University of South Australia

School of Information Technology and Mathematical Sciences

COMP 3023 Software Development with C++

Assignment

Dungeon Crawler

Introduction

This document specifies the requirements for the first assignment of the Software Development with C++ course, SP5 2018. This assignment will develop your skills by allowing you to put into practice what has been taught in class during the first 5 weeks of the course.

The assignment will require you to design, implement, test, and debug a text-based Roguelike dungeon crawler game using object-oriented methods and the application of appropriate design patterns. In addition, it will require you to use version control to keep track of your implementation as you progress through the assignment and to document your classes appropriately using Doxygen comments. The work in this assignment is to be performed **individually** and will be **submitted via LearnOnline** at the end of week 8: **due by 14 September 2018 11:59 PM**. Refer to section *Submission Details* for specific instructions.

If any parts of this specification appear unclear, please ensure you seek clarification.

Learning Outcomes

After completing this assignment, you will have learnt to:

- Design a class hierarchy using UML
- Apply Object-Oriented principles (encapsulation, reuse, etc.) and Design Patterns to the software design
- Implement the software design in C++ using class inheritance and polymorphism features
- Write well-behaved constructors and destructors for C++ classes
- Use C++ pointers (including smart pointers) and dynamic memory allocation and management
- Use stream I/O and manipulators for formatted input and output
- Write code adhering to coding standards, guidelines and good programming practices

Design Patterns

In developing the class design for the Dungeon Crawler game, the following the Design Patterns **must** be incorporated:

- Singleton
- Builder
- Decorator
- Strategy

This section provides brief descriptions of each relevant design pattern in turn. For more information refer to the book from which these summaries are derived:
Gamma, E, Helm, R, Johnson, R and Vlissides, J 1995, *Design patterns: elements of reusable object-oriented*

software, Addison-Wesley, ISBN: 978-0-201-63361-0.[1] Be aware that the C++ examples from the book are for an older version of C++. If using them for guidance, they must be updated to C++ 14.

Adequate descriptions of the above patterns along with some code examples can also be found on Wikipedia:

- Singleton: https://en.wikipedia.org/wiki/Singleton_pattern
- Builder: https://en.wikipedia.org/wiki/Builder_pattern
- Decorator: https://en.wikipedia.org/wiki/Decorator_pattern
- Strategy: https://en.wikipedia.org/wiki/Strategy_pattern

Singleton

The Singleton pattern addresses the problem of ensuring that a class has only one instance. Moreover, the one instance is easily accessible but in a controlled fashion. Such a pattern arises in situations where a single, globally accessible interface to a system or subsystem is required and, hence, is often used with other patterns such as Façade and Abstract Factory. An example of singleton a singleton is an application wide configuration. Rather than parsing the configuration file multiple times and storing multiple copies in memory, the configuration file should be read once and stored once. Moreover, since various parts of the application may need to access different settings, the configuration should be globally accessible to ensure the configuration is accessible when needed. Other examples include: a single print spooler governing access to the printer resources, a single thread/connection pool handing out connections or threads to clients that request them, etc.

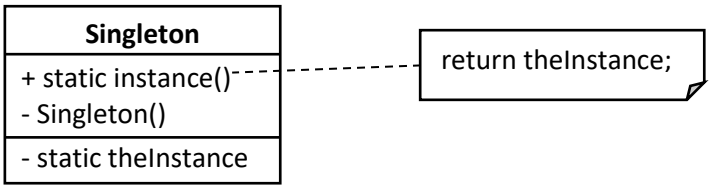
When to Use

The Singleton pattern should be applied to a class when:

1. Exactly one instance of the class must exist
2. A well-known access point to the instance is required

In general, the extensibility of a Singleton through subclassing should also be considered. It will not be required for this assignment; however, it is important to note as a common criticism of the Singleton pattern is that it becomes tied to implementation and makes testing difficult due to the inability to replace the singleton instance with a mock (i.e., a test specific class that is not a complete implementation).

General Structure



A simple representation of the Singleton pattern includes only a `Singleton` class, i.e., the class that should be a Singleton, containing: a static member function `instance()` that returns the unique instance of the class, a hidden constructor, and a static data member to hold the unique instance of the class.

Note the '+' indicates public accessibility and '-' indicates private accessibility. If considering extensibility of the Singleton through subclassing, the constructor should be protected rather than private.

[1] Scanned PDFs of the relevant sections are available from the course website as the library has limited copies of the book.

Implementation Hints

In C++, the compiler may generate some constructors and operators for you. Ensure you prevent the Singleton from accidentally being copied or assigned in ways you do not intend by *deleting* any automatically generated members that you do not need.

In C++, the static data member can be implemented in two different ways: within the scope of the class or within the scope of the function. Consider the consequences of each approach and justify your implementation choice in your comments.

Builder

The purpose of the Builder pattern is to separate the representation of a complex object from the process used to construct it. Further, such separation allows different representations to be constructed from the same process. For example, a Parser using a Builder to construct a parse tree in different systems: the recognition of the syntax is handled by the parser, which calls the builder to create the appropriate parse nodes and retrieves the final parse tree from the Builder once the parsing is complete. The different representations may or may not be related, for example, an executable parse tree of the parsed syntax, a parse tree for another language (such as in code transformation), or a composition of widgets for visualising the text processed by the Parser.

An important aspect of the Builder pattern that differentiates it from other creational patterns is that it supports the creation of complex objects in a step-by-step process, rather than all-at-once.

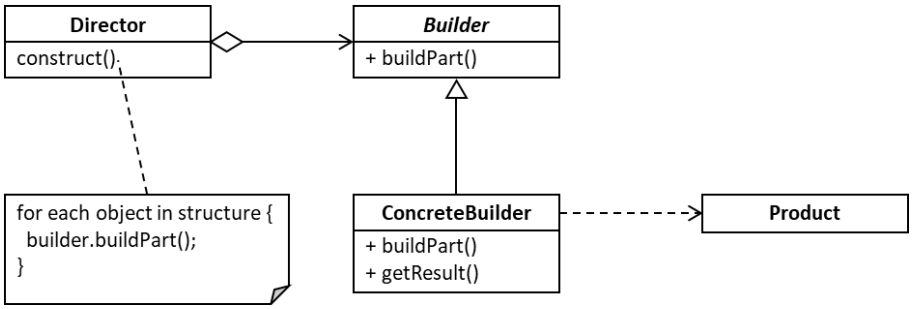
When to Use

The Builder pattern should be applied when:

- the construction process is, or should be, independent of the object, its parts, and the way that it is assembled
- the construction process should allow for different representations to be created.

