CMP1903M Object Oriented Programming 2024 – 2025

# Assignment 2: Report

[Expand the sections as necessary]

Name:

Student ID: 25320877

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

Video URL:

# 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 class. 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 overriding to allow subclasses to have different behaviour to a method in a superclass. This is used frequently in Monster and Monster’s children. Monster implements a base method for describing an attack, where as the subclasses have a more specific description.

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

I take special care to ensure that any time I ask the user for input that I check the recorded input before proceeding through my code. Typically, this would be asking the user to input an integer. I check that the integer is an integer, as the user may have provided a character or any other illegal character. I then check that it is in a specified range. If it is then the function returns the value to be used within my code. If not, or the user has provided illegal input, then the user is told what their input was and what their input should be (such as the accepted range of values). I am thorough with checking user input because this can very quickly lead to errors that are hard to catch.

I also check that the Player’s inventory is not empty before proceeding with any additional logic to alter the inventory. Checking if the inventory is empty before hand can help to avoid any other errors which may occur as a result of the empty list.

Other error handling includes checking if 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 checking if the max inventory size is too large,

For each error, I inform the user of why the error occurred. I think this helps the game be more intuitive to understand for the user and helps the user understand why the action that they may have asked for has not happened.

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

I used a testing class to ensure that arguments for methods were valid and had been declared. This ensured that any further code and logic could function as intended without encountering any errors. Things such as checking that objects were instantiated were used.

I also used a testing class to check that a variable (such as an integer) was within a specified range, such as checking that an integer was positive or that it was not negative.

I’ve used unit tests in my project to ensure that essential classes and methods work as intended. I have unit tests for the Game, Player, Spell, Weapon and all different type of Monsters. I use unit tests to check that all methods follow their intended logic, return the correct value within the range and in the format that I expected. Utilising unit tests helped me to catch any simple logic errors that I had made before I had even started the game

My unit test class keeps a log (called testResults.log) which contains information about which tests passed or failed. This is particularly helpful in cases where other’s are playtesting my game and may encounter an error.

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

# Reflection & Feedback:

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 before I started playtesting my game which has saved me a lot of time that would have been spent debugging.

Using Interfaces let me to create logic and ensure that I have created sufficient logic for classes that implemented the interface. Interfaces also meant that I could control which classes implemented the interface, I didn’t have to force a class to comply and create behavior 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 code that could be simplified using inheritance.

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

**Eg: I started painting as a hobby when I was little. I didn't know I had any talent. I believe talent is just a pursued interest.**

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

Because unit tests are new to me, I would like to receive feedback on the usage of my unit tests. My implementation of the unit test contains a lot of try/catch blocks which results in a lot of repeated code. I have done some research into this and it seems like it cannot be avoided but I would be interested to get your opinion on this.

# 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. |  |
| The player can explore at least two interconnected rooms. |  |
| Object instantiation, method calls evident. |  |
| Rooms have descriptions and can contain one item or one monster. |  |
| The Testing class is used. |  |
| The player can pick up items and battle monsters. |  |
| Implement at least one abstract class (e.g., Creature) with Player and Monster inheriting from it. |  |
| Use inheritance to define a hierarchy for items (e.g., Item with subclasses Weapon and Potion). |  |
| Demonstrate simple method overriding for Creature subclasses (e.g., different attack methods for Player and Monster). |  |
| Handle invalid commands gracefully without crashing the program. |  |

## 2:2 standard:

|  |  |
| --- | --- |
| Include at least three interconnected rooms with navigation between them. |  |
| Rooms can contain multiple items or monsters. |  |
| Implement at least one interface (e.g., IDamageable) applied to both Player and Monster. |  |
| Use a collection (e.g., List<Item>) to manage the player’s inventory. |  |
| Allow players to view and use multiple items. |  |
| Implement dynamic polymorphism (e.g., items like Potion and Weapon have different effects when used). |  |
| Use LINQ queries for at least one task, such as filtering inventory items or sorting monsters by strength. |  |
| Handle more complex errors, such as trying to attack a monster in a room with no monsters. |  |
| Method calls from ‘Main’ to methods in other classes |  |
| Class definitions show encapsulation. |  |
| A Statistics class is used |  |

## 2:1 standard:

|  |  |
| --- | --- |
| Include at least five interconnected rooms with varied challenges (e.g., some rooms have locked doors requiring a key). |  |
| Monsters have different levels of difficulty and attributes (e.g., health, strength). |  |
| Implement inventory management that supports item limits and item removal (e.g., discarding or using items). |  |
| Use LINQ for multiple inventory-related tasks (e.g., finding all healing items or the strongest weapon). |  |
| Implement at least two interfaces (e.g., IDamageable, ICollectible) and apply them appropriately to classes. |  |
| Demonstrate advanced inheritance with deep class hierarchies (e.g., Monster subclassing into Goblin, Dragon, etc., each with unique behaviours). |  |
| Use polymorphic methods for both items and monsters. |  |
| Provide clear feedback to the user for invalid actions (e.g., “You can’t attack; there’s no monster here.”). |  |
| Testing class uses verification methods in code (such as debug.assert()) to check code. |  |

## First standard:

|  |  |
| --- | --- |
| Use of virtual/abstract methods |  |
| protected access control is used in class hierarchy) |  |
| The Testing class implements a way to record testing data (through a log file for example |  |
| Create a fully navigable game map with at least seven interconnected rooms, including special rooms with unique challenges (e.g., puzzles, traps). |  |
| Add randomness to gameplay (e.g., monsters or items appear randomly in rooms). |  |
| Allow the player to win or lose the game based on health, inventory management, or defeating a final boss. |  |
| Implement and justify design decisions (e.g., why specific classes use inheritance vs. interfaces). |  |
| Use LINQ extensively, such as filtering, grouping, or sorting items and monsters. |  |
| Allow players to interact dynamically with inventory (e.g., sort by item type, use the strongest weapon automatically). |  |
| Demonstrate dynamic polymorphism through overridden methods for both items and creatures. |  |
| Use static polymorphism where applicable, such as overloaded constructors or methods. |  |
| Include a scoring system or player progression (e.g., levelling up after defeating monsters). |  |
| Implement save/load functionality to persist game state. |  |
| Add a basic AI for monster behaviours (e.g., monsters may attack first or flee when weak). |  |