Assignment Specification

> This page contains the tasks you'll need to complete for this assignment, and how you'll be assessed.

TOC

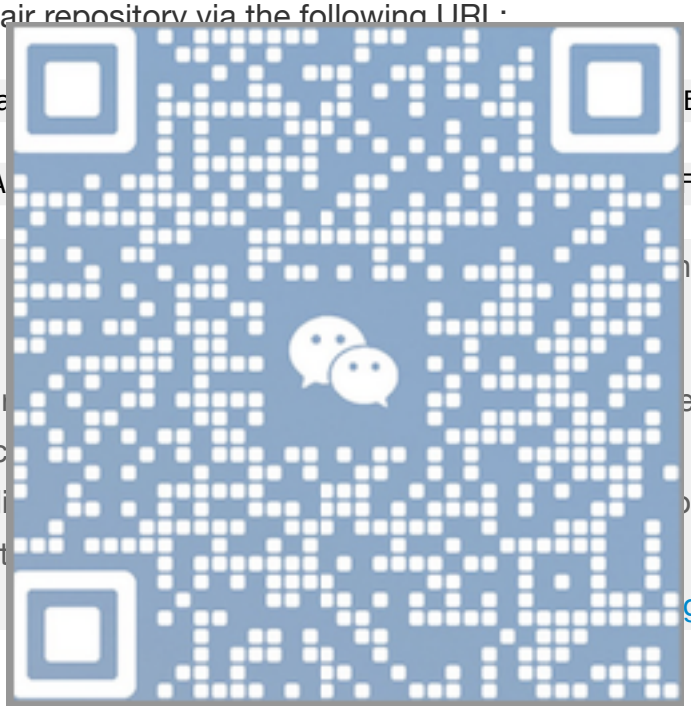
1. Getting Started

You can access your pair repository via the following URL:

`https://nw-syd-gitla` `EAM_NAME/assignment-ii`

Replace `YOUR_TEAM_NA` `=`)

> For this assignment, nd add these to your repo:
>
> - Your pai e questions and links to your c
> - Your indi our individual contribut
>
> You can g here.



2. Tasks

> Important Breadcrumbs…
>
> This Assignment has three tasks. However, **we recommend you read the whole spec before starting Task 1**. Parts of different tasks may affect how you complete the assignment.
>
> **Follow the trail of breadcrumbs that leads to Task 3…**

Task 1) Code Analysis and Refactoring (40 marks)

In this task, you will need to analyse the design of the monolith, including the patterns and smells present in the implementation, and apply refactoring techniques discussed in the course in order to improve the quality of the code.

Instructions for Task 1:

Put all of your answers to the following theory questions in your **pair blog post**.

For each refactoring change, **before you start coding** take some time to plan and write up a design in your p

- What fiel ass
- What ne

After you finish cod anch of your repository with the fo

- A meanir
- A brief d de
- Make su make any extra changes e number of files the changes
- Make su sion tests, linting, coverage) remain passing.

Your partner will need to **code review** and either:

- Leave comments in the MR with things to fix (you can do code reviews sync or async), requiring you to **iterate on the MR** and **resubmit** for review; or
- Approve the MR.

Once the MR is approved, copy and paste the link to your MR into your blog post.

Make sure to do this for each part of the question - **if you don't you will lose marks**.

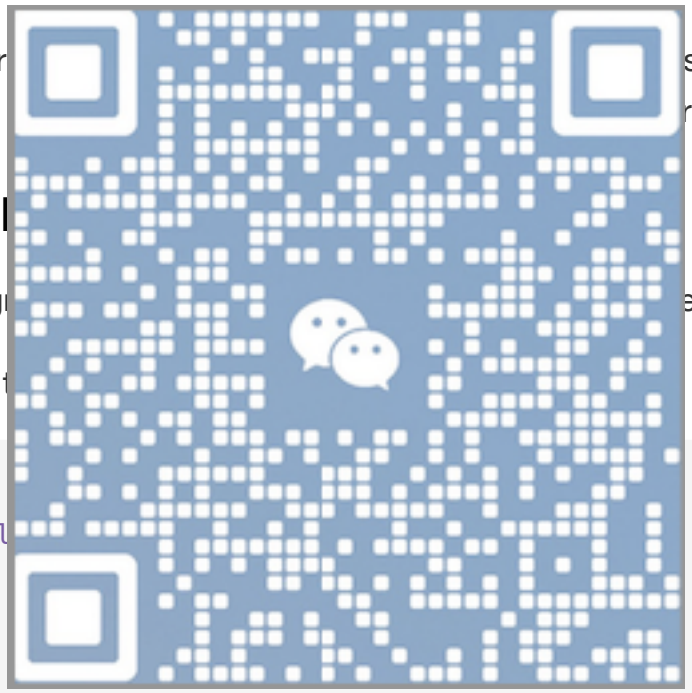
a) From DRY to Design Patterns (7 marks)

i. Look inside src/main/java/dungeonmania/entities/enemies. Where can you notice an instance of repeated code? Note down the particular offending lines/methods/fields.

ii. What Design Pattern could be used to improve the quality of the code and avoid repetition? Justify your choice by relating the scenario to the key characteristics of your chosen Design Pattern.

iii. Using your chosen Design Pattern, refactor the code to remove the repetition.

b) Observer Pattern (5 marks)

Identify one place wher  se, and outline how the implementation relates rn.

c) Inheritance I

Currently, there is a sig es.