Note that there does not always have to be multiple representations, only that multiple representations are possible. One of the motivations for good design is to support future change more easily.

General Structure



The Builder pattern contains several interacting classes, including:

- the `Director`, which represents the entity that requires the object to be constructed and 'directs' the `Builder` to construct it;
- the `Builder`, which specifies an abstract interface for constructing any `Product` object, this may include many different member functions for constructing different types of parts;
- one or more `ConcreteBuilder`s, which create specific `Product`s from its parts by implementing the abstract `Builder` interface and allows the `Product` to be retrieved once complete as it maintains the representation internally during construction;

- the `Product` class, which represents the complex object being created by a `ConcreteBuilder` as well as its parts. **Note:** this does not mean the classes for the parts are encapsulated by the `Product` class, only that the diagram is simplified to not explicitly illustrate any of the `Product`'s parts.

Implementation Hints

If the representations being constructed by the different `ConcreteBuilder`s are related, the abstract `Builder` may declare an appropriate `getResult()` member function as part of its interface.

The `ConcreteBuilder` object is not required to be supplied to the `Director` via a constructor, it could also be a parameter to a member function.

Decorator

The Decorator pattern is used to dynamically associate additional functionality or responsibilities to an object, rather than a class. The common exemplar is the addition of a border to GUI widgets: the widgets themselves may or may not have a border, scroll bar, etc., but can be wrapped in an object that understands how to draw a border around the widget, i.e., a decorator. This pattern is an example of favouring composition over inheritance, which results in greater flexibility. Consider using inheritance to add a border to a widget, displaying the widget would require a check for whether a border should be drawn then display the border followed by displaying the widget itself. Now consider that widgets can have a scroll-bar *and* a border, displaying the widget would have to check and display the border then check and display the scroll-bar. Most, if not all, widget classes would inherit this capability even though many instances of those widgets would not require it. By separating out these responsibilities into distinct classes that can be composed, the widget can worry about displaying itself, while the border decorator can focus on drawing a border and the scroll-bar decorator can focus on the scroll-bar. This has no impact on the calling code as the decorators conform to the same interface as widgets but delegate most function calls straight to their component, which may be another decorator. Apart from reuse, this pattern supports extensibility as new decorators can be added without affecting those already implemented.

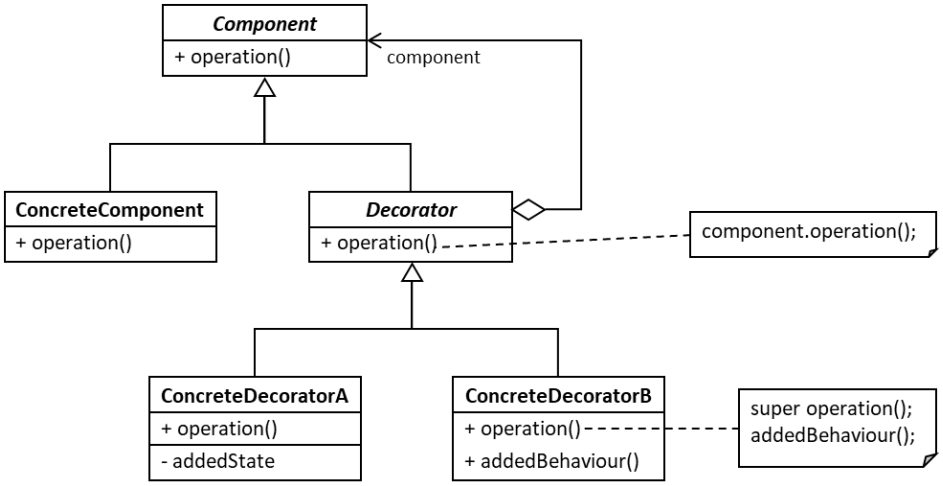
When to Use

The decorator pattern should be used when:

- responsibilities or functionality should be added to specific objects dynamically,
- such responsibilities can be removed dynamically, and
- the use of inheritance is impractical, e.g., supporting all combinations of responsibilities would become cumbersome.

Note: The decorator pattern is best applied when the shared base class is (or can be made) lightweight, with few data members, otherwise the overhead for having many objects may become a problem. This is in contrast to the strategy pattern (see next section), which can otherwise be used in similar situations.

General Structure



The Decorator pattern includes several classes with responsibilities as follows:

- an abstract `Component` that defines the common interface for objects that support dynamically added responsibilities;
- the `ConcreteComponent`, which represents the specific types of object that can have responsibilities attached;
- the abstract `Decorator` that maintains compatibility to the `Component` interface by referring to a `Component` object and delegating the function calls to the underlying object;
- the `ConcereteDecorator` classes, which add responsibilities to the `Component` object and may provide additional behaviour and/or state.

Implementation Hints

When implementing the decorator pattern, more than one operation of the component type can have their output modified by a decorator. For example, a border decorator might decorate the draw operation (to display the border) as well as the size operation (to add the border thickness to the reported size of the widget).

Strategy

To encapsulate a group of similar algorithms and allow them to be used interchangeably, the Strategy pattern should be used. Such an approach simplifies client code by separating the different algorithm variants into their own classes, rather than implementing them all in the one place, and allows an appropriate algorithm to be chosen dynamically based on the context. It can also allow behaviour to reused between objects of different types. A simple example of Strategy would be the formatting of dates as a string based on locale: the different formatting algorithms are implemented independently, and the locale object is configured with the desired algorithm. The strategy pattern can also be seen in libraries for Data Science where, for example, different clustering algorithms are implemented as classes with the same interface, so they can be used (for the most part) interchangeably. Another feature of the strategy pattern is that it allows the algorithm implementations to encapsulate internal state required to perform the operation without exposing it.

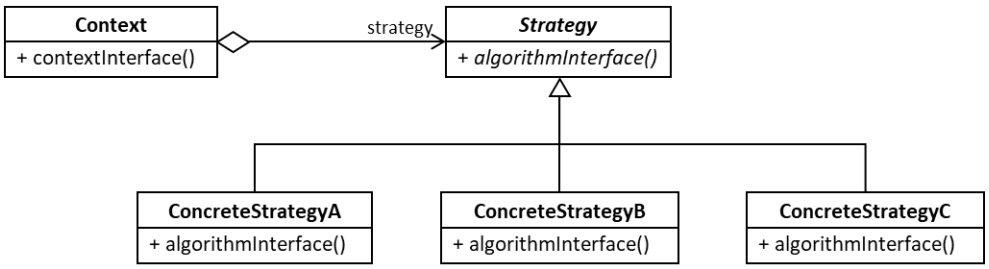
When to Use

There are several situations in which a Strategy pattern is applicable. The situations most relevant to this assignment are when:

- many related classes differ primarily in their behaviour, and
- multiple variants of an algorithm are necessary.

