Applied session (Bootcamp)

Applied Session objectives - Bootcamp Week 3

In this week's Bootcamp, you will:

- refactor code to improve its design and readability.
- use the built-in ArrayList container class.
- create a console menu for I/O.
- make and justify some design choices.

Bootcamp specifications

Assessment: This applied session is worth 2% of your final mark for FIT2099. It will be awarded in the form of a checkpoint to be completed during the applied session next week as it directly contributes to the bootcamp's final delivery in Week 6.

Please, read all the requirements listed below first, then start working on the design before doing any coding.

Common Requirements (REQ 1 & 2)

As indicated in the previous bootcamp, there will be 2 common requirements for each bootcamp and they are reflected in the marking rubric accordingly. Please note we will start assessing **git commits** from this bootcamp onwards. These 2 requirements will remain the same and consistent across all bootcamps and will be assessed every time. Please read them very carefully to get yourself familiar with the common requirements and be ready for future bootcamps as well!

Link: https://edstem.org/au/courses/25302/lessons/84435/slides/576398

Specific Tasks (REQ 3)

There will be 4 tasks under REQ3 for each bootcamp. These are designated to help you consolidate and assess your understanding of that specific week's content and are aligned with the Applied session objectives. The 4 tasks will be different for each week and will be built on top of the previous week's bootcamp. These tasks will cover both coding and design skills we anticipate you to hold throughout the learning journey.

Link: https://edstem.org/au/courses/25302/lessons/84435/slides/576399

Remember, if you encounter any issues or have any questions, please don't hesitate to reach out to a TA for help! If you still have questions after the applied session, you can post them on the Ed discussion forum or attend a consultation.

Bootcamp Common REQs (REQ 1 & 2)

Requirement 1. Your code on Gitlab

You must have **at least three non-trivial commits** to your GitLab repository. If your previous bootcamp work hasn't been assessed yet, and you want to continue this week's bootcamp, please create a new branch `bootcamp-week-3` from your `main` branch. Ensure that you merge it back into the `main` branch before the interview/assessment. **We only assess work inside the `main` branch.**

If you haven't figured out how to create a new branch, please, go through the material on how to create a Git branch and/or ask your TA for a demonstration.

Requirement 2. UML Class Diagram

For this bootcamp session, just like last week, you'll need to create a UML class diagram for the system you plan to implement (make sure you've read through all the tasks first!). Before you start coding, be sure to show your diagram to a TA for feedback. The system you're creating will incorporate work from multiple tasks, so it's important to carefully read through the lab sheet. Your diagram should include all classes and their relationships, like *associations*, *dependencies*, and other UML relationships. You don't need to include attributes and methods in your diagram.

Please remember to save your design documents, such as the UML class diagram, in the "docs/design" folder and use a prefix of "w3-" when naming the file (for example, "w3-uml.png"). Additionally, please save the UML class diagram as a PNG or PDF file.

Bootcamp REQ3: Implementation

Requirement 3. Implementation items

This week, we will make our simulation better by changing the way we store items so it can grow and shrink easily. We will make sure that we can put things into your backpack and take them out wherever they are, in your backpack or around the campsite. We will also set up which actions that we want to do in each iteration of this simulation. More details will be discussed in the following points.

Requirement 3a. Refactoring

Let's start this week with Backpack and Campsite. Last week, we created an array inside the Backpack class to store all available items. This time, we want to **unpack** the items in our backpack and set them up at the campsite. Furthermore, we can also **pack** the items available at the campsite and put them in our backpack.

Now, imagine we are camping right now. We could have a lot of items at any time, either inside our backpack or *scattered* around the campsite. In this case, would having a fixed-size array make sense? This suggests that both the backpack and campsite will not have a fixed number of items forever. How should we change this?

