Assignment 5
Build a console application in C#

RPG Characters

Build a console application in C#. 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 2019 with .NET 5.
- Unit testing project added to your solution.

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. You could even submit a generated class diagram from Visual Studio.

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

a) Various *character 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 character, causing it to deal more damage and be able to survive longer. Certain characters can equip certain item types.
c) *Summary (///) tags* for each method you write, explaining what the method does, any exceptions it can throw, and what data it returns (if applicable). You do not need to write summary tags for overloaded methods.
d) *Custom exceptions*. There are two custom exceptions you are required to write, as detailed in Appendix B.
e) *Full test coverage* of the functionality. Some testing data is provided, it can be used to complete the assignment in a test-driven development manner.

4) Submit

a) Create a GitLab repository containing all your code.
b) You can include the generated class diagram in this repository if you have made one.
c) The repository must be either public, or I am added as a Maintainer (@NicholasLennox).
d) Submit only the link to your GitLab repository (not the “clone with SSH”).

Appendix A: Character classes and attributes

1) Introduction and overview

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

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

- Mage
- Ranger
- Rogue
- Warrior

Characters in the game have several types of attributes, which represent different aspects of the character. They are divided into two groups:

- Primary attributes
- Secondary attributes

Primary attributes are what determine the characters power, as the character levels up their primary attributes increase. Items they equip also can increase these primary attributes (this is detailed in **Appendix B**). Secondary attributes affect the characters' ability to survive, they are calculated from primary attributes. Certain character classes are more durable against certain types of damage.

NOTE: Characters are not required to take damage. The secondary attributes are there to simply add value to attributes that do not affect the power of the character. For the requirements, it will be a simple calculate and display.

2) Attributes

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

- **Strength** – determines the physical strength of the character.
  - Each point of strength increases a **Warriors** damage by **1%.**
- **Dexterity** – determines the characters ability to attack with speed and nimbleness.
  - Each point of dexterity increases a **Rangers** and **Rogues** damage by **1%**.
- **Intelligence** – determines the characters affinity with magic.
  - Each point of intelligence increases a **Mages** damage by **1%**.
- **Vitality** – determines how much health a character has overall.
  - Each point of vitality increases a **character's** health by 10.

As for **secondary attributes.** As mentioned above, they are calculated based on primary attributes:

- **Health** – determines how much overall damage a character can take before they die.
  - Each point of vitality increases health by 10.
  - **Health = (Total Vitality) *10**
- **Armor Rating –** determines the physical resistance of the character.
  - Each point of Strength or Dexterity adds one armor rating.
  - **Armor Rating = Total Strength + Total Dexterity**
- **Elemental Resistance** – determines how resistance the character is to magic damage.
  - Each point of Intelligence adds one elemental resistance.
  - **Elemental Resistance = Total Intelligence**

Finally, a character can deal **Damage**. The amount of damage depends on the weapon equipped, the character class, and the amount of primary attribute the character has. This calculation is detailed in **Appendix B** when items are discussed.

It is recommended to make custom types called **PrimaryAttribute** and **SecondaryAttribute** to encapsulate the various attributes.

It is also recommended to **overload the + operator** for these custom types to simplify increasing attributes. This is detailed in a demo video.

3) Character classes and levelling

It is recommended to have a base abstract **Character/Hero** class that can be used to encapsulate the functionality. Functionality that is different between character classes can be defined as an **abstract method** in the base class, to be implemented in the inheriting class. This is to limit repeated functionality – DRY.

All characters have the following properties:

- Name
- Level
- Base Primary attributes
- Total Primary attributes
- Secondary attributes

NOTE: There are more properties a character has, those are relating to items and equipment. Those properties will be detailed in Appendix B.

When a character is created, they are provided a name. Every character begins at **level 1.** There should be a way to increase the level of a character.