Note: the strategy pattern can be applied more efficiently to fat objects, i.e., objects with many data members, by being an extra data member themselves. As such, the strategy pattern changes behaviour from the inside-out whereas the decorator pattern changes behaviour from the outside-in.

General Structure



The classes and responsibilities in the Strategy pattern include:

- the abstract `Strategy` class, which defines the interface that all the algorithm variants support;
- the `ConcreteStrategy` classes that each define a variant of the algorithm conforming the interface defined by the abstract `Strategy` class;
- a `Context` class that references a `Strategy` object, which may be configured at creation or dynamically through an interface (e.g., `setAlgorithm(…)`), and executes the Strategy object when desired.

Implementation Hints

In C++, the function call operator '`()`' can be overloaded like other operators, which makes it easy to execute the strategy object as if it were a function. Overloading the function call operator will not be explicitly covered until closer to the submission deadline of this assignment and, therefore, is **not** required in your implementation.

The interface of the `Strategy` class may specify parameters such that all the information it expects to require is passed in. Alternatively, the `Strategy` interface may accept a reference to the `Context` class, so that the `Context` can pass itself to the `Strategy` object, allowing it to retrieve data from the `Context` object as required. Consider the consequences of each approach and justify your implementation choice in your comments.

Usually the `ConcreteStrategy` will be chosen by the client and passed to the `Context`; however, in this assignment the `ConcreteStrategy` may be predetermined in most instances.

Task Description

Roguelike Dungeon Crawler games are a genre of game that exhibit features typically attributed to the 1980s computer game Rogue[2]. In general, the goal of a Roguelike game is to clear a dungeon, level-by-level, until the final level is completed. Common features of a Roguelike game include:

- procedurally generated dungeons,
- randomised items and weapons,
- permanent death, and
- a variety of creatures to fight that get progressively more difficult deeper into the dungeon.

Within each level of a dungeon there typically exists distinct rooms where creatures are encountered and items are waiting to be looted.

In this assignment, you will design and implement a simple text-based Roguelike Dungeon Crawler game that fulfils the following specification. Unlike the original Rogue, text-based does not mean ASCII art but instead will be

[2] https://en.wikipedia.org/wiki/Rogue_(video_game)

driven by a text-based menu interface inspired by another genre of text-based adventure game originating from Multi-User Dungeon (MUD)[3]. Moreover, combat will be turn-based with only one opposing creature at a time.

The aim of the assignment is to allow you to practise creating an object-oriented design and implementing it in C++ using the features learnt so far, namely: class inheritance and polymorphism, dynamic memory allocation, and smart pointers. The focus of the assignment is on identifying and applying appropriate Design Patterns to help develop the game in such a way that components are easily reusable and extensible—remember the DRY (Do Not Repeat Yourself) principle.

To that end you will, first, need to ***develop a class design using UML class diagrams*** based on the requirements in the remainder of this specification. A simple and free UML editor that can be used for this assignment is UMLet (http://umlet.com/). The UML class diagram must include all classes, public data members, public member functions, protected data members and member functions (if any), and associations between classes. The diagram may optionally display private member functions, but private member variables should not be included.

For this assignment the game has been simplified to a certain extent, e.g., dungeon creation is not completely procedural. Moreover, some of the underlying elements (such as the basic menu interface code) will be provided to get you started. The provided code will be incomplete and may need to be modified to fit your class design. Parts of the provided code that must not be modified will be marked as such. You will then need to implement the rest of the game according to the specifications and your design.

A brief example of how the game will be played is provided below. The remainder of the section describes the requirements of the game in more detail.

```
> dungeon_crawler.exe
Welcome to Ashes of Software Development: Rite of Passage!
                  Developed by Matt Selway
           COMP 3023 Software Development with C++

*Press any key to continue*
> [Enter]
What would you like to do?
 (p)lay the game
 (q)uit
> p

You need to create a character... what is your name?
> Dungeon Crusher

Welcome Dungeon Crusher, you have *6* stat points to allocate.
A high Strength stat will boost your damage in combat.
How many points do you want to add to your Strength?
> 3
You have *3* stat points remaining.
A high Dexterity stat will increase your ability to dodge creature attacks.
How many points do you want to add to you Dexterity?
> 2
You have *1* stat point remaining.
A high Wisdom stat will boost the effectiveness of magical items.
How many points do you want to add to your Wisdom?
> 1

*** Character Summary ***
Dungeon Crusher
Strength:         4
Dexterity:        3
```

[3] https://en.wikipedia.org/wiki/MUD1

```
Wisdom:           2
Health:     50 / 50
Damage:     10 - 10
Dodge:      20%
Weapon:     you look down at your fists and shrug, "these will do"
Item:       you look into your backpack, emptiness looks back.
            if only you had something to put in it.

While roaming the country side you encounter a strange fork in the road.
To the left lies a dark cave, the foul stench of rotting flesh emanates from it.
To the right is a mysterious tower, a strange magical energy lights the path.
What would you like to do?
 Go left: create a (b)asic dungeon
 Go right: create a (m)agical dungeon
 Go (b)ack the way you came (return to main menu)
> l

You enter the dark cave and see...
A dark and empty chamber.
To the NORTH you see an opening to another chamber.
To the SOUTH you see the entrance by which you came in.
What would you like to do?
 Go (n)orth
 Go (b)ack the way you came
 View your (c)haracter stats
 Return to the main (m)enu
> m

After exploring *0* levels, you run out of the cave as quick as your legs can carry you.
As you emerge from the cave you are startled by the bright light and pause while your eyes
adjust.
What would like to do?
 (p)lay the game
 (q)uit
> q

*Are you sure you want to quit? (y/n) *
> y

Goodbye!
```

Note: the following conventions will be used when illustrating example output:

- a '>' at the beginning of a line indicates the command line prompt
- `bold orange text` indicates input entered by the user
- user input surrounded in square brackets '[…]' indicates a specific key press
- hitting `[Enter]` after other input is implied

Workplan

In order to complete this assignment, it is important that you take a planned, methodical approach to the assignment. You will need to think about how to break down the assignment into smaller chunks of functionality that can be implemented and tested one at a time. Working in an incremental manner, and testing as you progress, helps to limit the scope of errors that are introduced as you perform the implementation. Moreover, it allows you to focus on a single problem at a time as well as refactor and redesign the application in a controlled manner as you (re)discover requirements or issues that were not obvious at first.