Consider the following

```
@Override
public void onOverl
    return;
}

@Override
public void onMovedAway(GameMap map, Entity entity) {
    return;
}

@Override
public void onDestroy(GameMap gameMap) {
    return;
}
```

i. Name the code smell present in the above code. Identify all subclasses of Entity which have similar code smells that point towards the same root cause.

ii. Redesign the inheritance structure to solve the problem, in doing so remove the smells.

d) More Code Smells (6 marks)

The previous engineering team has left you with the following note:

> Collectable entities are a big problem. We tried to change the way picking up items is handled, to be done at the player level instead of within the entity itself but found that we had to start making changes in heaps of different places for it to work, so we abandoned it.

i. What code smell is present in the above description?

ii. Refactor the code to [illegible] ausing it.

e) Open-Close[illegible]

Look inside the goals p[illegible]

i. Do you think the desig[illegible] lies with the open-closed principle? Do yo[illegible]

ii. If you think the desig[illegible] think the answer is no, pick a suitable Desi[illegible] code and refactor the code accordingly.

f) Open Refact[illegible]

Make any other refactoring improvements you see fit to the codebase. This can include resolving Design Smells, using Design Patterns discussed or any other general improvements to the health and quality of the code.

Some areas and questions you can consider:

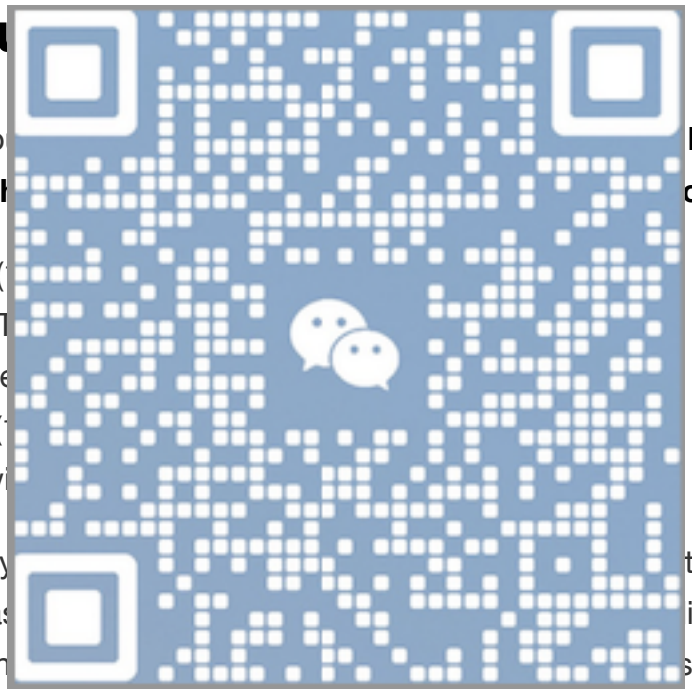
- Look for violations of the Law of Demeter/Liskov Substitution Principle;
- The effects of potions (invisibility and invincibility) has been implemented using a State Pattern. However, the State Pattern hasn't been used particularly effectively here and as a result, there is poor design.
- The current implementation of buildable entities contains a significant amount of

hard coding. Think about how you can improve this.

The above list isn't exhaustive; there are plenty of other areas to improve the quality of the code.

> Don't make solutions to problems that don't exist yet! 😃 Avoid over-engineering or over-abstracting in places where you might want to improve the design for some future change in requirements - instead improve the design of the current system. This will inherently make your software open-closed.
>
> You'll also want to split this task into one MR for each refactoring change you make.

Task 2) Evolu marks)



In this task, you and yo **Microevolution and the same number of th our group**:

- b) Bosses (
- c) Swamp T
- d) Sunstone
- e) Snakes (
- f) Logic Swi

E.g. If there are two of y there are three of you, you pick three. Each tas ifficulty. You should pick tasks you feel com s, that is fine - we will take your performance in the best N tasks, where N is the number of people in your team.

If you are in a group of 3, your mark will be out of 65 and scaled to 50.

Write all your tests for this question inside the `task2` folder.

Software Delivery - Task Lifecycle

For each part of this task, you will need to undertake the following process. This will help your tutor assess your design during marking.

1. **Requirements Engineering**. Analyse the task requirements, including the

technical and product specifications. If you need to, make some assumptions and document these in your pair blog post.

