Curtin University — Discipline of Computing

Object Oriented Software Engineering (COMP2003)

Semester 1, 2020

Assignment

Due: Thursday 21 May, 23:59 (UTC+8)

Weight: 40% of the unit mark.

This assignment will give you experience in taking complex software requirements and creatively using object orientation to implement them.

The assignment specification is divided into two parts:

- Sections 1–5 describe how the OOSE assignment works in general: what your responsibilities are, how to submit your work, and how the marking is to be conducted.
- The Appendix describes the actual requirements of the system that you must design and implement.

1 Your Task

Design and implement the system described in Appendix A, providing the following:

(a) Your design, expressed in UML, containing all significant classes, class relationships, and significant methods and fields.

(b) Your complete, well-documented code, in Java, C#, Python, or C++ (your choice). Do not use any third-party code without approval, in writing, from the lecturer.

(c) A `README.txt` file that explains how to compile and run your code.

(d) A 2–3 page written discussion of the following:

(i) How your design works, including: what design patterns you've used, how you have adapted them to the situation at hand, what they accomplish, and how they impact upon coupling, cohesion, reuse, and the future extensibility of your application.

(ii) Two plausible alternative design choices (significant ones), and their potential advantages and disadvantages.

2 Submission

Submit your entire assignment electronically, via Blackboard, before the deadline.

Submit one .zip or .tar.gz file containing:

Your declaration of originality – whether scanned or photographed or filled-in electronically, but in any case *complete*.
Your source code – your .java, .py or .cpp/.h files (and build.xml if you have one).
Your README.txt
Your UML – one .pdf, .png or .jpg file.
Everything else – exactly one .pdf file.

Avoid the .rar, .zipx, .7z formats, or any other non-standard archive/compression format. Do not assume the marker has extraction tools for anything but ordinary .zip and .tar.gz files. If the marker cannot unpack your work, then it won't be marked!

You are responsible for ensuring that your submission is correct and not corrupted. Once you have submitted, you are *very strongly advised* to download your own submission and thoroughly check that it is intact.

You may make multiple submissions. Only your last one will be marked.

3 Viva Voce

You *may* be required to participate in an oral interview ("viva voce", or just "viva") regarding your submission, after you have submitted. The purpose of this will be academic integrity, rather than as part of the marking process.

It is likely (though not certain) that only a sample of students will selected. The selection process will not be published in advance, but selection will be done in some academically-defensible manner.

4 Marking Criteria

Your submission will be marked out of 20 marks, including 13 marks that derive directly from the code you will submit, and 7 marks from associated design documentation. However, note that the validity of your design documentation is also highly dependent on the code.

Some marks are also directly conditional upon the following:

- Actually implementing the required functionality.
- Good commenting, formatting, and avoidance of global variables.

2 marks – Clear division of responsibilities (also requiring that the above conditions be met).

Avoid having a "god class". Be very clear about what responsibilities each class, interface and package (or namespace) has. Have meaningful packages/namespaces.

1 mark – Minimal redundancy (also requiring that the above conditions be met).

Avoid repetition, and otherwise unnecessarily long code.

2 marks – Testability (also requiring that the above conditions be met).

Ensure you understand what testability is. Don't confuse it with simply "doing testing".

To help make your code testable, get dependency injection happening. This may be assisted by careful use of factories, and may be *hindered* by use of singletons (unless you're very careful).

2 marks – Error handling.

We expect your application to handle errors gracefully and comprehensively. We expect you to anticipate all the different things that might go wrong at runtime, and deal with them properly.

Define your own exception classes as appropriate. Don't misuse exceptions. Don't catch high-level exception types. When handling exceptions, be careful to respect the division of responsibilities between classes and packages.

6 marks – Extensibility and decoupling through polymorphism.

A number of design patterns taught in the unit are based on polymorphism, and it's broadly the intention for you to use some of them. However, the marking guide does not allocate marks for any specific ones here.