This assignment should be completed in the following stages:

1. **Read** the assignment specification (more than once)

2. Ensure you **understand** the 4 design patterns
3. **Design** a class hierarchy using UML class diagrams that addresses the requirements and applies the 4 design patterns to the game appropriately. Note: the design does not have to be perfectly complete from the outset, you can revise it as you progress through the assignment. However, it is good to start by conceptualising your approach before diving into the implementation.
4. Implement **character generation**—refer to section *Player Character*
5. Implement the **basic dungeon**: walls and doors, no creatures nor items (standardised layout)—refer to section *Dungeon (Rooms, Walls, Doors)*
6. Implement **dungeon navigation**: at this point you should be able to play the game and transition between rooms and dungeon levels without combat—refer to sections *Dungeon (Rooms, Walls, Doors)* and *General Gameplay*
7. Add **items and item pickups**—refer to sections *Weapons, Items & Enchantments* and *General Gameplay*
8. Add **one creature** type and **combat** interactions: you should now be able to play the game with all general gameplay requirements/features complete (except second dungeon type)—refer to sections *Creatures* and *Combat*
9. Add remaining **creature types**—see section *Creatures*
10. Add **second dungeon type**—see section *Dungeon (Rooms, Walls, Doors)*

As you implement the functionality, you should tick, highlight, or otherwise mark the requirements that you have completed. This will help you keep track of your progress and ensure you have addressed all the requirements in your submission.

General Gameplay

The gameplay state will be controlled by a `Game` object. There can only ever be one game in play at one time.

The game allows a player character to explore a dungeon of your choice, room-by-room. Along the way they may encounter creatures that they will have to defeat before they can continue exploring (unless they can find a way around them). They will also come across items to loot, some of which may be weapons that will help them defeat the creatures they encounter. The last room in the dungeon typically has a more powerful monster that needs to be beaten for the exit to reveal itself but is not required.

When the game is first started, a welcome message will be displayed and the user will be given the option to play the game or quit from the main menu.

If the user chooses to quit, the game will prompt the user to confirm that they want to quit the game:

- If the user selects ‘yes’, the game will end.
- If the user selects ‘no’, the game will return to the main menu.

If the user chooses to play the game (from the main menu), the game will prompt the user to create a character. Refer to section *Player Character* for details on character creation.

After the user has created their character, the game will prompt the user for the type of dungeon they would like to explore. There must be two types of dungeon: a “basic” dungeon, and a “magical” dungeon. Refer to section *Dungeon (Rooms, Walls, Doors)* for details on the types of dungeon and their construction.

Once the user selects a dungeon type, the game must create the first dungeon level according to the selected type, place the character in the first room, display the room’s description, and prompt the user for the next action. The user can take various actions depending on the configuration of the current room, whether a creature is present, and the item(s) the character currently possesses. The following actions must be supported, as appropriate:

- Move the character to the next room: North, South, East, or West
- Move the character to the previous room (whether this is North, South, East, or West depends on the door the user "came through")
- Pick up an item/weapon
- Compare the weapon/item in the room with that currently held by the character
- Attack the creature
- Use an item
- Use the weapon's special ability
- Return to the main menu

In most cases, after a choice is made, the action menu is displayed again.

If the user chooses to move to another room, the current room will be updated to the room that was selected, the new room's description will be displayed, and the user prompted for the next action.

If a creature is present in a room, the user cannot move to another room (other than the one the user came from) until ***after*** the creature is defeated (refer to section *Combat*).

If a weapon or item is present in a room, the user can choose to pick it up. If they do so, the character's current weapon/item is replaced, and the ***previous weapon/item disappears***. To avoid accidental replacement of a weapon/item by the user, the game must warn the user that their weapon/item will be replaced and prompt for confirmation.

If the user chooses to compare the character's weapon/item with that of the room, the description and stats of both items will be displayed. Whether the weapon/item is the currently held weapon/item or that of the room must be clearly indicated.

If a creature and an item ***both*** exist in a room, the item ***cannot*** be picked up until ***after*** the creature is defeated (refer to section *Combat*).

If the user chooses to use an item, the item's effect will be applied and the ***item will be removed*** from the character.

If the user chooses to use the weapon's special ability, the ability's effect will be applied (the weapon will ***not be removed***).

If the user chooses to return the main menu, the ***current dungeon and character will no longer be playable***, i.e., a new character will have to be created. Therefore, the user must be warned that progress will be lost and prompted for confirmation. If the user confirms their intent to return to the main menu, the number of dungeon levels successfully completed must be displayed followed by the main menu.

Once the user reaches the final room of the dungeon level (and defeated any obstructions) the door to the next level will be revealed. If the user chooses to go through that door (via the appropriate direction, North, South, East, or West), the game will return the user to the main menu. If the user chooses to play the game (again), a new dungeon level must be built (of the currently selected dungeon type) but the ***existing character*** will be used.

If the character's health stat drops to zero or below, the game ends: the game will display a game over message, the number of dungeon levels that were successfully completed, and return to the main menu. If the user wishes to play again, they will ***need to create a new character***.

General Requirements

At any time a valid character exists, the user must be able to view their character details (stats, items, etc.). After viewing the character details, the user will be able to view the weapon specific details, the item specific details, or return to the main menu.

All user inputs must be validated. If an invalid input is given, a warning must be displayed to the user and the menu options redisplayed.

All classes must have appropriate constructors and destructors.

All objects that can be described must be able to be output to a stream using the stream output operator '`<<`'.

Player Character

The player character is represented by the `Character` class. This class will maintain the state regarding the player's character as well as define behaviours appropriate to the character (i.e., update the game state as appropriate based on the character's actions).

A character has a name (provided by the player), a combination of static properties and dynamic properties that may affect the gameplay and the character's actions, a weapon, and an item. When the player chooses the option, the character's stats must be displayed.

The static properties are defined at character creation and include:

- Strength—allowable values 1-6, each value > 1 contributes to the damage of the character
- Dexterity—allowable values 1-6, each value > 1 contributes to the dodge chance of the character
- Wisdom—allowable values 1-6, each value > 1 contributes to the effects of magical items

The dynamic properties are derived from various sources and must be updated as appropriate. They include:

- Health points—starts at 50 and is reduced by taking damage from creatures (whole numbers only). Once it reaches zero (or less) the game is over. Various items and special abilities may increase the amount of health, but not above the original value.
- Damage—calculated from the current weapon and Strength. The strength bonus is equal to: `2 * (Strength - 1)`. That is, each point of Strength above 1 adds twice as much to the damage.
- Dodge chance—percentage, used to determine whether the character will get hit by a creature when it attacks. 100% means the character will dodge every attack. Calculated as: `100 / 5 * (Dexterity - 1)` That is, each Dexterity point above 1 adds 20% to the dodge chance.