The separation of base and total primary attributes is for a very simple reason that will help avoid a great deal of trouble. It is done this way so the characters primary attributes from itself and its levels are separate from the bonus primary attribute from equipment. This is detailed more in **Appendix B**, but all that needs to be knows now is that it simplifies switching equipment.

3.1) Mage attribute gain

A Mage begins at level 1 with:

Vitality	Strength	Dexterity	Intelligence
5	1	1	8

Every time a Mage levels up, they gain:

Vitality	Strength	Dexterity	Intelligence
3	1	1	5

RECALL: Mages deal increased damage for every point of Intelligence.

3.2) Ranger attribute gain

A Ranger begins at level 1 with:

Vitality	Strength	Dexterity	Intelligence
8	1	7	1

Every time a Ranger levels up, they gain:

Vitality	Strength	Dexterity	Intelligence
2	1	5	1

RECALL: Rangers deal increased damage for every point of Dexterity.

3.3) Rogue attribute gain

A Rogue begins at level 1 with:

Vitality	Strength	Dexterity	Intelligence
8	2	6	1

Every time a Rogue levels up, they gain:

Vitality	Strength	Dexterity	Intelligence
3	1	4	1

RECALL: Rogues deal increased damage for every point of Dexterity.

3.4) Warrior attribute gain

A Warrior begins at level 1 with:

Vitality	Strength	Dexterity	Intelligence
10	5	2	1

Every time a Warrior levels up, they gain:

Vitality	Strength	Dexterity	Intelligence
5	3	2	1

RECALL: Warriors deal increased damage for every point of Strength.

Appendix B: Items and equipment

1) Introduction and overview

The game has items which exist. These items can be equipped by characters to increase their power, this is called equipping an item. The currently equipped items are called the characters equipment.

There are two types of items which exist:

- Weapons
- Armor

Weapons determine the damage a character can deal, which is then enhanced by the characters attributes. Armor adds to the attributes of a character, normally for defense.

Certain characters can only equip specific types of weapons and armor, custom exceptions are used to give proper feedback on this.

It is recommended to have a base abstract **Item** with **Weapon** and **Armor** can inherit. These simplifies equipment management greatly.

All items have:

- Name
- Required level to equip the item.
- Slot in which the item is equipped.

2) Weapons

There are several types of weapons which exist:

- Axes
- Bows
- Daggers
- Hammers
- Staffs
- Swords
- Wands

It is recommended to store these types as a property in the Weapon. An enumerator could be useful here to compose your weapon with its type.

Weapons have a **base damage**, and how many **attacks per second** can be performed with the weapon. The weapons damage per second (DPS) is calculated by multiplying these together.

- **DPS = Damage * Attack speed**

NOTE: When calculating the DPS of a character, the weapon DPS is used, not the base damage.

As mentioned before, certain character classes can equip certain weapon types. This is shown below:

- **Mages** – Staff, Wand
- **Rangers** – Bow
- **Rogues** – Dagger, Sword
- **Warriors** – Axe, Hammer, Sword

If a character 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.

It is recommended to think about how this Weapon check is implemented. Try out some implementations and decide on which is best (hint, think about some OO Design here, maybe it could be in the inherited classes with a base abstract method to be more extendable). You can use unit testing to see if your refactors break the functionality.

3) Armor

There are several types of Armor that exist:

- Cloth
- Leather
- Mail
- Plate

As with Weapons, it is recommended to store this type as property.

Armor has attributes that add to the character's power. These attributes are the same as the primary attributes and this means the **PrimaryAttribute** custom type can be reused.

Like Weapons, Armor is restricted to certain character classes. This is shown below:

- **Mages** – Cloth
- **Rangers** – Leather, Mail
- **Rogues** – Leather, Mail
- **Warriors** – 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.

Like with weapons, it is recommended to try different implementations of the Armor check and settle on one that is the best designed, according to you.

4) Equipment

A character can equip any item. An equipped item is stored in the character's equipment and is used to increase the characters power.

