Software Engineering Project: **ArtemisLite**

<u>Deliver methodically a well-designed and well-documented working system that satisfies the customer's requirements.</u>

Requirements

The emphasis in this project is on the process of requirements analysis, system design, software implementation and system testing that delivers reliable and appropriate functionality. The project will demonstrate your understanding of, and ability to put into practice, object-oriented software engineering principles, and your ability to work in a software engineering team. Your requirements analysis and system design will be represented in the graphical notation known as the Unified Modelling Language (UML). This will be a 'use case driven' development process, in which each use case describes "a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor" (Booch, Rumbaugh and Jacobson).[1]

The system to be developed is a virtual board game – but it does not have an elaborate graphical user interface. Instead the game is to be played via the console of the software development package (e.g. Eclipse). It will also have its own distinctive theme, based on NASA's mission to land the first woman and next man on the moon by 2024 (https://nasa.gov/specials/artemis/). It's called 'ArtemisLite' because it will reflect in very much simplified form some of the challenges (e.g. technical, logistical, financial,…) of the real lunar mission. Take some time to investigate the theme. Your game will involve investing in different parts of the project, developing different aspects of these, deriving benefit from them or devoting resources to them. Rather than use cash for your transactions (a traditional board-game favourite), you might want to think about allocating some other resource – people, equipment, know-how, time-and-effort. It's your job to think through and apply the 'metaphor' you want to use in your game.

The simple console-based interface gives you the opportunity to concentrate on the process of determining and designing the underlying object-oriented system, rather than focus on visual or audio effects. The system uses English to convey the state of the game and to ask its players what they want to do next. Though you don't have to develop a speech user interface (SUI), imagine a game where the state of play can be conveyed in words alone – whether a new development is being reported or the current state of play summarised.

[1] **Very important**: *each* use case is represented as a *single ellipse*, and each use case is **a complete set of sequences of actions in itself.** For example, a single use case might describe everything an actor does in order to Register with a system: enter first name, enter family name, enter DOB, enter house number, etc., etc. (note: this probably *won't* be one of the use cases in your *ArtemisLite* Game!). All those steps are represented by a *single use case* and a *single ellipse*. An ellipse in a diagram contains the *name of a use case*. The corresponding *use case description* describes what the sequence (flow) of actions would normally be to achieve a desirable outcome, which is the whole point of the use case! For example, if the Register use case executes successfully, the actor will have become a registered user of the system – that is the desirable outcome. However, the description of the Register use case *should also say* what alternative sequences (flows) are needed at particular steps under certain circumstances – what happens, for instance, if an actor enters an exclamation mark for the house number during registration!. In other words, the use case descriptions convey <u>much more information</u> that the ovals in the diagram. Several use cases (several ellipses) will typically appear in a single use case diagram. **Each ellipse represents a use case. Each use case must have a description**. **Never** use a use-case ellipse to represent a single step in a chain or sequence of steps. A single ellipse **IS** a set of sequences of steps – normal flow and alternative flows – that achieves some important outcome! So aim to have few rather than many ellipses in your diagram: each ellipse represents a use case, and each use case requires a description of the steps it involves! There are no bonus points for a jumble of ellipses. An ellipse without a description is pointless. Decide the important outcomes. Name the use cases accordingly. Describe the use cases carefully.

A game of this kind might start and unfold in the following manner (the example is not taken from ArtimisLite though the behaviour of your game will be broadly similar):

```
What is the first player's name? Janet
What is the second player's name? John

Janet, would you like to roll? y/n: y

OK Janet, you've rolled a 5 and a 6 – that makes 11.
You've landed on Square X.  No one is in charge of that yet.
Do you want to invest in this and take charge (y/n)? y

Investment made.
Your old balance was 1500 points.
Your new balance is 1440.

Janet is now in charge of Square X

John, would you like to roll? y/n: […]
```

Because the game is to be conducted in natural language only (i.e. English phrases that convey the state of play, etc.), it will have fewer ‘squares’ than a conventional board game. A separate game layout is required. This is meant to be *very, very simple* – literally like the little set of boxes below. (You should not produce a booklet about your game.) The purpose of the game layout is to show the position and attributes of the squares. You may create your game layout with any suitable drawing tool, such as PowerPoint or the drawing tool of Word, and should include this in the *Requirements Analysis* section of the *Short PDF Report* [see below]). Bear in mind that, in a full realisation of the game, the layout *could* [you don’t have to do this!] be converted into a non-visual equivalent, with, for example, embossed lines and Braille captions.