We're simply going to check for the existence of *two* separate inheritance hierarchies where polymorphism is employed. However, there are some important caveats here:

- Simply having inheritance is not enough. Remember that polymorphism occurs when you make a method call without knowing which subclass's method you're actually calling.
- Your inheritance and polymorphism must be *useful* in some way. That is, it should achieve extensibility and/or decoupling.

Some examples of things that are *not* polymorphic:

- When your interface is empty, or your superclass has no abstract methods, or none of the abstract methods are ever called.
- When there are no references (fields, parameters, return types) to your interface or abstract class.
- When the calling code creates the subclass object, and so actually does know which subclass it's dealing with; e.g.:

```
MySuperClass var = new SubClass4();
var.superclassMethod(); // Everyone knows that 'var' is 'SubClass4',
                        // even if it's declared as 'MySuperClass'.
```

- When you have to use `instanceof` and downcasting.
- When you avoid `instanceof` and downcasting by re-inventing them in another form; e.g.:

```
if(var.isSubClass4())
{
    var.subClass4Method(); // Technically a superclass method, but
}                          // only makes sense for SubClass4 objects.
```

2 marks – UML consistency and relative completeness.

Ensure you understand the difference between association, aggregation, inheritance and usage dependencies. Remember that (in general) only non-static class fields are going to result in association or aggregation, as these are *long-term, object-to-object* relationships.

Ensure your diagram is consistent with your code! You can make simplifications to promote readability, but it has to be clear that this is intentional. Don't miss out important classes, or association/aggregation/inheritance relationships.

3 marks – Explanation of key aspects of the chosen design.

This will be part of a 2–3 page written document accompanying your design and code. Don't underestimate the effort required here. Even once you have your design, *explaining* it can take significant time and effort in its own right, even if the end result is not that long.

Imagine yourself trying to communicate your design to a colleague who doesn't know anything about it. Ask yourself what your colleague would want to know.

These marks are not available if you haven't actually implemented the design in code.

2 marks – Explanation of two plausible design alternatives.

This is the other part of the written document, and the purpose is for us to see that you're thinking deeply about the design.

There are always a virtually unlimited number of design options. A software designer must not just randomly pick one design, but be aware of several and weigh up their advantages and disadvantages in the given situation.

To this end, in addition to your actual design, you must describe *two more* possible designs, and again there are some caveats:

- Your alternative designs must be *plausible*. We're not interested in things that are ridiculous and/or wouldn't work. We're interested in relatively sensible alternatives that actually *would* work, but which you simply didn't choose, for more subtle reasons.
- You must give enough detail for us to see how your alternative proposals address the problem(s) at hand. Simply saying "I could have used pattern X instead of pattern Y" doesn't give enough information. Perhaps you could, but *how* would the rest of the system be redesigned to accommodate it? There could well be several different ways to use the same pattern!
- Your two alternatives must be reasonably different from one another, and from the actual chosen design. If the differences wouldn't show up at the level of a UML class or object diagram, they're probably not different enough.

5 Academic Integrity

Please see the *Coding and Academic Integrity Guidelines* (available alongside this specification on Blackboard).

In summary, this is an assessable task. If you use someone else's work or assistance to help complete part of the assignment, where it's intended that you complete it yourself, you will have compromised the assessment. You will not receive marks for any parts of your submission that are not your own original work. Further, if you do not *reference* any external sources that you use, you are committing plagiarism and/or collusion, and penalties for academic misconduct may apply.

Curtin also provides general advice on academic integrity at
http://academicintegrity.curtin.edu.au/.

The unit coordinator may require you to provide an oral justification of, or to answer questions about, any piece of written work submitted in this unit. Your response(s) may be referred to as evidence in an academic misconduct inquiry.

A Problem Description: Turn-Based Combat Game

Design and implement a simple turn-based combat game. In the game, the player will operate a character who fights a sequence of battles against a range of enemies. In between each battle, the player may purchase items from the shop.