Character Creation

Before the game can be played, a character must be created by the user. Creating a character requires the user to specify some parameters of their character. These include the name of the character and the allocation of points to the static properties: Strength, Dexterity, and Wisdom.

The game will first prompt the user to enter the character's name.

The game will then prompt the user to allocate the points the three static properties. The rules for allocating stats are as follows:

- The user will start with 6 points to allocate
- The user will select the number of points to allocate to each property in turn: 1) Strength, 2) Dexterity, 3) Wisdom.

- The selected number of points will be added to the base value of 1; the resulting value will be assigned to that property of the `Character` object.
- The selected number of points cannot exceed 5, so that the total value of the property does not increase beyond the maximum of 6 points.
- Zero points can be selected if desired, in which case the value of the property will remain the base value.
- After the user selects a valid number of points, the selected number will be subtracted from the number of available points.
- When selecting the number of points to assign to subsequent properties, the current number of available points cannot be exceeded.
- After the user has selected the number of points to assign to each property there should be no more points available. If there are remaining points to be allocated, the game must prompt the user to confirm that they did not want to assign all the points. If the user indicates that they wish to continue assigning points, the game will go back to assigning points to each property in turn with the current available points.
- The game must not allow the value of a static property to exceed the maximum value of 6.

After the user has allocated all points, or indicated that they do not want to assign the remaining points, the character description and stats must be displayed to the user. This is the same description that would be displayed if the user chose to display their character details from the menu.

When a character is first created it has no consumable item and has the 'Fists' weapon type (refer to section *Weapons, Items & Enchantments*).

Character Details Display

When displaying the character details, the following must be included:

- The character's name
- Strength
- Dexterity
- Wisdom
- Health: current and maximum
- Damage, calculated as described above
- Dodge chance, calculated as described above
- The character's weapon, short description only (refer to section *Weapons, Items & Enchantments*)
- The character's item, if present, short description only (refer to section *Weapons, Items & Enchantments*)

After displaying the character details, the menu will allow the user to view the complete weapon details, view the complete item details, or return to the main menu. These options will not be available when the character details are being displayed for the first time immediately following character creation.

Dungeon (Rooms, Walls, Doors)

A dungeon level consists of a connected set of rooms and are to be represented by the `Dungeon` class. Each room of a dungeon (represented by the `Room` class) must be identifiable by a number (`integer`) so that they can be retrieved from the dungeon object using the numeric identifier. Each room has 4 directions: North, South, East, and West. In each of these directions, there may be a wall (the `Wall` class) or a door (the `Door` class). Doors are connected to adjacent rooms via a door in the opposing direction. For example, if a room has a door to the north, the northern room has a door to the south.

Doors are connected to each other to allow movement between rooms in both directions. There are two exceptions: the entrance to the dungeon level, and the exit. When the user chooses to go through the dungeon

entrance, they will be returned to the main menu as if they chose the option to return to the main menu, that is they will lose their progress and the dungeon and character will have to be recreated according to the *General Gameplay* section. When the user chooses to go through the exit door, they will be returned to the main menu ready to continue to the next dungeon level as described in the *General Gameplay* section.

`Door` objects must be connected to one another using bare pointers in C++.

Each room may contain a creature, a weapon/item, or both. If both are present in the room, the item cannot be picked up by the user and will "appear" after the user defeats the creature.

When the user enters a room, a description of that room must be provided to the user. The description includes what the room "looks like" (this is up to you, or you can use the text of the examples) and what the character "sees", that is, where the doors are located, whether a creature or item is present. Descriptions of walls may be included.

When a dungeon level is being created, various elements must be selected randomly including: the type of room, the type of creature (if present), and the type of item (if present). The size of the dungeon (i.e., the number of rooms) and the connections between rooms ***will not*** be randomly generated. When making a random selection, a simple even weighting between choices can be used. Moreover, it is **not required** to do any complex registration of types: a simple switch statement will be acceptable.

Dungeon Types

You must implement two dungeon types: a basic dungeon (`BasicDungeon` class), and a magical dungeon (`MagicalDungeon` class). Each dungeon type can contain different types of rooms, walls, doors, creatures, and/or items. Refer to sections *Creatures* and *Weapons, Items & Enchantments* for details of the creatures and items, respectively.

The basic dungeon is built out of the following:

- Rock Chamber—A dark and empty chamber (`Room` type)
- Quartz Chamber—The chamber glitters like a thousand stars in the torchlight (`Room` type)
- Basic Wall—Out of the darkness looms a jagged formation of rock: you cannot go that way (`Wall` type)
- Open Doorway—… you see an opening to another chamber (`Door` type)
- Short Sword (`Weapon` type)
- Battle Axe (`Weapon` type)
- Health Potion (`Item` type)
- Acid Flask (`Item` type)
- Goblin (`Creature` type)
- Werewolf (`Creature` type)

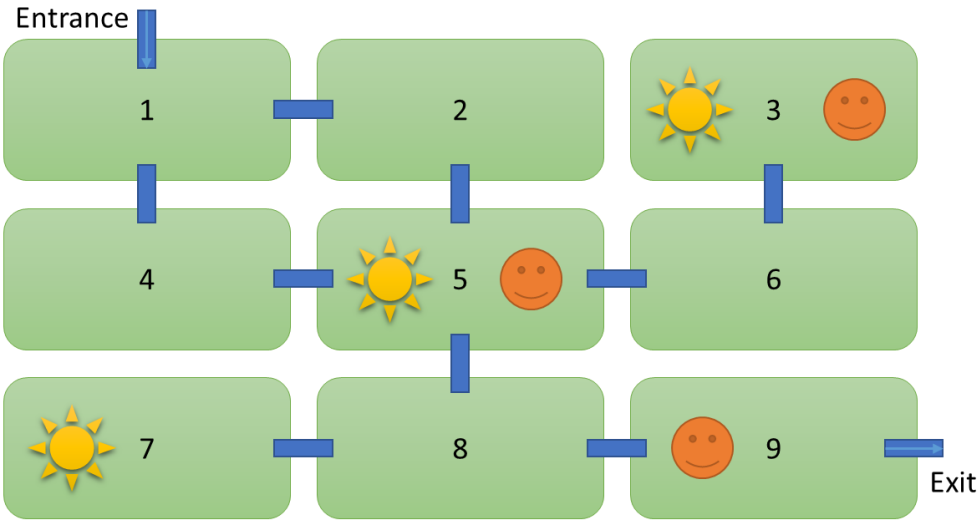
The magical dungeon is built out of the following:

