**Game Outline:**

I have designed a game in which the player must battle monsters and solve puzzles in order to progress to the next room. Each monster has a unique difficulty value, this affects their health and their attack damage. The player must progress through 7 rooms in order to complete the game. Each room can contain a weapon and/or a spell. These can be added to the player’s inventory and Weapons can be equipped. The player can use spells to heal their character, which removes the spell from their inventory.

**Inheritance:**

I’ve used inheritance in my code to better control the behaviour of Rooms, Creatures and Items. The Room class is the superclass that PuzzleRoom and MonsterRoom inherit from. Items is a superclass for Hint, Spell and Weapon. Creatures has the largest inheritance hierarchy: Creature is the superclass and Monster and Player are both child classes. The Monster class has 5 subclasses: Dragon, Shulker, Skeleton, Warden and Witch that inherit behaviour from Monster~~.~~ *~~[show inheritance tree diagram (“Creature Inheritance Drawing.png”), show this being true in the code too]~~*

**Dynamic Overloading:**

The Monster class contains a virtual method, which child functions have the option to override to implement their own behaviour. This is an example of dynamic polymorphism or runtime polymorphism *~~[show screenshot of override method in Skeleton class versus virtual method in Monster class]~~*

~~As an example of static polymorphism is shown in the two turn decision methods. I use static polymorphism to be able to display two types of instructions to the Console depending on the type of room that the Player is in.~~ *~~[show UserInterface.ShowTurnDecisions]~~*

**Static Overloading**

I have used polymorphism in to display different messages to the console depending on the type of room given as a parameter to the function. *~~[show UserInterface.ShowTurnDecisions]~~*. Using polymorphism in this way helps me to reduce the maintenance of code, if my code needs to be changed, it only has to be done in one place rather than in multiple. I think it enhances readability, which helps others understand my code quickly.

**Other Polymorphism Stuff**

I have used polymorphism a lot in my code. I use polymorphism to let me iterate over any object that implements the IEnumerable interface, allowing me to generalise my code (rather than creating a new function and logic flow for each type of collector.) *~~[show screenshot of UserInterface.DisplayEnumerable]~~.* My code also implements an inventory for the player. This is a list of type Item which contains both Spells and Weapons, which are subclasses and inherit from Item. This allows me to store various types items in a list whilst maintaining the ability to sort them into their specific types *~~[screenshot of Inventory.GetWeaponsInInventoryAscending() and Inventory.GetSpellsInInventory()].~~*

**LINQ**

The player’s inventory uses a List<Item> *~~[screenshot of inventory List<item>]~~* . Items is a superclass for Spells and Weapons that can be stored within the inventory. The user can pick up either item and add it to their character’s inventory. They can equip the weapon to attack the monster or use the spell to heal their character. *~~[Items inheritance tree diagram]~~*

I use LINQ to sort the Inventory into two smaller lists, which only contain one type of item. I’ve used this in my project to allow the user to view the Inventory of the Player in an organised way. *[Show screenshot of user displaying the user’s inventory].* I automatically sort the spells by descending amount of health that they heal, so the most important spells are ranked at the top of the list. The user also has the option to sort the weapons in 3 ways: by ascending and descending damage and alphabetically by name. *[Show screenshot of user having the functionality to sort the damage of the weapons (screenshot from the last level so they’ve got a range of weapons to sort from)]*.

**Interfaces**

Both of my interfaces, IHasSummary and ICanDamage only contain abstract methods. Abstract methods contain just a function signature. The interface is similar to a contract, any class that uses the interface must implement the behaviour of the functions that were declared in the interface *[Show screenshot of code for interfaces]*

IHasSummary has been used on subclasses of Item, signifying that it implements the functionality to generate a brief summary of key facts about the object. For Weapons this is the name of the weapon and the average damage that it does. For Spells this is the healing power of the spell if it was to be used. *[Show screenshot of IHasSummary and Item.CreateSummary() being called (perhaps when the user shows the inventory being used]*

ICanDamage has been used on subclasses of Creatures. All classes that implement ICanDamage adhere to a predefined contract, implementing the functionality to hold a Weapon and Calculate the amount of damage that the weapon does. I decided to use an interface versus classes inheriting this behaviour from their parents because this would, in the future, allow for friendly Creatures- this could be used for the player to carry out quests for example. *[Shows screenshot of ICanDamage interface, Show screenshot of the logic that has been implemented]*

**Virtual/Abstract Methods**

I have used virtual and abstract methods multiple times throughout my code.

The most frequently used example of a virtual method (and use of override methods) is in Monster.GetAttackMessage(). Each child class of Monster implements its own behaviour of GetAttackMessage()- for example Dragon’s GetAttackMessage() mentions that the dragon breathes fire and the witch’s mentions that it casts a spell. *[Show Monster.GetAttackMessage() and Dragon/Witch.GetAttackMessage()]*

**Protected Access Control**

I have used protected access control inside the Creature class. This is used to allow all subclasses to control specific properties of the superclass- health and name. Subclasses can take control of the health value of the class which has helped to implement some of the logic and behaviour of Spells. A Spell is used by the Player to increase the health of the character. *[Show Creature.\_health and Player.UseSpell]*

**Testing Class & Test Log**

I implemented two testing classes.

One of my testing classes is designed to carry out unit tests, which verifies that my classes and methods function as intended. *[show unit tests class]* The unit tests cover methods from most of the classes, such as game, spell, weapon, player and all of the monsters. I check that each class successfully instantiates, if this were to fail then other tests would fail too. *[show unit tests explorer (tests being passed]*

My unit tests outputs all test results to a log. *[show log file]*

I was sceptical about adding unit tests at first, I was worried that the extra effort to add them to my project would outweigh the saved effort in debugging but implementing them has proven me wrong.

The other test class I use contains other basic checks that verifies that the type and range of parameters is as expected during runtime. Hypothetically these shouldn’t be needed as my project features significant error handling but if they were to fail, this would catch any errors. *[show the other testing class and each of the methods within the class being used]*

**Error Handling**

I also have used error handling in my code whenever I need to prompt the user for input. It is used to verify that the input is within the intended range and in the required format. An uncaught invalid input would cascade and cause additional errors with logic in other parts of my code. *[show an example of my error handling during user input]*