Each battle involves the player's character and one enemy. During the battle, the player attacks (or takes some other action), then the enemy attacks, and so on. Each attack reduces the opponent's "health" by a certain amount (the exact amount depending on a range of factors, described below).

The player wins a battle when the enemy's health reaches zero, and wins the entire game when they defeat the dragon. The player loses a battle, and the entire game, when their own character's health reaches zero.

(For our purposes, there is no storyline, nor any area/map that the character and enemies can move around in. It is simply a sequence of battles.)

Future Extensibility: Pay attention to boxes like this.

There are certain kinds of functionality that you *do not* need to actually write the code for, but which your design should nonetheless accommodate. Consider how to make it as easy as possible for a future developer to extend your work.

A.1 Character

The player's character has the following attributes:

- Name.
- Maximum health – an upper limit on the health a character can have (default 30).
- Current health – an integer (by default, the maximum health).
- Inventory – a collection of up to 15 items that the player can access (see Section A.3).
- Chosen weapon and a chosen armour – two particular items in the character's inventory that the character will fight with.
- Gold – a number, used like money to buy and sell items (default 100).

On startup, the character will be automatically given (without cost) the cheapest available weapon and armour. (See Section A.7 for details on what these actually are.) These will be part of the character's inventory, and also their "chosen" weapon and armour.

A.2 Health, Damage, and Defence

Each attack, whether it's the character attacking the enemy, or the other way around, works according to the same basic mathematical formula.

An attack has a certain strength, called "damage", expressed as an integer (≥ 0). Meanwhile, the opponent (whoever is on the receiving end of an attack) possesses "defence", also an integer (≥ 0). Together, these determine the effect of an attack on the opponent's health, as follows:

```
health = health - max(0, damage - defence)
```

That is, if defence $\geq$ damage, then there's no effect. Otherwise, health is reduced by the difference between damage and defence.

However, this is a simplification, for two reasons:

(a) There are always *minimum* and *maximum* values for damage and defence (for any given situation). The actual values must be chosen at random from that range (inclusive).

(b) Damage can be further modified by various other effects, depending on the situation at hand.

So a more accurate pseudocode calculation is as follows:

```
damage = modifier(rand(minDamage, maxDamage))
defence = rand(minDefence, maxDefence)
health = health - max(0, damage - defence)
```

Where:

- "`rand(minValue, maxValue)`" is shorthand for generating a random integer between minValue and maxValue, inclusive. See Appendix B.

  The player can affect the minimum/maximum damage they inflict, and their defence against enemy attacks, through their choice of weapons and armour. For each enemy, their own damage and defence minimums/maximums are simply characteristics of that enemy.

- "`modifier(damage)`" represents some operation (or series of operations) that involve, for instance, adding to or multiplying the damage. This depends on weapon "enchantments" or enemies' "special abilities". See the relevant sections below for what these effects actually are.

A.3 Items

The character can possess (in their inventory) various items, including *weapons*, *armour* and *potions*. These can be bought or sold at the shop (see below).

Each different weapon inflicts a particular amount of damage (or rather minimum/maximum damage) on the enemy. Each different form of armour provides a particular minimum/maximum defence (i.e., absorbing a certain amount of damage from the enemy).

A potion can either be damaging or healing. A damaging potion can be used like a weapon against the enemy. A healing potion can restore some of the character's health (again, with minimum/maximum values):

```
healing = rand(minHealing, maxHealing)
health = min(maxHealth, health + healing)
```

All types of items have:

- A name.
- A cost in gold (for when the player wishes to buy or sell it).
- A minimum and maximum effect (integers).

In addition, a weapon has a type (e.g., "sword", "axe", "staff", etc.) and a damage type (e.g., "blunt", "piercing", "slashing", etc.). An armour item also has a material (e.g., "cloth", "leather", "chain", etc.). Potions have no extra attributes.

These attributes (and each item's name) are just strings. They don't have any special meaning within the game, but should be used to describe the item on-screen where appropriate.