Again, the example shown is NOT for *ArtemisLite*, nor is it necessarily complete; it is intended only to represent the manner in which such a graphical representation might be drawn. Remember that, through the comments it writes to the console, the *software* itself must remind players of their positions, and their custodianship of squares and the properties of those squares.

Acquire Funding Collect X_I points	Square X Colour Brown To take charge X_X pts; Dev costs Y_X pts; Maj Dev costs Z_X pts.	Square Y Colour Brown To take charge X_Y pts; Dev costs Y_Y pts; Maj Dev costs Z_Y pts.	Square A Colour Blue To take charge X_A pts; Dev costs Y_A pts; Maj Dev costs Z_A pts.	Square B Colour Blue To take charge X_B pts; Dev costs Y_B pts; Maj Dev costs Z_B pts.	Square C Colour Blue To take charge X_C pts; Dev costs Y_C pts; Maj Dev costs Z_C pts.

Your game will have many of the features of a board game, but in a much simpler form. Within the constraints of the ‘natural language interface’, your customer’s core requirements are set out below, between the dashed lines (■■■■■■)] .

Remember, this is what your customer is asking for and expects to be delivered. You can realize these requirements while giving them your own creative ‘twist’ – but your customer is not asking for additional features (e.g. the customer does not want more squares, more types of squares, or more players than the number stipulated below). On the other hand, where checks are needed to ensure that the game is usable (for example, to avoid a situation where two players have exactly the same name, or to avoid the game accidentally ending), then such checks should be implemented, even if they are not explicitly requested. That is simply good design.

Similarly, although your customer has no *immediate* plans to ‘adapt’ or ‘upgrade’ ArtemisLite to a more specialised or more complex game, there may be (OO) design features that, with only a little additional effort, you can incorporate ‘behind the scenes’ to make your system more *maintainable* and *extensible* – e.g. a well-designed game would allow the number of squares or the maximum number of players to be increased or reduced easily in the code, should that requirement arise in future. Good software design not only meets present requirements but can easily accommodate change. [N.B. A separate user interface for major re-configuration of the game is NOT one of the requirements for ArtemisLite.]

Here are the core requirements. Your game should do the following and have the following features (or do/have something that is functionally/conceptually equivalent[2]):

■■■

The game has up to four players, and their names should be entered – the names may represent organisations rather than individuals.

The players take turns. They throw 2 virtual dice. Players are told where they have landed and what their obligations or opportunities are. Where appropriate, they may indicate their choice of action. For example, by dedicating some of their resources, they may 'take charge' of a square no one else owns. If a player lands on a square no one owns, but they don't want to take charge of it themselves, it may be offered at the usual cost to another player. If a player's resources have changed, the system indicates the reason for the change and announces the player's new 'balance' (e.g. the 'funds', 'credits' or 'resources' that are still available).

There is a start square, where players pick up their resources (it's your choice what the 'resources' represent – and you should be inventive with the name of the square – this is the equivalent of a 'Collect X as you pass Go' square). There is a square where nothing happens – again, you decide what it is called in your game.

There are four 'systems', two consisting of three adjacent squares or 'elements' and two consisting of two adjacent squares or 'elements'. Decide what the systems are called and what elements they will include: these should be based on the real Artemis project – that is part of your requirements analysis – but you will have to simplify, merge or omit the details of the real-world project in order to achieve a match with the constraints of your game. For example, you may (but you don't have to!) decide that the Space Launch System of the real-world is appropriate as a system in your game too, and that SLS Boosters will be an element of the Space Launch System [say it out loud: it should makes sense!]. One of the two-element systems (in your game at least) is the most costly system to acquire and resource; another two-element system is the least costly system to acquire and resource.

Before you can develop an element within a system, you must own/manage/'be in charge of' the *whole* system (you decide what 'custodianship' means!). On your turn you can develop an element in a system that you already own, even if you are not currently positioned on that element. Unlike similar board games, with ArtemisLite you may, if you wish, develop one element fully before developing the others, and you may undertake more than one development per turn.

Decide what a development is called, what it represents, and how much it costs in your game. Three 'developments' of an element are needed before you can complete (and pay for, or otherwise 'resource') the equivalent of a 'major development' (again, you decide what this represents and what it costs).

