Assignment
Build a console application in .NET

Dungeon Master

Build a console application in .NET. Follow the guidelines given below, feel free to expand on the functionality. It must meet the minimum requirements prescribed.

1) Set up the development environment

Make sure you have installed at least the following tools:

- Visual Studio with .NET 6

2) Optional: Class interaction diagrams

You can draw out the planning of the various classes and their interactions to help visualize the application and its functionality.

3) Use plain C# to create a console application with the following minimum requirements (See Appendix A-C for details):

a) Various *hero classes* having attributes which increase at different rates as the character gains levels.
b) *Equipment*, such as armor and weapons, that characters can equip. The equipped items will alter the power of the hero, causing it to deal more damage and be able to survive longer. Certain heroes can equip certain item types.
c) *Custom exceptions*. There are two custom exceptions you are required to write, as detailed in Appendix B.
d) *Testing* of the main functionality.
e) CI pipeline to show that all tests are passed.

4) Submit

a) Create a GitLab repository containing all your code.
b) This repository should contain a well formatted README and appropriate commit messages and branches.
c) You can include the class diagram in this repository if you have made one.
d) The repository must be private, add your lecturer as a Maintainer.
e) Submit only the link to your GitLab repository (not the “clone with SSH”).

Appendix A: Hero classes

1) Introduction and overview

The Wizard shown here is an example of a hero select screen in an Action RPG called Diablo 2.

NOTE: This section is a big picture overview, the details for each aspect are covered in separate sections. This is just to have a checklist for functionality in an easy to access format.

In the game there are currently four classes that a hero can be:

- Wizard
- Archer
- Swashbuckler
- Barbarian

Each hero has the following shared fields:

- Name
- Level - all heroes start at level 1
- LevelAttribtues - total from all levels
- Equipment - holds currently equipped items
- ValidWeaponTypes – a list of weapon types a hero can equip based on their subclass
- ValidArmorTypes - a list of armor types a hero can equip based on their subclass

Heroes have the following public facing methods:

- Constructor – each hero is created by passing just a **name**.
- LevelUp – increases the level of a character by 1 and increases their LevelAttributes
- Equip – two variants, for equipping armor and weapons
- Damage – damage is calculated on the fly and not stored
- TotalAttributes – calculated on the fly and not stored
- Display – details of Hero to be displayed

2) Hero attributes

The attribute system found in this assignment is based on the traditional Three-Stat System leaning towards a **Diablo 3** implementation. Looking at **hero attributes:**

- **Strength** – determines the physical strength of the character.
- **Dexterity** – determines the characters ability to attack with speed and nimbleness.
- **Intelligence** – determines the characters affinity with magic.

You are to make a class to encapsulate these attributes, it should be called **HeroAttribute.** This class should have methods to add two instances together and return the sum OR increase the one instance by a specified amount. Which to use is up to you on what best suits your solution.

NOTE: This class is used as the datatype for both LevelAttributes.

3) Levelling attributes

There should be a base abstract **Hero** class to encapsulate all the shared functionality (fields and methods). Any methods which have a default implementation can be defined in this abstract class to be overridden in the child classes. If there is no default implementation for a method, make it abstract to be implemented in a child class.

NOTE: When testing the functionality, you need to interact with the abstract Hero class, and not the subclasses. This is to satisfy the Liskov Substitution Principle.

Each sub class will start at different attributes and increase at different rates when levelling up. These are detailed below:

3.1) Wizard attribute gain

A Wizard begins at level 1 with:

	Strength	Dexterity	Intelligence
	1	1	8

Every time a Wizard levels up, they gain:

	Strength	Dexterity	Intelligence
	1	1	5

3.2) Archer attribute gain

An Archer begins at level 1 with:

	Strength	Dexterity	Intelligence
	1	7	1

Every time an Archer levels up, they gain:

	Strength	Dexterity	Intelligence
	1	5	1

3.3) Swashbuckler attribute gain

A Swashbuckler begins at level 1 with:

	Strength	Dexterity	Intelligence
	2	6	1

Every time a Swashbuckler levels up, they gain:

	Strength	Dexterity	Intelligence
	1	4	1

3.4) Barbarian attribute gain

A Barbarian begins at level 1 with:

	Strength	Dexterity	Intelligence
	5	2	1

Every time a Barbarian levels up, they gain:

	Strength	Dexterity	Intelligence
	3	2	1

Appendix B: Items and equipment

1) Introduction and overview

Heroes can **equip** various items. The two types of items are: **Weapon** and **Armor.** There should be a parent **Item** abstract class which is inherited by the above-mentioned types. All items share the following fields:

- Name
- RequiredLevel
- Slot

If a hero is below the **RequiredLevel**, they cannot equip the item. Each item is equipped in a specific **Slot.**

Slot should be represented as an enumerator and have the following values:

- Weapon
- Head
- Body
- Legs

2) Weapons

There are several types of weapons which exist:

- Hatchets
- Bows
- Daggers
- Maces
- Staffs
- Swords
- Wands

You should encapsulate these types in a **WeaponType** enumerator and compose that into the weapon class. In addition to a weapon type, weapons deal damage. This is represented as a **WeaponDamage** field.