- Enchanted Library—The smell of musty pages and the tingle of magic fills the air (`Room` type)
- Alchemists Laboratory—Your ears are filled with the sounds of numerous elixirs bubbling away over open flames (`Room` type)
- Magic Wall—A strange barrier blocks your path, almost invisible to the naked eye (`Wall` type)
- Open Doorway—… you see an opening to another chamber (`Door` type)
- Wizard's Staff (`Weapon` Type)
- Magic Wand (`Weapon` Type)
- Health Potion (`Item` Type)
- Teleporter (`Item` Type)

- Goblin (`Creature` Type)
- Evil Wizard (`Creature` Type)

The Standard Dungeon Level

Dungeon levels would normally be randomly generated; however, to simplify the assignment and make it easier to test and mark, you are ***not required*** to implement randomly generated dungeon levels. However, the design and implementation should be structured such that this change could be made in the future. Some elements of the dungeon are still randomised, for example, the type of each room, the type of creature (if present), and the type of item (if present). The structure of the (first) dungeon level must conform to the standard structure defined below. There is no constraint on the structure of dungeon levels after the first.



In the layout diagram the following symbols are used:

- Large green rectangles represent rooms with the numbers indicating their ID
- Small blue bars joining rooms represent connected pairs of doors
- Entrance and Exit are labelled and indicated by a directional arrow
- Orange smiley faces represent creatures
- Yellow suns represent consumable items or weapons

Creatures

Dungeons, specifically rooms, can be inhabited by numerous foul creatures. There can be ***at most one creature per room*** and a creature ***never leaves the room*** in which it is created. All creatures must derive from the class `Creature`. When the user encounters a creature in a room they must defeat it in combat before moving on or return the way they came and attempt to find a way around it. Each type of creature has different stats, but otherwise has the same types of stats as the player's character. These stats are:

- Strength
- Dexterity
- Wisdom
- Health
- Dodge chance

The bonuses provided by the 3 core stats (Strength, Dexterity, and Wisdom) are calculated in the same manner as for player character's and are then halved. If a creature is a *boss* (i.e., it is in the room with the exit door) then it does not receive the penalty.

In contrast to the character, creatures have a long description in addition to their name (which may just be their type).

In combat, creatures may `attack` the character ***with various types of attack***. The different attack types will have the damage calculated differently. Some creatures may attack with a weapon, like the player, and others may not. Different creatures may share the same types of attack. A single creature may attack with multiple attack types or the same attack type multiple times. The different attacks must also have different descriptions that will be displayed to the user when the creature attacks the character.

You must define the following 3 creature types:

Goblin

A small nimble creature who favours the short sword.

Stat	Value
Strength	2
Dexterity	4
Wisdom	1
Maximum Health	20
Damage	Weapon damage (+1[4] from Strength)
Dodge chance	30% (calculated from Dex.)
Weapon	Short Sword

Goblins make their attacks with a weapon. They can attack only once per round.

Werewolf

Glowing red eyes peer out from behind a long nose and sharp teeth.

Stat	Value
Strength	5
Dexterity	2
Wisdom	1
Maximum Health	30
Damage	2-7 x 2 (claw attacks) 5-10 (bite attack) (each attack: +4[5] from Strength)
Dodge chance	10% (calculated)
Weapon	None

Werewolves have complex attack behaviour. They have three possible attacks: two claws and a bite. Each round the Werewolf attacks it may attack with only 1, 2, or all 3 of its attacks. There is a ***60%*** chance that it will ***attack with only 1*** attack, a ***30%*** chance that it will attack with ***2***, and a ***10%*** that it will attack with ***all 3*** of its attacks in a single round. Exactly which attack it uses is not specified as long as it does not exceed the number of attacks for that type, i.e., a Werewolf cannot do its bite attack more than once per round.

Evil Wizard

Cackling over an old tome, this wizard seems insanely evil; or maybe just insane? it is hard to tell

Stat	Value
Strength	1

[4] Non-boss creature bonus. Boss creature bonus will be twice as much.
[5] Non-boss creature bonus. Boss creature bonus will be twice as much.

Dexterity	2
Wisdom	5
Maximum Health	20
Damage	Weapon damage or Weapon special ability (+ 2^6 from Wisdom)
Dodge chance	10% (calculated)
Weapon	Wizard's Staff

Evil Wizards attack using a Wizard's staff once per round. However, since the Wizard's Staff has two modes of attack (a standard attack and its special ability) the Evil Wizard must choose whether to attack normally or use the staff's special ability.

Weapons, Items & Enchantments

The game allows characters to carry weapons and consumable items (one of each), which they use in combat or at any time to affect their environment. In addition, weapons can be ***enchanted*** to give them extra bonuses or special effects for the character to use. Special effects may have an arbitrary effect on the game state. Moreover, creatures that have weapons can attack with them or use their special ability (if they have one).

Weapons (represented by the `Weapon` class) have a name, a short description (used on the character details display), a long description (used on the weapon specific display), and a damage stat (minimum and maximum value). Their primary use is for attacking a creature (or the character if owned by a creature) during combat. Some weapons may also provide a special ability, which can be used by the owning character or creature as many times as they like. When displaying the details of a weapon (either when viewing the character details or when comparing two weapons) the display must include:

- name
- long description
- damage range (min. and max. values)
- description of the special ability, if present

Weapons can be ***enchanted***, which add bonuses and/or special abilities to the weapon on which they are placed. Each weapon can have up to two enchantments. Enchantments change the reported name of the item and can be either 'prefix' enchantments—add a term (or terms) to the beginning of the name—or 'suffix' enchantments—add a term or terms to the end of the name. Only one of each type may enchant a weapon at the same time, i.e., if there are two enchantments on a weapon then one will be a prefix and one will be a suffix. For example, the 'Flaming short sword of healing' is a short sword weapon with a prefix enchantment named 'flaming' and a suffix enchantment named 'of healing'. Enchantments on a weapon may change the standard damage of the weapon when it is used to attack, or it may add a special ability to the weapon. Special abilities added by an enchantment follow the same rules as special abilities directly of the weapon. When a weapon with an enchantment has its details displayed, the display must include the additional details of the enchantment(s). **Note:** you do not have to worry about both an enchantment and the weapon providing a special ability, i.e., you do not need to differentiate with multiple menu items to use each special ability individually.