Items can be equipped in one of several slots:

- Head
- Body
- Legs
- Weapon

Armor can be equipped in any non-weapon slot, and weapons can only be equipped in a weapon slot. You can create Slot as an enumerator.

NOTE: You do not have to add a check to see if a weapon has its slot as Weapon, you can assume data will be created properly. Testing data is provided to you to use.

It is recommended to store the equipment as a **Dictionary<Slot, Item>**. This is to ensure you only ever have one item in each slot. You will just replace the Item that corresponds to the Slot.

NOTE: You do not need to cater for unequipped items, they just are removed. Ideally, they would go into an inventory, but that is not part of the base requirements.

It is also recommended to have two **Equip** methods on a character, one with a **Weapon** as a parameter, and the other with **Armor**. This way there is one public facing Equip method that takes the different types of items.

4.1) Total attributes and calculations

As mentioned before, every character has a base and total primary attribute. When the total is needed you should look at what armor is equipped and add all the primary attributes present in those items to the base.

- **Total attribute = base + attributes from all equipped armor**

This can then be used to determine the character's DPS

- **Character DPS = Weapon DPS * (1 + TotalPrimaryAttribute/100)**

Recall, when we speak about primary attribute here, it is the one that increases the damage for the class (Strength for Warriors, etc..). Hint: you can see how this will behave differently based on the character class. Consider using polymorphism here to aid you.

NOTE: If a character has no weapon equipped, take their DPS to equal 1.

5) Character stats display

All characters need a way to display their statistics to a character sheet. For this example, a simple **string** generated by using a **StringBuilder** is a good solution. This sheet should show:

- Character name
- Character level
- Strength
- Dexterity
- Intelligence
- Health
- Armor Rating
- Elemental Resistance
- DPS

The attribute statistics is the total (base + gear bonus).

NOTE: These values change as the character levels up or equips new items.

Appendix C: Unit testing and test coverage

1) Introduction and overview

Unit testing is an integral part of any well-designed system. It allows us to know for certain what functionality is working as intended. When we speak of test coverage, it means that every public facing method is tested to ensure it outputs the expected values or has the desired effect. This does not mean we need to test for methods used inside the classes (the private methods) as those are the refactoring of the public facing methods.

This appendix will detail what functionality needs to be tested and will provide some data to use. It will not be written tests, but simply inputs and expected outputs. It is recommended to make two test classes, one for Character tests and one for Item tests.

When performing unit tests, there are a few things to keep in mind that have been previously discussed:

- One assert per [Fact]
    - If you have multiple instances of data, use [Theory], but only assert once in the test body.
- Remember AAA – arrange act assert.
- Be as explicit as possible, every parameter or output is stored in a variable.
    - Do not invoke the method in the assert, invoke it before and save the return.
- Naming is important.
    - MethodYouAreTesting_ConditionsItsBeingTestedUnder_ExpectedBehaviour().

If you need any refreshers on Unit testing best practices, refer to the Microsoft Official Documentation on Unit testing best practices.

2) Character attribute and level tests

These tests could be done in a separate test class to the items one. This makes our codebase easier to navigate. These tests should cover the following behavior:

1) A character is level 1 when created.
2) When a character gains a level, it should be level 2.
3) If you try to gain 0 or less levels, an **ArgumentException** should be thrown.
    - Use 0 and -1 as InlineData for a theory.
4) Each character class is created with the proper default attributes.
    - Use level 1 stats for each character as expected.
    - This results in four test methods.
5) Each character class has their attributes increased when leveling up.
    - Create each class once, level them up once.
    - Use the base attributes, plus one instance of the level up as the expected.
    - E.g. Warrior -> levelup(1) -> (Vitality = 15, Strength = 8, Dexterity = 4, Intelligence = 2) expected.
    - This results in four test methods.
6) Secondary stats are calculated from levelled up character.
    - Create a warrior, level up once.
    - Expected secondaries: Health = 150, ArmorRating = 12, ElementalResistance = 2

This should result in at least 12 test methods. 11 Facts and 1 Theory.

