CMP1903M Object Oriented Programming 2024 – 2025

# Assignment 2: Report

[Expand the sections as necessary]

Name: Charles Harrison

Student ID: 25320877

Code repository URL: https://github.com/notxcharles/CMP1903-Object-Oriented-Programming-A02

Video URL: https://youtu.be/UGXkPrQA8aE

# Application:

1. **Reflection on the OO features within your code. (~400 words)**

I use encapsulation throughout my code to group attributes and methods to a single class. Each class has a separate and obvious purpose. There are multiple sub-types of the Monster and Room base classes that have differing behaviour and override their parent’s virtual methods.

I’ve used inheritance in my Room, Creature and Item classes. Room provides the basic methods for PuzzleRoom and MonsterRoom which build on the base methods to contain either a puzzle or a monster. The Item class is a parent class for the Spell and the Weapon classes. Similarly to the rooms, these build on the functionality of the Item and can be used by the player to either use a Spell to heal or to do damage to a Monster with a Weapon.

I make use of abstraction to hide the logic and how attributes and methods function “behind the scenes”. This hides how some of the code works and functions, and just provides the user with exactly what they ask for.

I’ve used polymorphism in parts of my code. The most apparent use of polymorphism is in the inventory. The inventory is a list of items, but the Weapon class and the Spell class both inherit from Item. Using polymorphism allows me to have more control over the inventory, I can separate the Weapons and Spells and display them separately. I can also treat them as their parent class (Item) to get the summary of the item.

I used two interfaces: ICanDamage and IHasSummary. ICanDamage is specifies that the class should be able to deal damage, so it needs to implement a function to calculate the damage it can do and to create a string that can be displayed in the console demonstrating that it has attacked. I decided that this needed an interface because this would allow for future implementation of Creatures that weren’t hostile- perhaps this could be used to give the player quests. IHasSummary was designed for Spells and Weapons. IHasSummary indicates that the class must have CreateSummary(), a function which briefly describes the object and what it does.

I use overloading for constructors and certain methods. This allows me to use the same name for the method whilst having different parameters, letting me customise the behaviour of the class or method depending on the input given to me.

I use method overriding to allow subclasses to have different behaviour to a method in a superclass. This is used frequently in Monster and all children. Monster implements a base method for describing an attack, and the subclasses can implement a more specific description.

1. **Reflection on your handling of error conditions in your code. (~200 words)**

I take special care to ensure that I validate any input and handle error conditions transparently.

Whenever my program requests input from the user, I check that it is of the correct type and within an expected range. This could be an integer from 0 to 9, a character or a string. If the user enters an invalid input, then the program displays a message explaining the issue and prompts the user for a valid response. Ensuring valid input helps avoid runtime errors and prevents any of the game’s logic breaking.

I also check that the player’s inventory is not empty before proceeding with any additional logic to alter it. Checking if the inventory is empty beforehand can help to avoid any other errors which may occur as a result of the empty list, such as trying to access elements out of bounds.

Other error handling includes checking whether a monster is alive before the player tries to attack it, checking whether the puzzle in a room has been solved before unlocking the door and giving the player’s inventory a maximum size.

For each error, I provide user feedback explaining what went wrong. I think this helps the game be more intuitive to understand for the user and helps users understand why the action has not happened.

1. **Reflection on your testing activities: What did you test, and how did you do it? (~400 words)**

As part of my project, I created two different testing classes. The primary testing class is for unit tests, to check that my classes and methods function as intended. The second testing class is the more basic, used to verify that certain parameters for methods fall within an expected range. While the unit tests should verify that all methods are returning values as expected, the secondary testing class acts as a failsafe to catch any edge cases that may slip through.

The unit tests class ensures that essential classes and methods work and return the correct value as intended. My unit tests cover methods from a range of different classes- Game, Spell, Weapon, Player, Dragon, Skeleton, Witch, Warden and Shulker classes.