> The solution is quite simple. Instead of having a fixed-size array, we want to use a more flexible size so that we can easily add or remove things in the future. We can use ArrayList. Please refactor our Array to an ArrayList and make other modifications accordingly. We should have two ArrayLists: one for Items in the Backpack (containing all Items that are "packed") and another for Items in the Campsite (containing all Items that are "unpacked" and scattered in the campsite).

On top of that, our Camper will have 2 additional attributes, hydration level (20 - the lower the value is, the thirstier we are) and coldness level (20 - the lower the value is, the warmer our body is). Please adjust the `toString()` to display these two new attributes as well. We will use these two attributes in the following weeks.

Requirement 3b. Implementation of Action, PackAction, and UnpackAction classes

Since we have already set up an Item class, we will now perform some actions on the items. We are now introducing the notion of actions to help you prepare for your assignments later in the semester. The rationale behind the notion of an "action" is that, instead of implementing all the features in the Campsite class, we will let the corresponding classes tell us what action we can perform on a particular object (and that is the essence of Object-Oriented Programming). For example, in this case, we can create one or more action objects that would allow us to perform some actions on the Item

objects. Then, we let the `Item` objects tell us what actions we can perform on them by making them return a list of "allowable" actions, i.e. actions that we are allowed to perform on them.

Before we start working on the Action classes, we need to think of how the system could be extended. Since these action objects can be used to perform some actions on Item objects, there must be something in common among them. Hence, it makes sense to create an Action class that can serve as a base for specific actions later on. Think of what type of class this Action class should be: concrete or abstract? (Hint: Are we going to create an Action object?)

There are 2 things you may want to think about:
- What is/are the difference(s) between a concrete class and an abstract class?
- What is/are the difference(s) between a concrete method and an abstract method?

Inside this `Action` class, create two methods. The first method, `execute`, should return a String. The second method, `menuDescription`, also returns a String. While you are thinking about the class type (i.e., abstract or concrete), you might also consider what type of methods they should be (abstract or concrete). The execute() method will eventually contain all the logic needed to perform the actual action, while the menuDescription() method serves to provide information about how the action will be shown as an option in a text-based menu.

NOTE: This notion will also be relevant in preparation for the major assignments later in the semester.

Once we have finalised our Action class, we will introduce our first action to pack the item in the form of the `PackAction` class. You may use the skeleton below to help you implement the `PackAction`. How do we indicate that this new class should follow the structure of the Action class mentioned above? Some additional modifications might be needed to complete this class.

```java
public class PackAction extends Action{  1 usage
    private Item item;  4 usages

    public PackAction(Item item) { this.item = item; }

    @Override  1 usage
    String execute(Camper camper, Campsite campsite) {
        //The idea of this action is that:

        // we want to pack the item that's available at the campsite to our backpack.
        // if we can pack the item (meaning that the weight will not exceed the backpack weight limit)
        // then we will pack the item and remove the item from the campsite
        // otherwise, we will say that the item can't be packed as it will exceed the backpack limit

        if (/*the condition whether we can pack or not*/) {
            // do what we need to do here
            return camper + " packed " + this.item.getClass().getSimpleName() + " to the backpack";
        }
        return this.item.getClass().getSimpleName() + " can not be packed as it will exceed the backpack limit";
    }

    @Override  1 usage
    String menuDescription(Camper camper) {
        return camper + " will pack " + this.item.getClass().getSimpleName() + " to the backpack";
    }
}
```

In the code above, you should indicate that the `PackAction` class follows the structure of the Action class by using the extends keyword, making `PackAction` a subclass of Action. This ensures that `PackAction` inherits the methods and structure defined in the `Action` class. Additionally, you should use the `@Override` annotation to indicate that these methods override the corresponding abstract methods in the Action class. Once again, `PackAction` will remove an item from campsite and put it to the camper's backpack.

```java
public String getSimpleName() { return this.getClass().getSimpleName(); }

@Override
public String toString() {
    return String.format("%s (%s) has weight of %.2f kg",
            this.getSimpleName(),
            this.name,
            this.weight);
}
```