Consumable items (represented by the `ConsumableItem` class) have a name, short description, long description, apply an effect when used, and are removed from the character when used, i.e., they can be used only once. Consumable items can have an effect that manipulates any aspect of the game state. When displaying the details of a consumable item (either when viewing the character details or when comparing two items) the display must include:

[6] Non-boss creature bonus. Boss creature bonus will be twice as much.

- name
- long description

You must define the following weapon, enchantment, and item types:

Fists (Weapon)

Property	Value
short descr.	you look down at your fists and shrug, "these will do"
long descr.	Fists are the weapon of choice for someone who has nothing else.
damage	4 - 4
special ability	None

Short Sword (Weapon)

Property	Value
short descr.	a sharp and pointy instrument, good for stabbing
long descr.	Not very large, but with a sharp point. Short swords are designed more for stabbing than for slicing. The hilt is surprisingly ornate for such an inconspicuous weapon.
damage	5 – 10
special ability	None

Battle Axe (Weapon)

Property	Value
short descr.	heavy, but effective
long descr.	A large and heavy weapon. The battle axe must be wielded with two hands but even then you are almost as likely to cut off your own limbs as those of an enemy.
damage	10 – 15
special ability	None

Wizard's Staff (Weapon)

Property	Value
short descr.	it would break if you leant on it, but it shoots fireballs so that's something
long descr.	Not a very sturdy staff, but the swirl of magical fire around its top belies a magical secret: it shoots fireballs!
damage	1 – 2
special ability	Fireball, deals 10 - 20 damage to the opponent (plus the bonus from the creature or character's Wisdom stat)

Magic Wand (Weapon)

Property	Value
short descr.	birch with angel's feather core and rubberised leather grip
long descr.	Apparently, there is no other wand like this one in existence. The angel's feather at its core allows the bearer to perform unbelievable feats of healing.
damage	5 – 10
special ability	Healing: returns character to full health.

Health Potion (Consumable Item)

Property	Value
short descr.	a cloudy green fluid in a glass vial
long descr.	While it does not look appetising, drinking this potion is the ultimate boost to your immune system: kale really is that good for you.
effect	Healing: Restore 25 health points

Acid Flask (Consumable Item)

Property	Value
short descr.	the label reads "danger: corrosive substance!"
long descr.	Spilling this potion could really hurt. It is probably best if you keep the stopper in at all times.
effect	Deals damage to an the opponent: 10 damage

Teleporter (Consumable Item)

Property	Value
short descr.	a small device with a big red button, I wonder what it does
long descr.	Inspecting the device closely reveals nothing about its origin nor function. But the big red button is very tempting…
effect	Moves the character to a random room in the dungeon level

Flaming (Enchantment)

Property	Value
prefix/suffix	prefix
extra descr.	Holding it feels warm to the touch and sparks leap out when it makes contact with something.
effect	Deals an extra 5 damage when the weapon is used to attack.
applies to	Short Sword, Battle Axe, Wizard's Staff, Magic Wand

Electrified (Enchantment)

Property	Value
prefix/suffix	prefix
extra descr.	The air crackles around it making the hairs on your arm stand on end.
effect	Deals an extra 5 damage when the weapon is used to attack.
applies to	Short Sword, Battle Axe, Wizard's Staff, Magic Wand

of Healing (Enchantment)

Property	Value
prefix/suffix	suffix
extra descr.	Just being near it makes you feel all warm and fuzzy inside.
effect	Special ability: heals the owner for 5 health points.
applies to	Short Sword, Battle Axe

of Vampirism

Property	Value
prefix/suffix	suffix
extra descr.	Occasionally drops of blood appear on the surface but you are unsure from whence they came.
effect	Half of the damage dealt during an attack is given back to the owner as an increase in health points.
applies to	Short Sword, Battle Axe, Wizard's Staff, Magic Wand

Combat

Combat between a character and a creature will be performed in rounds. Each round the user will choose their action and then the creature will perform an attack according to their own specific rules and their available attack type(s). The user will go first and can choose one of the following options:

1. Attack—will perform an attack using the character's weapon dealing damage to the creature if it does not dodge the attack
2. Use item (if available)—the currently held item's effect will be applied and the item removed from the character

3. Use special ability (if available)—the weapon's special ability effect will be applied, if it is a damaging ability the opposing creature cannot dodge it (in contrast to a standard attack)
4. Return the way you came—will leave the room and return the character to the room from which they came (the creature must not follow). The user cannot pass through any door other than the one by which they came while a creature is present in a room.

After choosing any of first three options, the creature will perform its attack. If the user chooses to return to the previous room the creature does not get an attack.

Each time an action is performed (either from the user or the creature) an appropriate message and the action's description will be displayed to the user.

If an attack succeeds—i.e., if the target character or creature does not dodge it—the target's health is reduced by the damage amount as determined by the weapon's damage properties and the bonus provided by the strength stat of the character or creature performing the attack. Weapons have a damage range, so the actual damage value of any given attack will be determined by obtaining a random value within the range of the weapon's damage properties (inclusive).

Whether a creature or character dodges an attack is determined randomly based on their percentage *Dodge Chance* stat. This can be determined by representing the *Dodge Chance* as a real number (`float` or `double`) and generating a random real number between 0 and 1: if the randomly generated number is less than the *Dodge Chance*, the character or creature dodges the attack.

A special ability that does damage will reduce the health of the target by the amount calculated: if the item specified a damage range for the special ability, the actual damage value of the ability will be determined by obtaining a random value within the range (inclusive).

When the creature's health reaches zero or below, a "win" message must be displayed to the user and the creature must be removed from the room.

If the user's character reaches zero or less health points, the user is returned to the main menu and the character is destroyed as per the general gameplay rules specified in section *General Gameplay*.

Menu

The basic menu structure will be provided for you, but you will need to implement the interactions with the `Game` and the transitions between menus for the different contexts, for example, the main menu to the action menu while playing the game. You are quite free to implement the menu as you desire, that is, the format and text descriptions of the menu ***does not have to exactly match*** the output examples provided. However, the ***values*** entered to perform specific functions ***must match*** the following specification to ensure consistency and support testing/marking. Note: it should not matter if the typed character is uppercase or lowercase.

If in doubt, refer to the provided 'input script', which is a series of text input which can be fed automatically to the application to test the game, and the reference implementation executable.