The first (and arguably the most important) unit tests that I have implemented is to check that all classes successfully instantiate. If one of the instantiation tests fails, then subsequent unit tests for that class are likely to fail.

The monster class contains a virtual method called GetAttackMessage(). The behaviour of this method can be overridden by any subclasses of Monster (Dragon, Skeleton, etc) by using the override keyword when declaring the method. Each subclass overrides this method to return a unique string. I’ve created unit tests to confirm that all classes that implement their unique behaviour is done correctly and returns a string that is not empty.

I have other unit tests to check that methods follow the intended logic and return a value within the range that is needed for the program to continue functioning. This includes checks for a Weapon’s damage range, spell effects, inventory management, ensuring methods are in place to catch erroneous user input and other gameplay mechanics

My unit test class keeps a log (testResults.log) that records which tests pass or fail. This was particularly helpful in cases where others were playtesting my game and would encounter an error.

Using unit tests within my project proved to be very useful. It was nice to be able to catch bugs early, almost in real time, instead of catching them later when I was playtesting the game. The tests helped me to refactor and optimise certain parts of my code without worrying about accidentally affecting the functionality and creating new bugs.

I was worried that the extra effort to add unit tests to my project would outweigh the effort that it saved through debugging, but I am glad that I implemented them. It would take a minute at most to run all of my unit tests, but that immediately confirmed if something was broken and if so, what it was.

In a professional environment where money, customers, customer data, and business reputation are on the line, unit tests act as a first line of defence to avoid bugs affecting any critical infrastructure

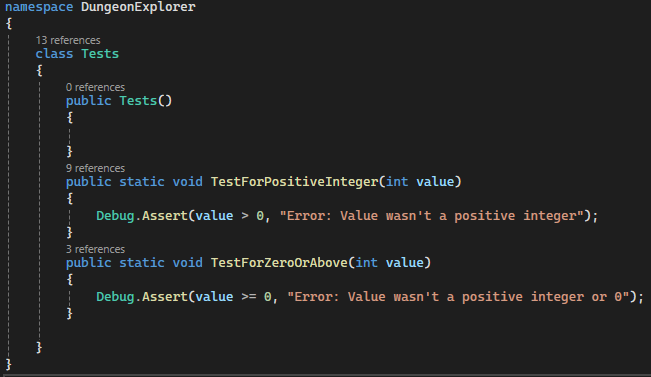
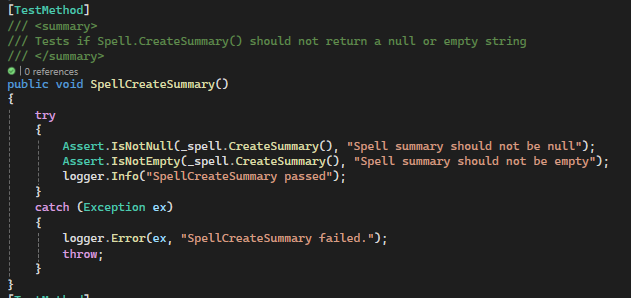
1. **Include evidence of the tests (screenshots are OK)** 

Figure : Testing for Positive Integers and Non Negative Integers

Figure : Check that Spell.CreateSummary returns a string that is not null or empty. Similar functionality was implemented for all other classes