⚠ Update Mon Aug 11th, noticed that we have the `this.item.getClass().getSimpleName()` and it would be harder if we need to call this chain of action over and over again. It would be great if we add this method in our Item class so every time we just want to get the name of the class, we can just call the `getSimpleName()` directly. Just a quick note about this, we shouldn't ever use the getSimpleName as a simple check to do something (i.e. if the object simple name equals to 'xyz' then we would do something, that would be a bad design practice.)

```java
public boolean add(Item item){  1 usage
    double estimatedWeight = this.totalWeight() + item.getWeight();
    if (estimatedWeight > limit){
        return false;
    }
    this.storage.add(item);
    return true;
}
```

An additional feature that we can add to make our simulation more meaningful is adding a weight limit to the backpack. It would be up to you how many kg that we want to have for each of the backpack, in the sample output, we will have 10kgs weight limit on our backpack. Furthermore, every time we pack an item to our backpack, it will check first whether it can fit the weight limit or not, if it doesn't, then the `PackAction` will return another statements inside the `execute()` (i.e. `Bedroll can not be packed as it will exceed the backpack limit`). The snipped code that we have above is the add method inside the `Backpack`.

⚠ Edit Aug 8th 10:47am: Added the weight limit to the backpack which is related to `PackAction` that we have.

Great! Now we can pack scattered items that we find at our campsite into our backpack. Next, it would be weird if we could pack items into our backpack, but we can't unpack those items. It would be great if we could unpack those items from our backpack to our campsite. In short, create a new `UnpackAction` class that removes an item from the `Backpack` and adds it to the `Campsite`.

ℹ **Forward reference**. If you are not sure why we added the `execute` and `menuDescription` methods to this action class, you might want to refer to this Assignment Support Module reading in the future: https://edstem.org/au/courses/25302/lessons/84438/slides/576425

Requirement 3c. Implementation of Menu

We will now proceed with the actual interaction! Nowadays, most programs use a graphical user interface that you can access online. However, there are times when a console-based user interface is needed. This is like having a conversation with an old typewriter - you type something in, and the computer responds with text.

Let's simply create a console menu. As for now, the only functionality that the console menu should support is "allowing us to pack and unpack items".

Alright, it's just read inputs and output something, how hard can that be? Thinking in an OOP way, how we can reuse such reading input and outputting work is the crucial part.

✗ A sub-optimal way would be like this

DO NOT USE THE FOLLOWING CODE SNIPPET IN YOUR BOOTCAMP WORK

```
public void consoleMenu() {

  int selection;

  do {

    System.out.println("1) Pack Item");

    System.out.println(“2) Unpack Item”);

    selection = Integer.parseInt(scanner.nextLine());

    switch (selection) {

      case 1 -> pack();

      case 2 -> unpack();

      default -> system.out.println(“We go home from campsite”)

    }

  } while (selection > 0 && selection < 3 );

}
```

However, this would result in the `Campsite` class being responsible for a lot of things (we will see why this is not good in week 5). Furthermore, if we want to add new things, e.g. allowing us to perform some other actions, the `Campsite` will have more and more lines of code, which could make it difficult to maintain. If a new developer is asked to maintain this code, they would need to read hundreds or even thousands of lines of code, which could be overwhelming!

A more effective way is to ask each individual class to take specific responsibilities. For example, an item allowing the `PackAction` / `UnpackAction` to be performed (remember what we talked about before about the notion of Action?) This reduces the responsibility of the `Campsite` class, and consequently, makes it more maintainable in the future.