2. **Detailed Design**. In your pair blog post, plan out a detailed design for the task. This should include:
    - What fields/methods you will need to add/change in a class
    - What new classes/packages you will need to create
3. **Design Review**. Have your partner review the design, and go back and iterate on the design if needed.
4. **Create a Test List**. Once the design is approved, write a **test list** (a list of all the tests you will write) for the task. Map out each of the conceptual cases you want to test. This can be written in your blog post, or if you want make function stubs for JUnit tes  g).
5. **Test List R** the test list to make sure the tes ce.
6. **Create the** class/method prototypes.
7. **Write the t** y since the functionality
8. **Developme** ts pass.
9. Run a **usab** frontend).
10. Where nee d design while keeping the tests pa
11. Put up a **m** e CI should be passing. Th and contain a description of the changes you have made. In most cases you should just be able to link them to your design/test list blog.
12. **Code Review** from your partner, iterate where needed then they should approve the MR.

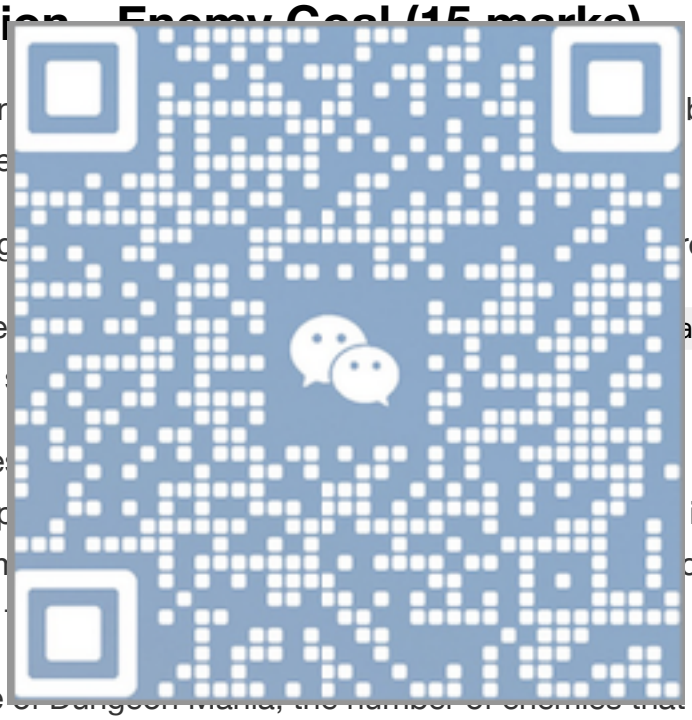
Feel free to split tasks further where you see fit, especially for larger tasks (e.g. Logic Switches). Keep your Merge Requests small and make the iteration cycle short, incrementally building your MVP as you go.

> 💡 If you've done Task 1 well and have a nice healthy codebase, this task should be relatively straightforward! If you're finding parts of this task difficult to integrate with the existing design, that's probably a sign you need to do some more refactoring 😃

♻️ A note about **backwards compatibility**:

- **All** the regression tests we have provided to you in the starter code should remain passing.
- All of the MVP configuration files (in the provided config files) do not currently contain the fields listed in the technical specifications for Tasks 2a)-f). Rather than retroactively adding these fields to the existing configuration files, you will need to design your implementation to accomodate for this and maintain backwards compatibility. **All configuration files in our Task 2 autotests will contain all values from Task 2** and the MVP.

a) Microevolution - Enemy Goal (15 marks)



In this section you will r base to introduce a new type of goal - the e

To achieve the enemy g requirements:

1. Defeat at le `ast` , `Spider` , etc)
2. Destroy all

Other existing goal rule exits must be completed last, still app is set to be a conjunction of the enen considered complete if the player first achieve

Note: for a given game must be defeated is determined by the configuration field listed in the **Technical Specification** section below.

Technical Specification

`enemies` is now included in the list of goals specified in 4.1.4. of the MVP specification.

The list of configuration fields in Section 4.2.1. of the MVP specification now includes the following fields:

JSON Format	Description
`enemy_goal`	At least x enemies must be killed to complete the enemy goal.

b) Bosses (10 marks)

In this task, you need to implement the following new entities.

Bosses are moving entities which are harder to defeat/conquer than normal enemies.

Entity	Image	Description
Assassin		Assassins are exceptionally powerful mercenaries which deal significantly more damage. When bribing an Assassin, there is a certain chance that the bribe will fail; the gold will be wasted and the Assassin will remain hostile. Battles still do not occur with an
Hydra		ecial creatures similar movement n a hydra is attacked its health will amount, as two .

Technical Specifi

The list of inputs in Sec s the following entities:

Entity		ngeon Map?
Assassin	`assassin`	Yes
Hydra	`hydra`	Yes

The list of configuration fields in Section 4.2.1. of the MVP specification now includes the following fields:

JSON Format	Description
`assassin_attack`	Attack damage of the assassin.
`assassin_bribe_amount`	The amount of gold required to perform an attempt