A.4 Enemies

Each time a player starts a new battle, a single random enemy must be selected. Here are the range of different enemy types, and the initial probability of each one appearing:

Species	Initial probability
Slime	50%
Goblin	30%
Ogre	20%
Dragon	0%

After each battle, these probabilities *change*. The probability of each non-dragon enemy type decreases by 5% (unless that would take it below 5%), and the probability of the dragon increases accordingly (15% at first). For instance, after the first battle, slime would have a 45% chance of being selected for the second battle, goblins 25%, ogres 15%, and dragons 15%.

All enemies have the following features:

- A species name.
- Maximum health.
- Current health (initially set to the maximum).
- Minimum and maximum attack damage (to be inflicted on the player's character).
- Minimum and maximum defence (to absorb the damage from the player's attacks).
- Gold awarded to the player upon defeat.
- Certain special abilities.

More specifically, the four types of enemies have the following characteristics:

Species	Max. Health	Damage	Defence	Gold	Special Abilities
Slime	10	3–5	0–2	10	20% chance that its attack will have no damage.
Goblin	30	3–8	4–8	20	50% chance that its attack will have 3 extra damage.
Ogre	40	5–10	6–12	40	20% chance that it will attack twice in a row (without the player having a turn in between).
Dragon	100	15–30	15–20	100	35% chance of *one* of the following happening when it attacks: (a) damage inflicted will double (25% chance), or (b) it will recover 10 health (10% chance).

Special abilities are only triggered *sometimes*, randomly. See Appendix B. When triggered, they modify the standard calculation of damage, but (as you can see) they may have other effects as well.

For example, if the player is fighting an Ogre, then for each Ogre attack there is a 1-in-5 chance that the damage will *not* just be `rand(5,10)`, but rather `rand(5,10)+rand(5,10)`; i.e., two random numbers between 5 and 10 will be generated, and added together. (Note that this is different from multiplying one random number by 2.)

By contrast, when a Goblin attacks, there is a 1-in-2 chance that its damage will be increased by 3; i.e., `rand(3,8)+3`.

> **Future Extensibility:** While you only need to implement these four enemy types, your design should seek to make it easy to add others in the future.

A.5 Weapon Enchantments

Apart from different weapon types, each weapon can have "enchantments" added to it, which will increase the damage it can inflict. Any number of enchantments can be added to a weapon. For simplicity, they cannot be removed.

There are four types of enchantments:

Name	Cost (in gold)	Effect
Damage +2	5	Adds 2 to attack damage.
Damage +5	10	Adds 5 to attack damage.
Fire Damage	20	Adds between 5–10 (randomly) to attack damage.
Power-Up	10	Multiplies attack damage by 1.1.

For an weapon with multiple enchantments, their effects are evaluated in the order in which they were added. For instance, if the player adds a "Damage +5" *and then* "Power-Up", and the weapon would ordinarily inflict between 10–12 damage, then the actual damage is: `(rand(10,12)+5) * 1.1`.

> **Future Extensibility:** As with enemies, although you only need to implement these four enchantments, your design should seek to make it easy to add others in the future.

A.6 Game Menu and Information Screen

Upon startup, and after each (successful) battle, the game should present the following menu options to the user:

- Go to Shop,
- Choose Character Name,
- Choose Weapon,
- Choose Armour,
- Start Battle, and
- Exit Game.

Alongside these options, the game should display all the character's attributes, including their name, health, gold, inventory, etc.

A.7 The Shop

At the start, and in between battles, the player may visit the shop to purchase new items, add enchantments to weapons in their inventory, or sell items in their inventory.

When in the shop, the game should continue to show the player's attributes, but also list everything available to be purchased, and provide options for purchasing and selling.

Each item or enchantment for sale has a cost in gold, and the player must have enough gold to pay for it. Once bought, the player's gold is reduced by that amount. The shop has an unlimited number of each item and enchantment for sale.

