### Creational Patterns

## Web Scraper

Pattern: **Singleton**

You are given a **WebPageRepository** class that stores web page URL addresses. Your task is to implement the singleton design p attern. Ensure that the class:

* Cannot be instantiated from outside
* Cannot be inherited
* Is thread-safe (see **Double-null check** pattern)

The code below should download the HTML content of all web addresses stored in the repository.

|  |
| --- |
| **Sample Source Code** |
| **var repository = WebPageRepository.Instance;**  **var downloader = new Downloader();**  **while (!repository.IsEmpty)**  **{**  **var url = repository.Remove();**  **downloader.Download(url, GenerateFileNameFromUrl(url));**  **}** |

## Tank Manufacturer

Pattern: **Factory Method**

Write a program with several factories for manufacturing **tanks**. The following factories should be supported:

* **German** factory - manufactures a **Tiger** tank, with speed **4.5** and damage **120**
* **Russian** factory - manufactures a **T 34** tank, with speed **3.3** and damage **75**
* **American** factory - manufactures a **M1 Abrams** tank, with speed **5.4** and damage **120**

Steps:

1. Define an abstract factory with a **CreateTank()** method
2. Define concrete factories with concrete implementations of that method (e.g. **GermanTankFactory** creates a **Tiger** tank)
3. Use the concrete factories interchangeably in the code

|  |  |
| --- | --- |
| **Sample Source Code** | **Sample Output** |
| **TankFactory tankFactory = new GermanTankFactory();**  **// tankFactory = new RussianTankFactory();**  **// tankFactory = new AmericanTankFactory();**    **var tank = tankFactory.CreateTank();**  **Console.WriteLine(tank);** | **-Tank**  **...Model: Tiger**  **...Speed: 4.50**  **...Damage: 120** |

### Structural Patterns

## DOM Builder

Pattern: **Composite**

Write a program that allows building complex **tree**-like structures with DOM elements.

Steps:

1. Create an **Element** class that holds a **type** and a **collection of its children**
2. Define **Add()** and **Remove()** methods for adding/removing children - add validation for null
3. Define a **Display()** method that renders the current element and all its children elements
4. Create a constructor that accepts **type** + an arbitrary number of **elements**

|  |  |
| --- | --- |
| **Sample Source Code** | **Sample Output** |
| **Element html =**  **new Element("html",**  **new Element("head"),**  **new Element("body",**  **new Element("section",**  **new Element("h2"),**  **new Element("p"),**  **new Element("span")),**  **new Element("footer")));**  **File.WriteAllText("index.html", html.Display());** | **<html>**  **<head>**  **</head>**  **<body>**  **<section>**  **<h2>**  **</h2>**  **<p>**  **</p>**  **<span>**  **</span>**  **</section>**  **<footer>**  **</footer>**  **</body>**  **</html>** |

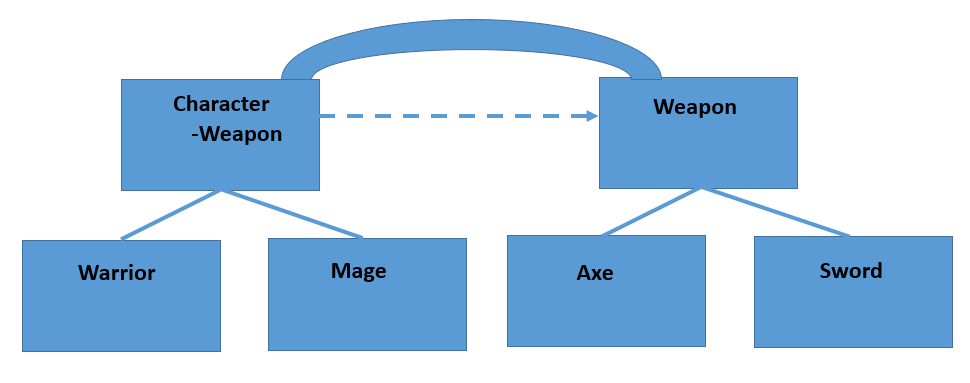
## RPG

Pattern: **Bridge**

Write a simple RPG Game where there are **characters** and **weapons**. The game is simple - each character can have **one weapon** at a time. Implement a **Mage** and a **Warrior** as characters. Implement a **Sword** and an **Axe** as weapons. Any character should be able to use any weapon.

