**Game Outline:**

I have designed a game where the player must battle monsters and solve puzzles to progress to the next room. The player must progress through 7 rooms to complete the game. Each monster has a unique difficulty value, which affects its health and attack damage. Each room has a chance of containing a weapon or healing spell. These can be added to the player’s inventory. The user wins when he progresses through all the rooms or loses when his player character’s health is 0. The monsters have a random chance of fleeing if they are below a certain health value *~~[start of game screenshot]~~*

**Inheritance:**

I’ve used inheritance in my code to define common behaviour and share it with classes that inherit from it, reducing code duplication and improving maintenance. PuzzleRoom and MonsterRoom inherit from Room. Items is a superclass for Hint, Spell and Weapon. Creatures has the largest inheritance hierarchy: Creature is the superclass for Monster and Player and Monster class has 5 child classes: 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 (dynamic runtime polymorphism):**

The Monster class contains a virtual method, which child functions have the option to override to implement their behaviour, in this case, their unique attack message. *~~[show screenshot of override method in Skeleton class versus virtual method in Monster class]~~*

**Static Overloading**

I have used polymorphism 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, by allowing changes to be made in one place rather than in multiple~~. I think it enhances readability, which helps others understand my code quickly.~~

**Other Polymorphism Stuff**

I use polymorphism to let me iterate over any container 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 of 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 to allow the user to view their Inventory 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. 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. Interfaces are 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 some 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]~~*

**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 which is used to implement some of the logic and behaviour of Spells. *~~[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 verify that my classes and methods function as intended~~.~~ *~~[show unit tests class]~~* The unit tests cover methods from most of the classes. 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 output 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. Just in case this error handling fails, debug.assert is used to ensure that invalid input is caught. 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]~~*

**Saving and Loading**

I have managed to implement a way to save your current game save and load previous game saves in my game. I can achieve this by using object serialisation to serialise a class that represents the state of all other classes in the current playthrough of the game. *~~[show SaveHandler.cs~~]* This class is the GameState class. The GameState class contains the current room number, player object, list of rooms and the statistics class. This is enough data for me to be able to recreate all the objects used in the game from a save file. *~~[show gamestate file]~~*

When the user starts the game, they have the option to load a game from a file or create a new game and overwrite their current save file. *~~[show screenshot of the start of the game]~~* The user has the option to save the game at any point, so games can even be saved mid fight. *~~[Show the user having the option to save a game during a monster fight (Can save mid game.png]~~*

The state of the game is saved to a file in the user’s My Documents folder with the name DungeonExplorerSave.json *[show C://My Documents/DungeonExplorerSave.json]*