelerator.SetAccelerationTarget(50);
brake.Press();
ignition.Stop();
```

When completed, the first few lines of the result would probably resemble following:

C:\WINDOWS\system32\cmd.exe

```
Ignition turned on
Accelerator enabled
Ignition        Acc     Brake   Neutral    0 km/h
Gear box enabled
Ignition        Acc     Brake   Neutral    0 km/h
Ignition        Acc     Brake   Neutral    0 km/h
Speed targeted to 50 km/h
Brake disabled
Ignition        Acc     Brake   Neutral    0 km/h
Ignition        Acc     Brake   Neutral    5 km/h
Switching gear to First Gear
Ignition        Acc     Brake   First      5 km/h
Ignition        Acc     Brake   First     10 km/h
Ignition        Acc     Brake   First     15 km/h
Ignition        Acc     Brake   First     20 km/h
Ignition        Acc     Brake   First     25 km/h
Switching gear to Second Gear
Ignition        Acc     Brake   Second    25 km/h
Ignition        Acc     Brake   Second    30 km/h
```