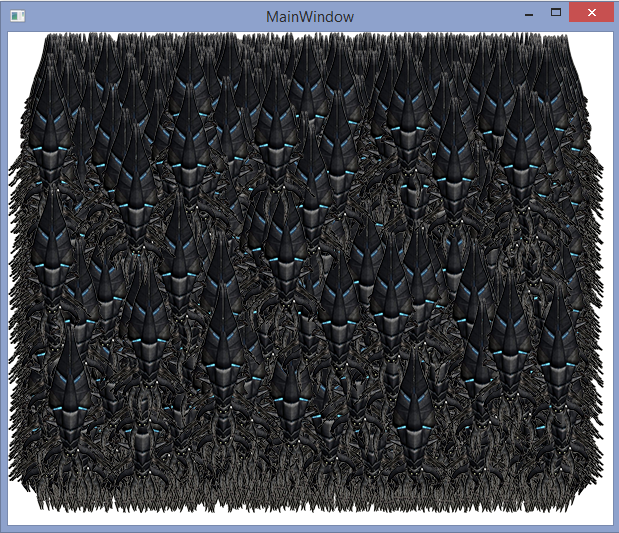
Steps:

1. Create an abstract class **Character**
2. Create an abstract class **Weapon**
3. A **character** should receive an abstract **Weapon** (through dependency injection)



## Reaper Invasion

Pattern: **Flyweight**

You are given a working **WPF application** that renders **10000 Reaper images** on the screen. However, its performance is quite **poor** (takes at least 10 seconds to draw everything).

Your task is to study the application, **pinpoint the bottleneck** and apply the **Flyweight Design Pattern** in order to optimize its performance.

**Hints**:

* Use a performance testing tool such as [**dotTrace**](https://www.jetbrains.com/profiler/)to pinpoint the bottleneck method(s)
* Look for places where a **heavy operation** is performed **too often unnecessarily**
* **Save** (cache) the **operation result** and reuse it, instead of repeating It each time

### Behavioral Patterns

## C# Compiler

Pattern: **Strategy**

You are given a **simple API** for **compiling C# code** **strings** at runtime. The process is as follows:

* Preprocess - prepares the code for compilation by performing **validations**
* Compile - compiles the code to **Intermediate Language (IL)**
* Execute - begins program execution from **IL** to **machine instructions**

Your task is to open the **Preprocess** step for extension by **leaving the user to decide** what sort of **validation** should be made. **Tip**: Use **Dependency Injection**.

Steps:

1. Edit the **CSharpCompiler** constructor so that it receives a **strategy** from outside (through dependency injection). The strategy will be used during the preprocess step.
2. Create an interface that defines a strategy - e.g. **ICodeValidationStrategy** with a **Validate()** method
3. Create concrete validation strategies:
   * Validates if the code is at least 20 characters long
   * Validates if no "**using System.Net**" reference is contained in the code string

**CompilationException** with a proper message should be thrown if validation fails.

1. The validation strategy should be **optional**. Ensure **NullReferenceException** is never thrown.

|  |
| --- |
| **Sample Source Code** |
| **ICodeSyntaxValidationStrategy codeValidationStrategy = new CodeLengthValidator();**  **// codeValidationStrategy = new SystemNetValidator();**  **var compiler = new CSharpCompiler(codeValidationStrategy);**  **compiler.Compile(code);**  **compiler.Execute(EntryClassName);** |

## Skyrim

Pattern: **Observer**

You're in Skyrim - there are **dragons** and **warriors**. Warriors fight dragons and whenever a dragon dies, it drops an item (such as a sword) to all warriors.

You are given ready classes for **Dragon** and **Warrior**. Apply the Observer pattern so that when a dragon dies (its **health** falls below **0**) all warriors **attached** to it receive a Sword item in their inventories.

The **dragon** is the **subject** (i.e. the "event" the warriors will be listening to and his death will trigger that event). As such, it should keep a **list of all observers** of that event.

Steps:

1. Create an interface **IDragonDeathObserver** with method **Update(Weapon weapon)**
2. The warriors should implement that interface and add the received weapon whenever they are updated of a dragon's death
3. The dragon itself should hold a list of all warriors and offer methods **Attach()** and **Detach()** for adding/removing listeners.
4. Add a **Notify()** method to the dragon. Whenever his health falls below **0**, the method should be called and all attached observers should be updated with the dropped item.

|  |
| --- |
| **Sample Source Code** |
| **foreach (var warrior in warriors)**  **{**  **dragon.Attach(warrior);**  **}**  **// Nothing happens**  **dragon.HealthPoints -= 20;**  **// Nothing happens**  **dragon.HealthPoints -= 10;**  **// Dragon dies**  **dragon.HealthPoints -= 90;** |

\* Implement the Observer pattern using **events**.