Let’s try to build up a menu that can be extended. We will create a new **`Menu`** class to show all our current actions, as shown below.

```java
import java.util.*;

public class Menu {  1 usage
    /**
     * Credit to Minh Hoang Bui to optimize the old code for this
     */
    public static Action showMenu(List<Action> actions, Camper camper) {  1 usage
        Scanner scanner = new Scanner(System.in);
        Character startChar = 'a';

        Map<Character, Action> keyToActionMap = new HashMap<>();

        for (Action action: actions) {
            keyToActionMap.put(startChar, action);
            System.out.println(startChar + ": " + action.menuDescription(camper));
            startChar++;
        }

        char key;
        do {
            key = scanner.next().charAt(0);
        } while (!keyToActionMap.containsKey(key));

        return keyToActionMap.get(key);
    }
}
```

Let's analyse the above `Menu` class, or to be specific, the `showMenu` method. So first of all, we can see that the method itself is a static method, meaning that we are **able to call this method directly without instantiating a `Menu` object**. Second, the method takes a list of Action objects and returns a single Action object. Third, the body part before the do-while loop is intended to pair each action from that list with a character/alphabet. And the do-while loop is to make sure that you have selected a valid option (action) from the menu before it can proceed, so the final return Action object is the action that you choose.

You may be wondering why we need this class to just handle our Action objects. In later bootcamps, we will introduce more specific actions and see how this Menu class can handle additional actions without needing us to modify anything for reading inputs and writing outputs.

The notion of Actions and the Menu will be essential for Assignments 1-3. What we are currently doing in the bootcamp is a simplified implementation of the Menu class that you will find in the assignment engine.

Requirement 3d. Update our Simulation

Up to now, we have been working on small components, and we will work on the final small component before combining everything into our simulation.

```
public List<Action> allowableActions(){  1 usage
    List<Action> actions = new ArrayList<>();
    // add all the possible actions that can be done with the object
    // to the list of actions that we have just instantiated.
    return actions;
}
```

Let's add a new method called `allowableActions()` to our `Campsite`, `Camper`, and `Backpack` classes. This method will return **a list of all the possible actions that can be done with that object.**

Think of `allowableActions()` as the way each object will tell us:
"Hey, here are the things you can do with me right now!"

Here's how it works in plain English:

Camper

A Camper might be holding a Backpack. That means:

- The camper can ask the backpack: *"What can I do with you?"*
- The backpack will give a list of things it allows, like unpacking items.

So the Camper's `allowableActions()` will include:

- Actions from the Backpack it holds
  (Because the Camper has direct access to the Backpack, and we can just call the backpack's `allowableActions()` directly.)

Backpack

A Backpack may have many Items inside it. So it can say:

- "I have this item, and it can be unpacked."

The Backpack's `allowableActions()` will collect:

- An unpack action for each item inside it
  (Because you can take those items out)

Campsite

At the Campsite, some Items might be scattered on the ground.

- These items are not in the backpack yet, so we can choose to pack them.

The Campsite's `allowableActions()` will include:

- A pack action for each loose item on the ground
  (Because you can put those items into the backpack)

Inside the Campsite's `allowableActions()` method, instead of directly creating new `PackAction` instances for each scattered item, we should call each item's `getPackAction()` method to obtain the corresponding action. Similarly, inside the Backpack's `allowableActions()`, rather than directly instantiating `UnpackAction` for each item, we should invoke each item's `getUnpackAction()` method.

```
public Action getPackAction() { return new PackAction( item: this); }

public Action getUnpackAction() { return new UnpackAction( item: this); }
```

This approach helps keep actions organised and makes each object responsible for telling us what we can do with it - we can just ask the objects themselves.

Now, let's chain everything together. We just need to update the `simulate()` that we have inside the campsite. As we already know, there are actions that can be done from camper as well as from campsite itself, it would be great if we had a list of actions that contains all the allowable actions that we can retrieve from `Camper` and `Campsite`. Remember that the static method inside the Menu class requires the list of actions, so we can just pass in actions that we have just retrieved before to this static method.