NOTE: When a weapon is created, it is automatically given the Slot of Weapon.

3) Armor

There are several types of armor that exist:

- Cloth
- Leather
- Mail
- Plate

You should encapsulate these types in a **ArmorType** enumerator and compose that into the armor class. In addition to an armor type, armor has attributes which provide bonuses to a hero's attributes when equipped. This field is of type HeroAttribute and should be called **ArmorAttribute.**

4) Equipment

Recall, Heroes have equipment, which is a collection of items. The data structure for this should meet the requirement of: **<Slot, Item>** as a key value pair. When a new Hero is created, the equipment is initialized to have each slot as an entry with null values to represent empty slots. This should result in 4 entries with keys for each Slot and null values.

4.1) Equipping weapons

Certain hero classes can equip certain weapon types:

- **Wizards** – Staff, Wand
- **Archers** – Bow
- **Swashbucklers** – Dagger, Sword
- **Barbarians** – Hatchet, Mace, Sword

If a hero tries to equip a weapon they should not be able to, either by it being the *wrong type* or being *too high of a level requirement*, then a custom **InvalidWeaponException** should be thrown with an appropriate message.

4.2)Equipping armor

Certain hero classes can equip certain armor types:

- **Wizards** – Cloth
- **Archers** – Leather, Mail
- **Swashbucklers** – Leather, Mail
- **Barbarians** – Mail, Plate

If a character tries to equip armor they should not be able to, either by it being the *wrong type* or being *too high of a level requirement*, then a custom **InvalidArmorException** should be thrown with an appropriate message.

NOTE: When equipping a new item, don't worry about the old item, just replace it. It does not need to be stored anywhere.

5) Calculations and display

Recall, there are several methods dedicated to calculations for a Hero. These are: **Total Attributes** and **Damage**. A hero should also be able to **Display** themselves.

5.1) Calculating total attributes

A hero's total attributes is simply the sum of their levelling attributes and all the armor attributes from their equipment. This is represented by the calculation:

- Total = **LevelAttributes** + (Sum of **ArmorAttribute** for all Armor in Equipment)

NOTE: The sum is not the three fields added together, its an object of type HeroAttribute with its strength value being the total strength, its dexterity value being the total dexterity, and its intelligence being the total intelligence.

This requires the equipment to be looped through with all the non-weapon slots being used in the calculation.

5.2) Calculating a hero's damage

The damage a hero deals is based in their currently equipped weapon and increased by their attributes. Each hero class has a **damaging attribute** which contributes to their total damage:

- **Barbarian** – damage increased by **total strength**
- **Wizard** – damage increased by **total intelligence**
- **Archer** – damage increased by **total dexterity**
- **Swashbuckler** – damage increased by **total dexterity**

NOTE: Damaging attribute does not need to be stored, it can just be calculated from TotalAttributes

A hero’s total damage is equal to the equipped weapon damage increased by 1% for every point in their damaging attribute. It can be represented by the formula:

- Hero damage = **WeaponDamage** * (1 + **DamagingAttribute**/100)

NOTE: If a Hero has no weapon equipped, take their WeaponDamage to be 1.

5.3) Hero display

A hero has a public method which returns a string to display their state. This needs to show their:

- Name
- Class
- Level
- Total strength
- Total dexterity
- Total intelligence
- Damage

A good option for this is to use a string builder.