Not only is there a cost associated with developing elements within systems: when you land on an element, but do not 'own' it yourself, you ***normally*** *have to give up some of your resources for it – the more developed the element, the greater the resource that you are normally expected to give the owner. However, in the interests of the greater good, the owner may* ***decline*** *to accept your offer of resources. Why would that be…?*

Well, if one player runs out of resources during the gameplay, or if one player no longer wants to play, the game ends for *all* the players. Is the player without resources the loser, or have they heroically given their all in the noble cause of space exploration? You decide. When the game ends, show the final state of play: who held which

[2] For example, in your video demo you should be able to show that 'developing an element has a cost': if you have called elements 'parts' and your cost 'effort', then you should be able to show how a 'developing a part' entails 'effort'.

squares, how developed were they, and what other resources did the players hold? There is no need to convert ‘developments’, etc., to an equivalent value in your ‘resource units’.

On the other hand, if all the elements of all the systems are fully developed, Artemis is ready for launch: all systems are ‘go’ and the project is on the path to ultimate success! It pays, in other words, to keep people in the game, let them acquire and develop systems, and together complete the mission! As soon as development is complete, announce the path ahead: this will be like a summary of future events at the end of a movie (an epilogue). Display the successful outcome dynamically as a sequence of headlines: e.g. in 2021[…], then in 2022 […] until finally a successful landing is achieved, with congratulations all round! Also give the final state of play that made it possible.

■■

Deliverables

(See the separate Activity Plan for detailed delivery dates and times, and see the General Instructions at the end of this document!) These are group deliverables, to be produced and agreed by project teams. Once GitLab repositories are allocated, teams should use appropriately named folders in GitLab to co-ordinate their work. Final deliverables are uploaded to Canvas. One team member should upload each deliverable to Canvas on behalf of the whole team.

A: Working System – Code and Video Demo

Functionality that satisfies the requirements between the dashed lines (■■■■■■) attracts up to **10** marks. Up to **10** additional marks are available for systems that, within the constraints of the text user interface, deliver the required functionality in a manner that demonstrates excellent usability, including clear, timely and engaging interaction with the players, and a novel and coherent interpretation of the ArtemisLite theme. Though it is sometimes fun, elaborate ‘ASCII art’ – i.e. making shapes, etc., out of text – is not required for this system and is likely to detract from it: remember the SUI/’words-alone’ guideline above!

The **working system** (in the form of a .zip archive file) must be uploaded to Canvas by **15:00, Thursday 22nd April 2021** (**Course Week 11 (Semester Week 31)).**

In addition, teams are required to produce a **video** (with commentary; max **5 minutes**) of their working system. The video should illustrate the working implementation of the functionality requested above. The video must be in MP4 format and must run in VLC software: check that it does by using the software available at https://www.videolan.org/vlc/index.en-GB.html. Please do not record your video in HD or 4K, but ensure that any on-screen text is legible. This video is required for review by the external examiner. The project cannot be marked if the video is not submitted or cannot be played by the assessors. The video should be uploaded to Canvas in Semester 2 Week 12, at the same time as the working system is uploaded.

B: Short PDF Report

Each team should produce a short **PDF** report to accompany their game.

Please note that this is a **technical report.** The main body of the report should consist primarily of the UML diagrams requested, along with use case descriptions and test plan extracts in tabular form. These are the ‘technical deliverables’. Commentaries should be brief, taking the form of just a few sentences (bulleted where appropriate) that emphasise the most notable features of the technical deliverables. *Please avoid extended prose narratives, especially those that express personal impressions or that paraphrase third-party sources. The main body of the report is limited to 20 pages.*

The main body of the report should include the following sections.

A ***Requirements Analysis*** section, comprising Use Case Descriptions and an accompanying a UML Use Case Diagram. **You need to produce just one use case diagram to represent the whole system.** The Descriptions and

Diagram should concentrate on the *main* sets of sequences of actions that will be realised by the system. Plan your game's behaviour so that it can cope if problems arise (e.g. what happens if two players enter the same name?), and in such circumstances make sure you have an appropriate alternative flow or an extending use case. In your Requirements Analysis, include your layout of the virtual 'board' on which your game is played (remember: this graphical representation is for guidance only; it will NOT be implemented as a GUI in this text-only system!) (Requirements Analysis: **30** marks)

A ***Realisation*** section, comprising a number of UML Sequence Diagrams with a brief written commentary. Your sequence diagrams should show how your software components make method calls to each other, and interact with the players, in order to realise the behaviour of a selection of important use cases described in the previous section: clearly identify the use cases for which you have chosen to show realisations. **Remember: any class that you show at the top of lifeline in a sequence diagram should also appear somewhere in your class diagram** – see below**.** (Realisation: **20** marks)