Eventually, let's modify our simulate method inside Campsite class to the code below

```java
public void simulate() {  1 usage
    List<Action> actions = new ArrayList<>();
    actions.addAll(this.allowableActions());
    actions.addAll(camper.allowableActions());

    System.out.println("##########################################");

    camper.checkAllItems();
    this.listOutItems();

    System.out.println("##########################################");

    Action action = Menu.showMenu(actions, camper);
    System.out.println(action.execute(camper,  campsite: this));
}
```

You might notice that we have `camper.checkAllItems()`, the idea of this method is that we want to know what items we have inside our backpack, and we can show them via the terminal.

To make things easier when we test this week's content, feel free to adjust the campsite.simulate() method that we have inside the application to use a do-while loop, or a for loop. The output we provide uses a do-while loop, allowing us to run our simulation 5 times.. Also, you can decide the number of iterations for running the simulation as long as the simulation works correctly, e.g., the pack and unpack functionalities work well.

Checkpoint:
- Items that are scattered in the campsite, can be **packed** into the backpack
- Items inside our backpack, can be **unpacked** and we just scatter it around the campsite
- Allowable actions = list of possible actions that we can do toward the object (it is applicable for Campsite, Camper, and Backpack)
- Add Menu class to handle the input and output of our simulation
- Camper has additional attributes such as hydration (with starting value of 20) and coldness level (with starting value of 20).

Expected Output (Text Format)

In this example output, the simulation will be done in 5 iterations. Feel free to change it as mentioned in the brief, as long as we can see the pack-unpack features are working, that's great!

```
########################################
Here are the items that Cloudy has in the Backpack (8.50 / 10.00kg):
Bedroll (KAMUI) has weight of 7.00 kg - to rest.
Bottle (Mountain Franklin) has weight of 1.00 kg - to drink, with 1.0 liter left.
FlintAndSteel (Aurora) has weight of 0.50 kg - to start a fire.

Here are the items that we have on campsite:
Bedroll (KAMUI V2) has weight of 7.00 kg - to rest.

########################################
a: Cloudy (hydration level: 20, coldness level: 20) will pack Bedroll to the backpack
b: Cloudy (hydration level: 20, coldness level: 20) will unpack Bedroll from the backpack
c: Cloudy (hydration level: 20, coldness level: 20) will unpack Bottle from the backpack
d: Cloudy (hydration level: 20, coldness level: 20) will unpack FlintAndSteel from the backpack
a
Bedroll can not be packed as it will exceed the backpack limit
########################################
Here are the items that Cloudy has in the Backpack (8.50 / 10.00kg):
Bedroll (KAMUI) has weight of 7.00 kg - to rest.
Bottle (Mountain Franklin) has weight of 1.00 kg - to drink, with 1.0 liter left.
FlintAndSteel (Aurora) has weight of 0.50 kg - to start a fire.

Here are the items that we have on campsite:
Bedroll (KAMUI V2) has weight of 7.00 kg - to rest.

########################################
a: Cloudy (hydration level: 20, coldness level: 20) will pack Bedroll to the backpack
b: Cloudy (hydration level: 20, coldness level: 20) will unpack Bedroll from the backpack
c: Cloudy (hydration level: 20, coldness level: 20) will unpack Bottle from the backpack
d: Cloudy (hydration level: 20, coldness level: 20) will unpack FlintAndSteel from the backpack
b
Cloudy (hydration level: 20, coldness level: 20) removed Bedroll from the backpack
########################################
Here are the items that Cloudy has in the Backpack (1.50 / 10.00kg):
Bottle (Mountain Franklin) has weight of 1.00 kg - to drink, with 1.0 liter left.
FlintAndSteel (Aurora) has weight of 0.50 kg - to start a fire.

Here are the items that we have on campsite:
Bedroll (KAMUI V2) has weight of 7.00 kg - to rest.
Bedroll (KAMUI) has weight of 7.00 kg - to rest.

########################################
a: Cloudy (hydration level: 20, coldness level: 20) will pack Bedroll to the backpack
b: Cloudy (hydration level: 20, coldness level: 20) will pack Bedroll to the backpack
c: Cloudy (hydration level: 20, coldness level: 20) will unpack Bottle from the backpack
```

```
d: Cloudy (hydration level: 20, coldness level: 20) will unpack FlintAndSteel from the backpack
a
Cloudy (hydration level: 20, coldness level: 20) packed Bedroll to the backpack
########################################
Here are the items that Cloudy has in the Backpack (8.50 / 10.00kg):
Bottle (Mountain Franklin) has weight of 1.00 kg - to drink, with 1.0 liter left.
FlintAndSteel (Aurora) has weight of 0.50 kg - to start a fire.
Bedroll (KAMUI V2) has weight of 7.00 kg - to rest.

Here are the items that we have on campsite:
Bedroll (KAMUI) has weight of 7.00 kg - to rest.

########################################
a: Cloudy (hydration level: 20, coldness level: 20) will pack Bedroll to the backpack
b: Cloudy (hydration level: 20, coldness level: 20) will unpack Bottle from the backpack
c: Cloudy (hydration level: 20, coldness level: 20) will unpack FlintAndSteel from the backpack
d: Cloudy (hydration level: 20, coldness level: 20) will unpack Bedroll from the backpack
c
Cloudy (hydration level: 20, coldness level: 20) removed FlintAndSteel from the backpack
########################################
Here are the items that Cloudy has in the Backpack (8.00 / 10.00kg):
Bottle (Mountain Franklin) has weight of 1.00 kg - to drink, with 1.0 liter left.
Bedroll (KAMUI V2) has weight of 7.00 kg - to rest.

Here are the items that we have on campsite:
Bedroll (KAMUI) has weight of 7.00 kg - to rest.
FlintAndSteel (Aurora) has weight of 0.50 kg - to start a fire.

########################################
a: Cloudy (hydration level: 20, coldness level: 20) will pack Bedroll to the backpack
b: Cloudy (hydration level: 20, coldness level: 20) will pack FlintAndSteel to the backpack
c: Cloudy (hydration level: 20, coldness level: 20) will unpack Bottle from the backpack
d: Cloudy (hydration level: 20, coldness level: 20) will unpack Bedroll from the backpack
d
Cloudy (hydration level: 20, coldness level: 20) removed Bedroll from the backpack
```