Command	*Character*	*Command*	*Character*	*Command*	*Character*
Main Menu		**Dungeon Selection Menu**		**Action Menu**	
Play game	p	Basic Dungeon	b	Go North	n
Character Details	c	Magical Dungeon	m	Go East	e
Quit	q	Go back/Return	r	Go South	s
Character Details Menu		**Combat Menu**		Go West	w
Weapon Details	w	Go back	b	Pick-up Item	p
Item Details	i	Attack	a	Compare Items	o
Previous Menu	m	Use Item	i	Use Item	i
		Use Special Ability	l	Use Special Ability	l
		Character Details	c	Character Details	c
		Main Menu	m	Main Menu	m

Documentation

Your code must be documented appropriately using Doxygen comments. Comment blocks ***must be used in your header*** file to document your ***classes and class members***. Private members should also be documented when it is not obvious for what they are used—but first check if the code can be improved as clean code is better than comments. Use ***comments sparingly in source files***, document blocks of code (switch statements, if else groups, loops, etc.) rather than individual statements: do not comment individual statements, unless the outcome of the statement is not obvious. Optionally, you can add comments to 'main.cpp' to document the 'main page'.

Code Style

You must write you code conforming to the code style set for this course.
See http://codetips.dpwlabs.com/style-guide

Version Control

You must use version control to keep track of your progress during implementation, preferably using git. You should perform regular commits as you implement features and fix errors. Your commit comments should reflect the ***context*** of the changes that were made in each commit—a fellow developer can always diff the contents to see *exactly* what changed but that does not provide the context. You should try to ensure each commit comment contains a short subject line.

Your submission will comprise your entire version control repository, so it should be specific to the assignment. For example, if you use git, you must create the git repository in the root directory for your assignment. Do not commit your assignment to a repository created in a parent folder. Your repository must contain all source files required to compile and run your program (including a Doxygen config file) but must not include any generated or extraneous files.

You ***do not*** have to use an online version control repository such as GitHub. If you do use an online repository, ensure that the assignment is ***private***, i.e., ***cannot be seen by people other than yourself***.

Implementation Rules and Hints

Your initial UML class design may be incomplete or incorrect. Your first design should attempt to cover the major requirements, in particular the identification of the 4 design patterns. Do not worry if you need to revise and refactor the design as you progress through the assignment.

It is a good idea to commit the source file (but not the PNG or PDF) of your UML class design to version control. This will allow you to track the evolution of your design as you progress through the assignment.

It is recommended that you test individual components as you go, rather than trying to test the whole application through the menu-driven interface. Writing code to test specific elements will speed up development as you will not need to constantly enter data through the menu interface.

You may change the descriptions of items, creatures, actions, etc., if you like.

Be sure to check for null pointers before attempting to use an object through a pointer and reset pointers to null where appropriate.

The game will need to keep track of (at least) the current room, the character, and the previous room/door.

Ensure you have considered similarity between classes so that you can achieve reuse through inheritance where appropriate. Also, remember to *program to an interface not and an implementation*. For example, if you have related classes but only some have particular data members and you want to ensure substitutability, you can define an interface on a base class using `virtual` functions that the derived classes then override appropriately. The derived classes that do not have those particular data members may then rely on a suitable default implementation that does not use those data members.

Consider when and where it is appropriate to use smart pointers vs. bare pointers and the effect this choice may have on constructors, destructors, etc. In most cases the choice is up to you with the exception of the connection between two doors, ***doors must be connected via bare pointers***.

Smart Pointers – the `Dungeon` class uses the smart pointer `std::shared_ptr<Room>` to store and retrieve the rooms (among other things). You must use the `std::make_shared<`*DerivedType*`>()` template function to create the rooms to pass to the function `Dungeon::addRoom`. The below code snippet is an example of adding a room to a dungeon:

```
Dungeon &d = Dungeon{};
d.addRoom(std::make_shared<RockChamber>(/* constructor arguments go here */));
```

The `Dungeon` and `Room` classes may not be the only occurrence of smart pointers in your implementation.

Casts – in your code, casts should only need to be performed between a base class pointer and a derived class. When doing so, a dynamic cast should be performed (using `dynamic_pointer_cast<`*Type*`>`). For example, to cast an `Item` pointer to a `Weapon` pointer you would use the following code snippet:

```
std::shared_ptr<Item> item = …; // Wherever the value comes from
std::shared_ptr<Weapon> weapon = std::dynamic_pointer_cast<Weapon>(item);
```

Type defines – if you know what you are doing, you may define type aliases for the shared pointer types for use in the `MenuInterface` and your classes.

Submission Details

You must submit your complete Qt Creator project folder inside a ***single zip*** file. The zip file must include ***all source files*** required to compile and run your program (including a ***Doxygen config file***) as well as your ***version control directory***: e.g., if using git you must ensure the '.git' folder is present in the zip file. It must not include any generated or extraneous files such as the '*.pro.user' file—the UML image file is the one exception to the no generated files rule. Place your UML class diagram in your zip file. Your diagram ***must*** be submitted in ***PNG*** or ***PDF*** format.

Once you have created the zip file according to the specifications, you are required to upload it to the ***Assessment Item 2: Assignment*** submission via ***LearnOnline***. The deadline for submission is **14 September 2018 11:59 PM**. After which time the submission will be considered late and attract a penalty as detailed in the course outline.

Marking Criteria

Your assignment will be marked both automatically and manually; automatically using unit testing, shell scripts and static code analysis tools. Your code will be inspected for style – remember consistency is the primary goal of all style guides, the easier it is to understand your code, the easier it is to allocate marks.
Note: the criteria with negative marks are penalties that will be applied if the submission does not meet the expected criteria. They are structured such that an otherwise perfect submission will be unable to receive an HD if and only if the *full* penalties are applied.

Criteria	**Mark**
Correct identification and use of the design patterns:	—
• Singleton	5
• Builder	10
• Decorator	10
• Strategy	10
Correct use of inheritance	5
Correct separation of code in header and source files: declaration vs. definition	5
Correct implementation of classes: constructors/destructors, data members, member functions, public/private accessibility	10
Correct use of (bare) pointers and smart pointers	5
Correct implementation of functional requirements:	—
• General gameplay	10
• Player Character	5
• Dungeon Creation	5
• Creatures	5
• Weapons, Items, & Enchantments	5
• Combat	5
• Menu Interface	5
Inappropriate use of version control	(-5)
Source code *does not* follow style guide	(-5)
Inadequate/inappropriate use of comments (Doxygen)	(-5)
Total Possible Marks	**100**

Academic Misconduct

This is an individual assignment: your submitted files will be checked against other student's code, and other sources, for instances of plagiarism.

Students are reminded that they should be aware of the academic misconduct guidelines available from the University of South Australia website.

Deliberate academic misconduct such as plagiarism is subject to penalties. Information about Academic integrity can be found in Section 9 of the Assessment policies and procedures manual at:

http://www.unisa.edu.au/policies/manual/