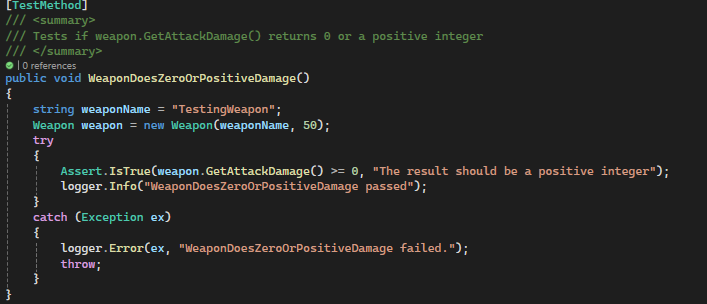
1. ****

Figure : Unit test to determine if a Weapon never returned a negative number

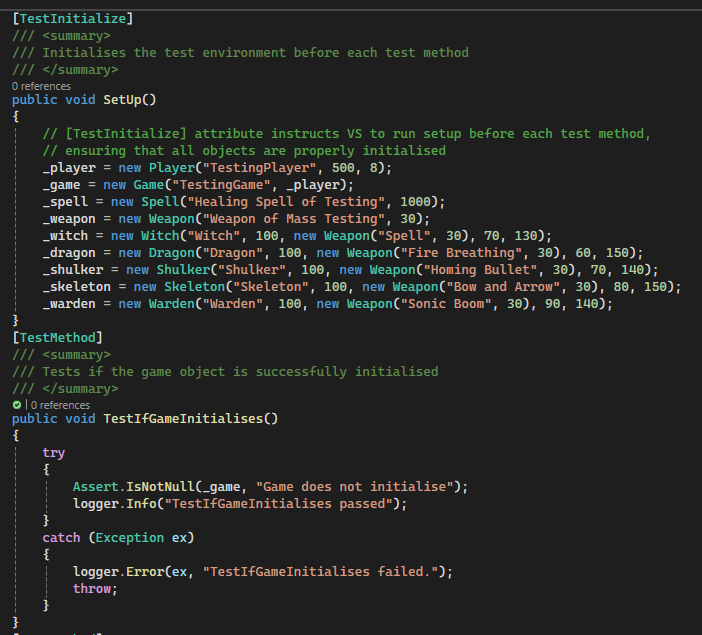
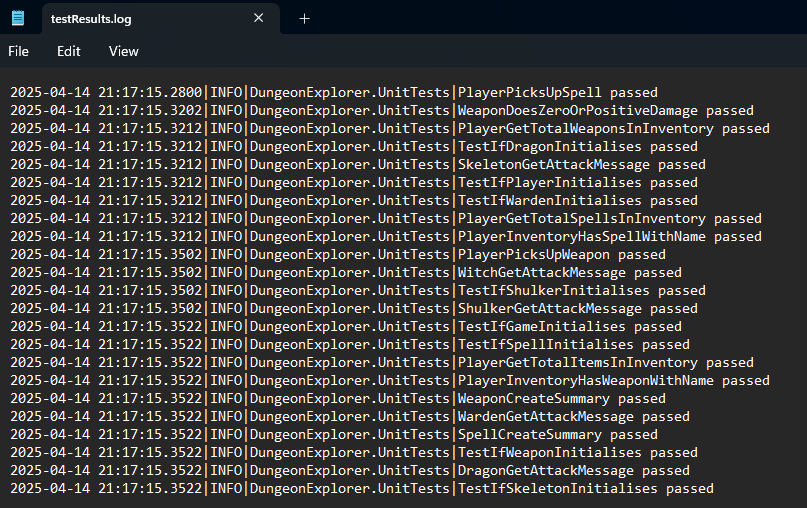
****  ****

Figure : Screenshot of a unit test. Some of my unit tests tested if a class could be initialised.

Figure : Screenshot of testResults.log. This demonstrates the status of all unit tests- whether they have passed or failed

# Reflection & Feedback:

Figure : Unit test to test whether the player can pick up an item

Figure : Screenshot demonstrates that I had a unit test to test instantiation of many different types of classes

1. **What was the most important thing you learned from this assessment? (< 200 words) Eg: I learned that If you don't think every day is a good day - try missing a few. You'll see.**

The most important thing that I’ve learned from this assessment was how to create unit tests. Being able to avoid many logic errors within a codebase is very useful. It helped to catch errors in development, before I started playtesting my game, which has saved me a lot of time that would have been spent debugging.

Interfaces enforce a contract, ensuring that any class which implements them provides the required methods and associated logic. Interfaces also meant that I could control which classes implemented them, so I don’t have to force a class to comply (through inheritance) and create behaviour that was not intended or would never be used.