Conditions for your bootcamp submission

(Same as last week)

> ⚠ **Be mindful of your bootcamp submission time as that depends on your allocated applied session time.**
> **Moodle submissions are not required as all bootcamp assessments will be conducted during the applied session.**

1. Last-Minute commits will not be considered

- Any commits made **during class** when the bootcamp is due for marking **will not** be considered.
- The version of your work at the official submission deadline is what will be assessed.
- Make sure all your work is committed and pushed **well before** the marking session begins.

2. Researching Previous Semester Works on GitHub

- Although we do not recommend this, you **are allowed** to refer to past semester's work by conducting your own research on GitHub. Please, note that the quality of any referenced work is **not guaranteed**—it could be a high-distinction submission or a failed attempt.
- This is an opportunity for you to learn how to **properly cite code**.
- You **must write all the code yourself** instead of copying and pasting.
- If you choose to build on someone else's work, you **must** ensure your contribution is original.
- If you are found copying code directly or excessively relying on another project, an **academic integrity alert** will be triggered.
- During your interview, you must show **a complete and solid understanding** of both the code you wrote and any referenced code. **Failure to do so will result in a score of 0 for the interview.**

3. Using ChatGPT & Proper Citation

- If you use ChatGPT to generate code, you **must** provide proper citations.
- Citations must be included in:
  - **In-line comments** within your code
  - **As per the citation guidelines for each assessment task in Moodle. For example:**

AI & Generative AI tools may be used in GUIDED ways within this assessment / task as per the guidelines provided.