3) Items and equipment tests

The items and equipment tests use a **Warrior** class for testing, an axe for an example **weapon** and plate body armor as an example **armor**. We also use a bow for a wrong weapon type, and cloth head armor for wrong armor type.

Keep in mind, because this functionality is shared between all classes and items, just testing these minimal examples covers all the code. You do not need to create one of every item type and character. This is because we are testing behaviors not implementations. We are answering the question “Can a character equip a weapon and armor as expected?”. If this is unclear to you, go back to the provided resources in the lectures and read about “writing minimally passing tests”.

Another thing to keep in mind, the test data will expose one possible solutions data structure for these items. You do not need to follow it exactly for naming and composition, but the main aspects need to be captured to fulfill the functional requirements.

Example weapon used for testing:

```
Weapon testAxe = new Weapon()
{
    ItemName = "Common axe",
    ItemLevel = 1,
    ItemSlot = Slot.SLOT_WEAPON,
    WeaponType = WeaponType.WEAPON_AXE,
    WeaponAttributes = new WeaponAttributes() { Damage = 7, AttackSpeed = 1.1 }
};
```

Example plate body armor used for testing:

```
Armour testPlateBody = new Armor()
{
    ItemName = "Common plate body armor",
    ItemLevel = 1,
    ItemSlot = Slot.SLOT_BODY,
    ArmourType = ArmourType.ARMOUR_PLATE,
    Attributes = new PrimaryAttributes() { Vitality = 2, Strength = 1 }
};
```

Example bow used for testing:

```
Weapon testBow = new Weapon()
{
    ItemName = "Common bow",
    ItemLevel = 1,
    ItemSlot = Slot.SLOT_WEAPON,
    WeaponType = WeaponType.WEAPON_BOW,
    WeaponAttributes = new WeaponAttributes() { Damage = 12, AttackSpeed = 0.8 }
};
```

Example cloth head armor used for testing:

```
Armour testClothHead = new Armor()
{
    ItemName = "Common cloth head armor",
    ItemLevel = 1,
    ItemSlot = Slot.SLOT_HEAD,
    ArmourType = ArmourType.ARMOUR_CLOTH,
    Attributes = new PrimaryAttributes() { Vitality = 1, Intelligence = 5 }
};
```

NOTE: These should not be globally declared and reused, they should be redeclared in each test method (best practices).

Feel free to change the item names or use different items or character classes for the tests. As long as they fulfill the testing requirements.

The item and equipment tests should cover the following behaviors:

1) If a character tries to equip a high level weapon, InvalidWeaponException should be thrown.
    - Use the warrior, and the axe, but set the axes level to 2.
2) If a character tries to equip a high level armor piece, InvalidArmorException should be thrown.
    - Use the warrior, and the plate body armor, but set the armor's level to 2.
3) If a character tries to equip the wrong weapon type, InvalidWeaponException should be thrown.
    - Use the warrior and the bow.
4) If a character tries to equip the wrong armor type, InvalidArmorException should be thrown.
    - Use the warrior and the cloth armor.
5) If a character equips a valid weapon, a success message should be returned
    - "New weapon equipped!"
6) If a character equips a valid armor piece, a success message should be returned
    - "New armour equipped!"
7) Calculate DPS if no weapon is equipped.
    - Take warrior at level 1
    - Expected DPS = 1*(1 + (5 / 100))
8) Calculate DPS with valid weapon equipped.
    - Take warrior level 1.
    - Equip axe.
    - Expected DPS = (7 * 1.1)*(1 + (5 / 100))
9) Calculate DPS with valid weapon and armor equipped.
    - Take warrior level 1.
    - Equip axe.
    - Equip plate body armor.
    - Expected DPS = (7 * 1.1) * (1 + ((5+1) / 100))

We could add tests before the DPS tests to see if the total primary attribute is being added properly, but to save some time we went straight to DPS. Ideally you wouldn't do this skip.