Operator overloading was a concept that I had not heard of or thought existed. Even though I didn’t implement this in my code, I can see how this would be important within a Math class or within advanced physics class where the user required more functionality than just adding two numbers (such as multiplying vectors)

I decided to dedicate some time to planning how my game would work and what different classes would be needed. This helped me to brainstorm the methods needed from each class and be able to highlight any repeated logic that could be simplified using inheritance.

**What was the most challenging aspect of this assessment and how did you approach it? (<200 words)**

There isn’t much in this world more infuriating than mixing Microsoft Word and images. I’m incredibly stubborn at times and didn’t want to let it get the better of me. Albert Einstein said that "If you keep doing the same thing, you'll keep getting the same result", clearly he has never used Microsoft Word.

Developing my understanding of what a testing class was and how it was to be used and implemented within the project was challenging. I was unsure where to even start with this. My approach was to use Google, Reddit, StackOverflow, ChatGPT and CoPilot to see examples of implementation which I could use to focus my research further. This led to me discovering unit tests, which meant that I needed to understand what a unit test was (which was quite different from what I had expected a unit test to be). I then researched how to implement them. This was made easier through following a YouTube tutorial and referencing Microsoft’s extensive C# documentation.

I found the implementation of saving and loading the game to be overwhelming at first. I had never implemented such functionality before so my initial idea to solve this problem was very convoluted approach to solving this problem. Through a similar method to researching and learning about testing class functionality, I manage to solve this issue.

1. **What would you particularly like to receive feedback on in this assessment?**

Unit testing is new to me so I would appreciate feedback on how I’ve implemented my tests. My current approach involves using several try/catch blocks which results in some repeated code. From what I've researched, this seems difficult to avoid, but I welcome your thoughts on whether this is the best approach or if there are any better alternatives.

I would like to know if there are any improvements that I could make to any of the object-oriented principles that I have followed. I do not have much experience with C#; however it is clear that the object-oriented principles have influenced many other languages.

# Assignment 2 Checklist

All of the elements in a section must be checked for it to be considered for that grade (this isn’t guaranteed though). All previous elements must also be complete for a grade to be considered.

## Pass standard:

|  |  |
| --- | --- |
| The code compiles and runs.  ***Implemented*** |  |
| The player can explore at least two interconnected rooms.  ***Implemented*** |  |
| Object instantiation, method calls evident.  ***Implemented*** |  |
| Rooms have descriptions and can contain one item or one monster.  ***Implemented:*** *Rooms can have names and descriptions and contain multiple items or a monster* |  |
| The Testing class is used.  ***Implemented:*** *Testing class is used to verify user input* |  |
| The player can pick up items and battle monsters.  ***Implemented:*** *Player can pick up weapons and spells. Player can attack monsters* |  |
| Implement at least one abstract class (e.g., Creature) with Player and Monster inheriting from it.  ***Implemented*** |  |
| Use inheritance to define a hierarchy for items (e.g., Item with subclasses Weapon and Potion).  ***Implemented:*** *Item is a superclass for Weapon and Potion. Creature is a superclass for Monster and Player class. Monster class is the superclass for 5 different types of monsters, Dragon/Skeleton etc* |  |
| Demonstrate simple method overriding for Creature subclasses (e.g., different attack methods for Player and Monster).  ***Implemented:*** *All child classes of monster have different behaviour.* |  |
| Handle invalid commands gracefully without crashing the program.  ***Implemented:*** *Each time that the user is prompted for input, their input is verified to be within the correct range of expected values* |  |