A player may sell an item for 50% of its purchase price. However, a player *cannot* sell their currently-chosen weapon and armour. To do so, they would need to first choose an alternative.

Once sold, an item disappears from the player's inventory. When selling a weapon, the weapon and all its enchantments are sold as one item (with the cost of the enchantments added to the cost of the original weapon). Enchantments cannot be sold individually.

A.8 Shop Item Data

The items available for purchase are to be read in from a file. Each item appears on a separate line, starting with "W", "A" or "P", with their specific details separated by commas, as follows:

Weapons: `W`, *name*, *minDamage*, *maxDamage*, *cost*, *damageType*, *weaponType*
Armour: `A`, *name*, *minDefence*, *maxDefence*, *cost*, *material*
Potions: `P`, *name*, *minEffect*, *maxEffect*, *cost*, `H` *or* `D` (*for Healing or Damage*)

All enchantments are also available for purchase. However, they don't appear in the input file, because their details are already fully specified in section A.5.

Note: Make sure to comprehensively check for and handle errors in this file!

Future Extensibility: Your design should make it easy to add *alternate sources* of shop item data in the future. Rather than loading the data from a file, it could instead be loaded from a database, downloaded from a web service, or randomly-generated.

As part of this, we want to allow (in future) for the shop data to be changed/reloaded mid-way through a game, so that the player's choices change as the game progresses.

Here's an example file:

```
W, Short Sword, 5, 9, 10, slashing, Sword
W, Great Axe, 8, 15, 15, slashing, Axe
W, Magic Staff, 1, 40, 20, Bludgeoning, Staff
A, Leather Armour, 5, 15, 10, Leather
A, Chain Mail, 10, 18, 50, Chain
A, Dragon Skin, 20, 30, 80, Dragon Scale
```

```
P, Potion of Healing, 5, 10, 12, H
P, Potion of Greater Healing, 15, 20, 20, H
P, Explosive Potion, 20, 20, 20, D
```

A.9 Battles

Prior to a battle, the player may select a weapon and armour item from among their character's inventory. During a battle, this weapon and armour cannot be changed.

When engaged in battle, the character's attributes must continue to be displayed. The events of the battle must be listed as they occur, including who attacked (or healed) who, what the damage, defence, healing (if any) were, what special abilities were triggered (if any), and what potions were used (if any).

There is no time limit to the player's turn, but during their turn the player may choose to either:

- Attack with their chosen weapon (and its enchantments, if any), or
- Use a potion from their inventory (which could either heal the player's character, or inflict damage on the enemy). The potion will disappear from the player's inventory once used.

The enemy (chosen at random at the start of the battle, as discussed in Section A.4) then makes an attack on the player's character. Its special ability(ies) now have a chance to be triggered. Then the player's character (if still alive), takes another turn.

If the player's character reaches zero health, the game ends. Otherwise, if the enemy's health reaches zero first, the battle ends. At that point:

- The character obtains the gold awarded to them (based on the enemy type);
- The character's health increases as follows:

```
health = min(maxHealth, health * 1.5)
```

For example, say the player defeats "slime" and has 12 (out of 30) health remaining. The player would receive 10 gold for defeating the enemy, and their health would increase to 18 (12×1.5).

A.10 Selecting Weapons or Armour

When selecting weapons or armour, the game should display the currently-chosen weapon/armour, and list all the other available options. The player may select one, or return to the main game menu without altering their selection.

B Reference: Random Numbers

Random numbers can be generated in any language:

Java: using objects of the standard `Random` class, or `Math.random()`.
C#: using objects of the standard `Random` class.
Python: using the standard `random` module.
C++: using `rand()` and `srand()` (from C), or `<random>`.

In order to make some decision *probabilistically* – say, 20% of the time – you generate a random number between 0 and 1, and check if it's lower than your required threshold:

```
if(Math.random() < 0.2)
{
    ... // Do something 20% of the time.
}
```

End of Assignment