## Customer Service

Pattern: **Visitor**

You are given a **repository** of customers. **Customers** have **name**, **discount** and keep all the **purchases** they have made. When accessing customers from the repository, you might want to make changes to them (e.g. raise the discount of all premium customers).

Apply the Visitor pattern so you can make changes to the customers on the fly. Create visitors for the following operations:

* Raise discount of all **premium customers** by **5%**
* Give out free "**SteamOp**" purchases to **all customers**

Steps:

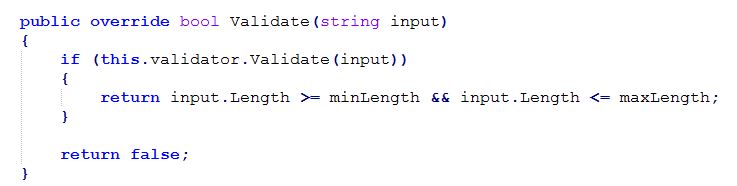
1. Create an interface **ICustomerVisitor** with a **Visit(Customer customer)** method.
2. Add an **Accept()** method in the **Customer** class that accepts a visitor and tells him to visit the current instance.
3. Create concrete visitors for the above described operations:
   * **FreePurchaseVisitor** - adds a free purchase of an item to the visited customer
   * **DiscountRaiseVisitor** - adds 5% discount to visited customer
4. Use the **DiscountRaiseVisitor** to increase all premium customers' discount
5. Use the **FreePurchaseVisitor** to give all customers' a free SteamOp

## Input Validators

Pattern: **Decorator**

You need to create a simple application that validates input according to **different** **criteria**. All the criteria should be **combinable**. Instead of creating different classes for all possible combinations, you should consider implementing the **decorator** pattern.

You are given a simple **validator** class that only checks if the input is not **null.** You are also given a base decorator that takes as a parameter an IValidator. You can add any type of validation in the decorator but you should call the base validation first.

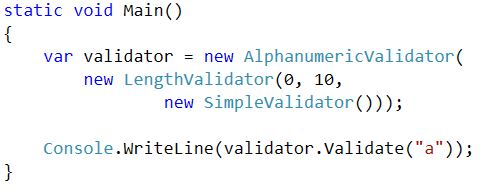


Consider implementing the following decorators:

* **Alphanumeric** -> the input string should contain only letters and digits
* **Length** -> the length of the input string should be between some minimum and maximum length
* **PasswordSecurity** -> the string should contain at least one non-digit and non-letter character

You can think of other type of validation decorators.

Ideally, you should be able to combine them in the following way:



## \* Unit Testing Framework

Pattern: **Decorator**

You are given the source code of a **unit testing framework** - mUnit. Just like any C# testing framework, it supports attributes for test methods:

* **[TestClass]** - applied to classes which contain test methods
* **[Test]** - applied to methods that should execute without throwing any exceptions
* **[ShouldThrow]** - applied to methods which are expected to throw an exception

The framework currently supports running and executing test methods with the **[Test]** and **[ShouldThrow]** attributes.

There is a third attrbiute, **[TestCase(*arg*)]**, but it is currently not supported by the framework. Unlike **[Test]** and **[ShouldThrow]**, this attribute can be applied many times on a single test method, e.g.:

|  |
| --- |
| **Test + 3 Test cases** |
| **[Test]**  **[TestCase(300.0d)]**  **[TestCase(400.0d)]**  **[TestCase(500.0d)]**  **public void TestDeposit(double sum)**  **{**  **var account = new Account();**  **account.Deposit(sum);**    **Assert.AreEqual(sum, account.Balance);**  **}** |

|  |
| --- |
| **ShouldThrow + 2 Test cases** |
| **[Test]**  **[ShouldThrow(typeof(ArgumentException))]**  **[TestCase(-300.0d)]**  **[TestCase(-400.0d)]**  **public void DepositNegativeSum\_EmptyAccount\_ShouldThrowException(double sum)**  **{**  **var account = new Account();**  **account.Deposit(sum);**  **}** |

The test method should then be run as many times as there are **[TestCase]** attrbiutes. The attribute's argument (e.g. **-300.0d**) should be passed to the test method.

The attribute is there, but the engine does not currently recognize it. Implement the necessary logic, so the engine may execute both normal tests (**[Test]**) and **[ShouldThrow]** tests with many cases.

**Hints**:

* Take a look at the **TestRunner** classes. What is their role in running tests? How does the engine create and use them?
* Now create a test runner for the **[TestCase]** attribute. The test runner should support running multiple tests in combination with either **[Test]** or **[ShouldThrow]**.
* Using the Decorator pattern, your new test runner should ***decorate*** the existing two test runners and reuse their functionality.