A ***Design*** section, comprising a UML Class Diagram, that describes the system components. The class diagram will correspond closely to the coded implementation of the game; it should show classes and methods that support the sequences of method calls described in the previous section. Again, provide a brief written commentary on your design, pointing out any instances of good design where you have considered questions of *maintainability* and *extensibility*. (Design: **20** marks)

A ***Test*** section, comprising a *sample* from your ***Test Plan*** with a brief description of your approach. (Your Test Plan may take the form of an Acceptance Plan, like the one shown in Chapter 12 of your lecture notes. Your *full* test plan and test strategy should be documented in *Appendix I* – see below.) (**Test**: **5** marks)

Adherence to ***Process*** *should* be documented in **appendices**.

Place your *full* ***Test Plan*** for the implemented system in *Appendix I* – along with your acceptance tests you may provide evidence of unit testing (for example, screen dumps showing that unit tests ran successfully). A set of weekly ***Team Minutes*** should be placed in *Appendix II* – use the Weekly Team minutes template provided on Canvas under *Activity Plan and Project*. GitLab version management software should be used to facilitate collaborative working: place evidence of the team's use of GitLab (e.g. a representative sample of screen dumps from GitLab's Activity record) in *Appendix III*. Other evidence of day-to-day project management (e.g. brief descriptions of backlog items and tasks, with estimates; screen dumps of sprint backlog tables and team/team-member burndown charts, project timelines, etc.) should be placed in *Appendix IV* (**Process**: **5** marks)

<u>The main body of the report (excluding appendices and the peer assessment) should not exceed **20 pages**. Individual team members should place their initials (e.g. [A.B.; C.D.]) next to the sections for which they were the principal authors.</u>

Once the team's GitLab space becomes available, authors (whether of code or documentation) should upload their work regularly to appropriate team folders.

Teams should meet and agree the **Peer Assessment** prior to the submission deadline and include this at the front of the PDF Report. The Peer Assessment form is available under Activity Plan and Project on Canvas

The Peer Assessment scores are used to calculate each individual student's mark from the raw mark that the assessor gives to the project. Teams should meet (using Microsoft Teams) in good time in order to conduct the Peer Assessment exercise.

As with the working system, the submission deadline for the PDF Report is 15:00, Thursday 22nd April 2021 (Course Week 11 (Semester Week 31)). Submit the report to Canvas. See the separate <u>Activity Plan</u> also.

<u>Electronic assessments</u>
There will be two short **electronic formative feedback exercises** during the semester. These exercises **do not** count towards the final module mark, but will test basic UML for representing *requirements analysis (use case diagrams*

and descriptions) and *system design (class diagrams and sequence diagrams)* – the same elements of the UML that you will be using to document your property game, during the semester and in the final *PDF Report*. The exercises will take place on **Tuesday 2nd February 2021** (Course Week 3 (Semester Week 20)) and **Tuesday 23rd February 2021** (Course Week 6 (Semester Week 23)). **See the separate Activity Plan also.**

In summary…

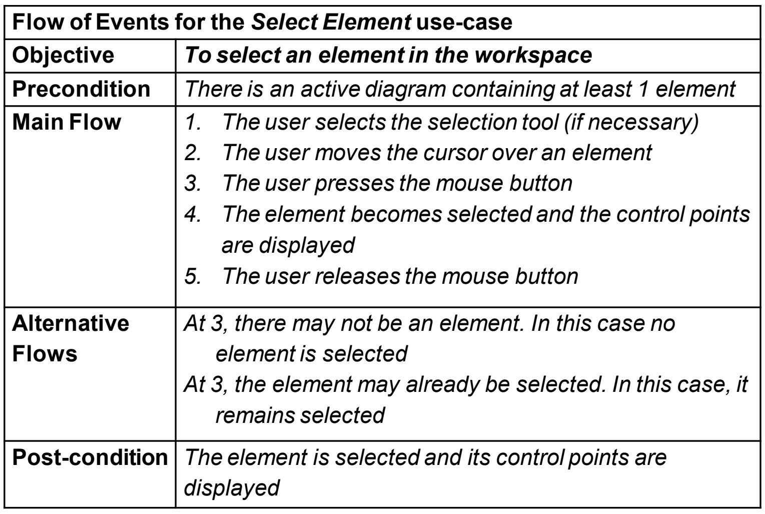
What is required?	*What is assessed?*	*Marks*	*When?*
Video Demo & Code	Working functionality	20	15:00, Thursday 22nd April 2021
PDF Report	Requirements Analysis (incl. game layout)	30	15:00, Thursday 22nd April 2021
	Realisation	20	
	Design	20	
	Testing	5	
	Adherence to Process	5	
Peer assessment	Participation in group work		15:00, Thursday 22nd April 2021
	Total	**100**	
Electronic feedback 1 Electronic feedback 2	UML Use Case Diagrams and Descriptions UML Class and Sequence Diagrams	- -	13:00 Tuesday 2nd February 2021 13:00 Tuesday 23rd February 2021