In this task, AI can be used as specified for one or more parts of the assessment task as per the instructions.
Within this class, you are welcome to use foundation models (ChatGPT, GPT, DALL-E, Stable Diffusion, Midjourney, GitHub Copilot, and anything after) in a totally unrestricted fashion, for any purpose, at no penalty. However, you should note that all large language models still have a tendency to make up incorrect facts and fake citations, code generation models have a tendency to produce inaccurate outputs, and image generation models can occasionally come up with highly offensive products. You will be responsible for any inaccurate, biased, offensive, or otherwise unethical content you submit regardless of whether it originally comes from you or a foundation model. If you use a foundation model, its contribution must be acknowledged in the submission (in the form of a txt, pdf or word file titled FoundationModelsUsed added to your "docs" folder in GIT); you will be penalised for using a foundation model without acknowledgement.

Having said all these disclaimers, the university's policy on plagiarism still applies to any uncited or improperly cited use of work by other human beings, or submission of work by other human beings as your own. Moreover, missing contribution logs/GIT commits and/or failing any handover interview will be treated as a potential breach of academic integrity which will be further investigated.

Where used, AI must be used responsibly, clearly documented and appropriately acknowledged (see Learn HQ).

Any work submitted for a mark must:

1. represent a sincere demonstration of your human efforts, skills and subject knowledge that you will be accountable for.
2. adhere to the guidelines for AI use set for the assessment task.
3. reflect the University's commitment to academic integrity and ethical behaviour.

Inappropriate AI use and/or AI use without acknowledgement will be considered a breach of academic integrity.

- During your interview, you must demonstrate **a solid understanding** of any ChatGPT-generated code. **Failure to do so will result in a score of 0 for the interview.**

Marking rubric (Checkpoint)

In the **next** applied class, you will need to demonstrate progress through **a brief checkpoint review.** In this, you will showcase your development, progress and understanding of this week's bootcamp. Your teacher will use the following rubric to mark the progress of this bootcamp.

What is checkpoint? Both in software and in organisational contexts, a "checkpoint" generally refers to a predetermined point or moment where a status check, review, or evaluation is made. Checkpoints are commonly used internally to assess specific deliverables or milestones to date, offering indicators regarding progress towards meeting intended goals.

A pre-requisite for marking is that Req (1) should be awarded 1 or 0.5 points.

Req (1). GIT Commits

Excellent - Work has been progressively committed to GIT (more than 5 well formatted commit messages), changes and progress have been correctly commented, and progression of ideas and development seems clear.

1 points

Satisfactory - There is some evidence of progression of ideas and development (between 5 and 2 commit messages). There is some inconsistency or vagueness in commit messages.

0.5 points

Unsatisfactory - There is little or no evidence of progression of ideas and development, work is not commented correctly (vague commit messages) or commit messages do not correspond to the changes in the code or all the code seems to have been uploaded in one go.

0 points

Req (2). UML Design documents - clarity

Excellent - The implementation (the code) seems to fully address the requirements as per the UML design document (applying Object-Oriented principles).

2 points

Satisfactory - A design document is provided and it seems to address most of the requirements. All required classes are present and the UML syntax is generally correct. Some key relationship(s) between classes are/is missing and there may be one or more minor syntax errors which do not

affect the overall clarity of the documentation.

1 points

Unsatisfactory - A design document is not provided, OR the design does not address the requirements OR the documentation is unclear and not in accordance with UML syntax conventions or the application does not run OR Req (1) got 0 points.

0 points

Req (3). Implementation

Excellent - The implementation (the code) seems to fully address the requirements as per the UML design document (applying Object-Oriented principles).

2 points

Satisfactory - The implementation addresses the requirements with some design issues (not following the recommended design principles or the UML design document).

1 points

Unsatisfactory - The implementation does not really address the requirements even though some attempt was done or the application does not run OR Req (1) got 0 points.

0 points