## 2:2 standard:

|  |  |
| --- | --- |
| Include at least three interconnected rooms with navigation between them.  ***Implemented:*** *the project contains seven rooms* |  |
| Rooms can contain multiple items or monsters.  ***Implemented:*** *Rooms can contain multiple type of items (weapon, spell), or a monster or puzzle* |  |
| Implement at least one interface (e.g., IDamageable) applied to both Player and Monster.  ***Implemented:*** *Two interfaces. ICanDamage and IHasSummary* |  |
| Use a collection (e.g., List<Item>) to manage the player’s inventory.  ***Implemented:*** *Inventory is a List<Item> which contains Weapon and Spell types* |  |
| Allow players to view and use multiple items.  ***Implemented:*** *Player can equip different weapons from their inventory. Player can use Spells, which heal the player.* |  |
| Implement dynamic polymorphism (e.g., items like Potion and Weapon have different effects when used).  ***Implemented:*** *Potions positively change Player’s health, Weapons are used to attack Monsters. All child classes of Monster use an overridden method from Monster* |  |
| Use LINQ queries for at least one task, such as filtering inventory items or sorting monsters by strength.  ***Implemented:*** *Inventory is sorted by item types and can be by weapon damage.* |  |
| Handle more complex errors, such as trying to attack a monster in a room with no monsters.  ***Implemented:*** *Player will be told that they can’t attack a monster that doesn’t exist. Player told if their input is invalid and informed of the desired range of input values* |  |
| Method calls from ‘Main’ to methods in other classes  ***Implemented:*** *Program.cs initialises the player and starts the game. Game.Start() handles the logic flow of the game but utilises other functions to do so* |  |
| Class definitions show encapsulation.  ***Implemented:*** *Every method and property is declared as private unless another class needs to directly access it. Often use private setters and public getters if the property should not be modified by others.* |  |
| A Statistics class is used  ***Implemented:*** *Statistics.cs, can track the player’s dealt damage, damage received, number of rooms completed and their score* |  |

## 2:1 standard:

|  |  |
| --- | --- |
| Include at least five interconnected rooms with varied challenges (e.g., some rooms have locked doors requiring a key).  ***Implemented:*** *Two types of rooms, one containing a monster and the other involving a guessing puzzle. The monster must be defeated/puzzle solved before the door to the next room is unlocked)* |  |
| Monsters have different levels of difficulty and attributes (e.g., health, strength).  ***Implemented:*** *Monsters have a random difficulty (Monster.CalculateRandomDifficulty). This difficulty influences the health and attack damage of the monster (health = health \* difficulty)* |  |
| Implement inventory management that supports item limits and item removal (e.g., discarding or using items).  ***Implemented:*** *Inventory.Add contains logic for setting a maximum inventory size and not allowing the user to pickup items if it is full. Also contains logic for removing items from inventory,* |  |
| Use LINQ for multiple inventory-related tasks (e.g., finding all healing items or the strongest weapon).  ***Implemented:*** *Inventory can be sorted by type of item, or by damage of any weapons* |  |
| Implement at least two interfaces (e.g., IDamageable, ICollectible) and apply them appropriately to classes.  ***Implemented:*** *Two interfaces, ICanDamage (classes must implement a way to calculate their attack damage) and IHasSummary (classes must implement a way to summarise what they are and do)* |  |
| Demonstrate advanced inheritance with deep class hierarchies (e.g., Monster subclassing into Goblin, Dragon, etc., each with unique behaviours).  ***Implemented:*** *Dragon/Skeleton inherit from Monster, which inherits from Creature. Player also inherits from Creature* |  |
| Use polymorphic methods for both items and monsters.  ***Implented:*** *The player has an inventory, managed by a list that contains Item. It can contain both Weapon or Spell types. MonsterRoom.cs constructor accepts any subclass of Monster that inherits from Monster* |  |
| Provide clear feedback to the user for invalid actions (e.g., “You can’t attack; there’s no monster here.”).  ***Implemented:*** *Best see in UserInterface.GetInput() if user provides an invalid input they are told what input they should enter. User is also told if they can’t fight a monster if it is dead, can’t pickup an item if their inventory is full etc.* |  |
| Testing class uses verification methods in code (such as debug.assert()) to check code.  ***Implemented****: one of the testing classes I use performs debug.assert tests during run time that verifies that method parameters are within an accepted range* |  |