Appendix C: Unit testing

1) Introduction and overview

Unit testing is to verify the behavior of Heroes. This is how the assignment is "run". No code needs to be in a main method, only a test suite needs to be run. The tests should cover all the public facing methods of the Heroes as well as any functionality used in those processes. An overview of the tests to write are provided here:

- When a Hero is created, it needs to have the correct name, level, and attributes.
- When a Heroes level is increased, it needs to increment by the correct amount and result in the correct attributes.
    - Creation and leveling tests need to be written for each sub class
    - A test to see if HeroAttribute is being added/increased correctly should also be written
- When Weapon is created, it needs to have the correct name, required level, slot, weapon type, and damage
- When Armor is created, it needs to have the correct name, required level, slot, armor type, and armor attributes
- A Hero should be able to equip a Weapon, the appropriate exceptions should be thrown if invalid (level requirement and type)
- A Hero should be able to equip Armor, the appropriate exceptions should be thrown if invalid (level requirement and type)
- Total attributes should be calculated correctly
    - With no equipment
    - With one piece of armor
    - With two pieces of armor
    - With a replaced piece of armor (equip armor, then equip new armor in the same slot)
- Hero damage should be calculated properly
    - No weapon equipped
    - Weapon equipped
    - Replaced weapon equipped (equip a weapon then equip a new weapon)
    - Weapon and armor equipped
- Heroes should display their state correctly

It is important to have as much coverage as feasible with tests, you can run tests with coverage to see what % of your code is tested. 100% code coverage is not something to aim for, as it often is not feasible. It is also very important to remember best practices for writing tests, these include:

- Appropriate, clear, test names
- No magic variables
- Verbose test bodies
- AAA structure
- One assert per test
- Minimally passing tests
- No global instances for classes involved in tests

Consult the notes to refresh your memory if needed. For testing data, aside from the provided attribute values, the other values are up to you. The sub section below provides an example of a single test, keep in mind, you are not required to repeat this exact test, but use it to guide your process for the other tests.

1.2) Example single test walkthrough

We have a **Weapon** with the following state:

- Name = "Common Hatchet"

- RequiredLevel = 1
- Slot = Slot.Weapon
- WeaponType = WeaponType.Hatchet
- WeaponDamage = 2

We have **Armor** with the following state:

- Name = “Common Plate Chest”
- RequiredLevel = 1
- Slot = Slot.Body
- ArmorType = ArmorType.Plate
- ArmorAttribute = str:1, dex: 0, int: 0

We use a **Barbarian** as our Hero.

1) Calculate damage if no weapon is equipped.
    - Take Barbarian at level 1
    - Damage = 1*(1 + (5 / 100))
2) Calculate damage with valid weapon equipped.
    - Take Barbarian level 1.
    - Equip Hatchet.
    - Damage = 2*(1 + (5 / 100))
3) Calculate damage with valid weapon and armor equipped.
    - Take Barbarian level 1.
    - Equip Hatchet.
    - Equip plate body armor.
    - Damage = 2*(1 + ((5+1) / 100))

Appendix D: Extensions and alternative approaches

1) Extending base functionality

NOTE: This should only be attempted once you are complete with the assignment. You should implement these changes on a separate branch to not conflict with your main/master branch. You should always have a clean working version of your project according to the specifications to submit in case something goes wrong with your extensions.

If you wish to extend the functionality of the assignment the following functionalities will be the simplest to extend:

- Creating a new class with their own starting attributes
- Creating an interactive console-based gameplay loop with a **GameController** with other classes/methods to help**.**
  - This can prompt users to pick a class and then have a game loop to battle monsters, gain levels, and equip new items.
  - Random generation can be used here to great effect, provided the functionality is tested beforehand.
- Including an **Inventory** to hold items the Hero does not have equipped.
- Adding **Abilities** into the game, these can be spells/attacks a hero can perform.
  - You can use Damage/Attributes to scale the damage of the spells.
    - E.g. Fireball – Damage + 150% total Intelligence
    - E.g. Heavy Smash – Damage + 200% strength
  - Or the abilities can do base damage that is increased by the heroes damage
    - E.g. Arcane Blast – 15 * (1 + Damage/100)

2) Alternative approach

NOTE: This should only be attempted after you have completed the assignment on a separate branch, or you are very confident in OOP and OOD. The assignment is graded along the same lines, meaning, the same functionality needs to be present, it will just be implemented differently. Undertaking this approach from the beginning is at your own risk.

If you are very familiar with OOP and want to practice design, then you can approach the assignment in the following way:

- Have just a single Hero class
  - Need to store HeroType for display purposes now
- Implement the **strategy pattern** for the attributes
- Implement a **factory** or **builder** pattern for Hero creation
  - Can do the same for Weapon and Armor
- Move Equipment out to an **EquipmentManger** interface composed into the Hero
  - This manages EquippedWeapons and EquippedArmor
  - Should have public methods for equipping weapons and armor, as well as getting the total armor attributes
- Hero should not be composed with any concrete classes, should all be abstractions.
  - Look to decouple, using Dependency Inversion Principle as a guide
  - This ties into the creational design patterns mentioned above