PTO for some UML samples…

Some samples of UML notation

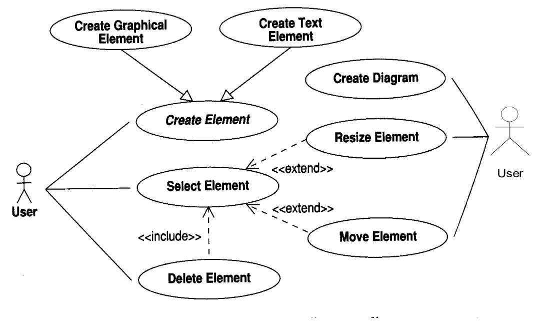
*It is important to look at the **full module notes** and **recommended texts** in order to appreciate the variety of ways in which the UML notation and accompanying descriptions may be used. The descriptions and diagrams below are working samples only and do not represent a full solution. Choose use case names, write use case descriptions, and create classes and objects that suit the software you are developing – your diagrams and descriptions will be different from the examples shown below!*

e.g. From Chapter 4 of the Module Notes:



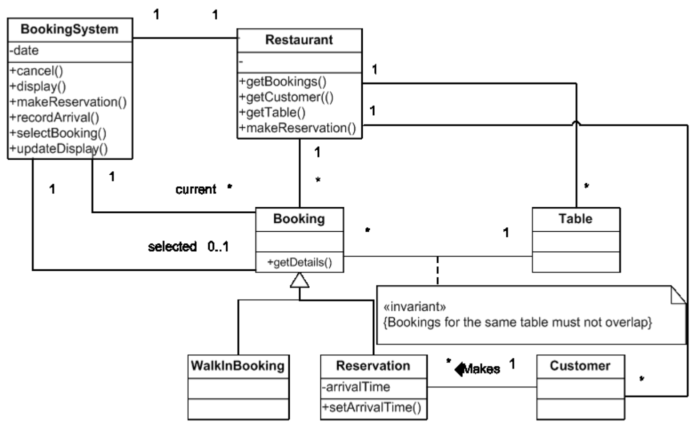
Flow of Events for the *Select Element* use-case	
Objective	***To select an element in the workspace***
Precondition	*There is an active diagram containing at least 1 element*
Main Flow	*1. The user selects the selection tool (if necessary)* *2. The user moves the cursor over an element* *3. The user presses the mouse button* *4. The element becomes selected and the control points are displayed* *5. The user releases the mouse button*
Alternative Flows	*At 3, there may not be an element. In this case no element is selected* *At 3, the element may already be selected. In this case, it remains selected*
Post-condition	*The element is selected and its control points are displayed*

A Use Case Description

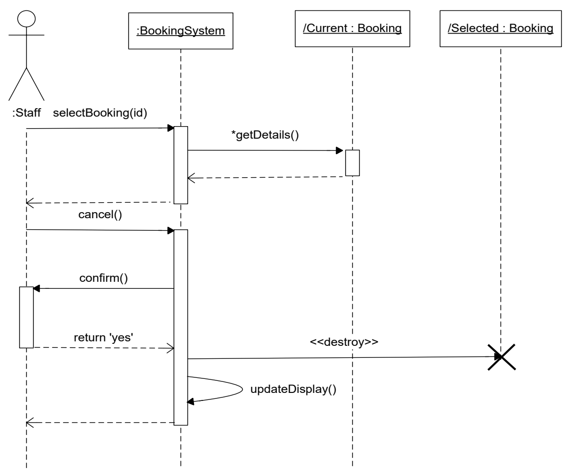


A Use Case Diagram

e.g. From Chapter 5 of the Module Notes (though this chapter is about Analysis, class and sequence diagrams are used to document design too!):



A Class Diagram



A Use Case Realisation (Sequence Diagram)