## First standard:

|  |  |
| --- | --- |
| Use of virtual/abstract methods  ***Implemented:*** *Used virtual methods within Monster class. I use two interfaces, both contain function signatures (abstract methods)* |  |
| protected access control is used in class hierarchy)  ***Implemented:*** *protected health and name used in Creature.cs so that only child classes can modify the value* |  |
| The Testing class implements a way to record testing data (through a log file for example  ***Implemented:*** *A testing class contains unit tests. The result of these tests are output to a testing log file* |  |
| Create a fully navigable game map with at least seven interconnected rooms, including special rooms with unique challenges (e.g., puzzles, traps).  ***Implemented:*** *7 rooms, a mix of Rooms containing a monster or a puzzle to be solved before progressing* |  |
| Add randomness to gameplay (e.g., monsters or items appear randomly in rooms).  ***Implemented:*** *Weapons do a random amount of damage modelled using a normal distribution. Weapons and Rooms have random names and descriptions* |  |
| Allow the player to win or lose the game based on health, inventory management, or defeating a final boss.  ***Implemented:*** *Player character loses the game when health becomes 0. Player wins the game by reaching the end room. Player can manage their inventory, use and remove items from it.* |  |
| Implement and justify design decisions (e.g., why specific classes use inheritance vs. interfaces).  ***Implemented:*** *I use inheritance when any subclasses also need the behaviour of other classes, but is a different thing with it’s own behaviour (Dragon is a type of monster, but is different and contains it’s own unique behaviour.) I use interfaces when I need to ensure a class complies and implements certain behaviour* |  |
| Use LINQ extensively, such as filtering, grouping, or sorting items and monsters.  ***Implemented:*** *LINQ is used to sort the inventory (see Inventory.GetWeaponsInInventoryAscending, Inventory.GetWeaponsInInventoryDescending, Inventory.GetWeaponsInInventorAlphabetically). Also used to sort the inventory by type (eg just return a list of the spells – Inventory.* *GetSpellsInInventory() or Inventory.GetTotalWeaponsInInventory())* |  |
| Allow players to interact dynamically with inventory (e.g., sort by item type, use the strongest weapon automatically).  ***Implemented:*** *The player can chose to sort the weapons in their inventory by damage (ascending or descending) or alphabetically by name. The logic for this is rooted in Inventory.GetWeaponsInInventory() which calls to other methods (GetWeaponsInInventoryAscending, GetWeaponsInInventoryDescending, GetWeaponsInInventorAlphabetically)* |  |
| Demonstrate dynamic polymorphism through overridden methods for both items and creatures.  ***Implemented:*** *Monster.cs contains a virtual method GetAttackMessage(), all subclasses of monster (Dragon.cs, Shulker.cs, Skeleton.cs, Warden.cs, Witch.cs) contain override methods implementing their own behaviour, such as “The dragon breathed fire” or “The witch cast a damage spell”* |  |
| Use static polymorphism where applicable, such as overloaded constructors or methods.  ***Implemented:*** *Frequently used in constructors for classes. I have two functions UserInterface.DisplayRoomInformation with different parameters, one for PuzzleRooms and one for MonsterRooms.* |  |
| Include a scoring system or player progression (e.g., levelling up after defeating monsters).  ***Implemented:*** *The user’s score is calculated after each round. The logic for this is located within Statistics.CalculateScore(). The equation for calculating the score = (Total Damage Dealt \* 1.5) – (Total Damage Received \* 0.9) + (Number of completed rooms \* 10), rounded to the nearest integer* |  |
| Implement save/load functionality to persist game state.  ***Implemented:*** *Most of the logic for this is contained within the SaveHandler.cs class, some of the logic (such as preparing the gameState object to be saved is in Game.cs and GameState.cs* |  |
| Add a basic AI for monster behaviours (e.g., monsters may attack first or flee when weak).  ***Implemented:*** *Monster has a random chance of fleeing when it is below a certain health value. This logic is in Monster.WantsToFlee()* |  |