Software Engineering Project: ***Technopoly***

Deliver a well-designed and well-documented working system that satisfies the customer's requirements.

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. The game will have something of the flavour of the well-known board game *Monopoly* from Hasbro/Parker. However, your game, while it should take its inspiration from *Monopoly*, will definitely not be a straight copy of the original. **It will also have its own distinctive theme, drawn from the world of computing and informatics!** Instead of buying and developing properties and charging rent as you do in the traditional game, think of taking over different fields of computer technology, developing products in different areas, and generating resources from them or devoting resources to them. Rather than use cash for your transactions, you might want to think about allocating 'human resources' – investing in an area from your own pool of talent, and acquiring new talent from other companies. It's your job to think through and apply the 'metaphor' you want to use in your game.

Furthermore, your game 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). 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

[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 will not necessarily be one of the use cases in your *Technopoly* 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 much more information 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. Remember also that, apart from generalisation, the only relationships that are shown on the diagram between use cases are <<extend>> and <<include>>: these relationships imply that one use case is extended by another or includes another as actions take place **in real time**. If a use case must have occurred at some time in the past before another use case can occur, then the first use case is a precondition of the second: this information is conveyed in the use case descriptions, not in the diagram! On the **diagram** it is possible to have a use case that has a connecting line to an actor but that does not have an arrow from or to another use case.

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.

A game of this kind might start and unfold in the following manner (the example is not taken from *Technopoly* – it's more like a very restricted version of traditional Monopoly – 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

Janet, you've rolled a three and a two, which makes five.
Janet, you've landed on Square X
Janet, do you want to invest in this (y/n) ? y

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

Square administered by Janet: Square X

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

The game is to be conducted **in natural language only** (i.e. English phrases that convey the state of play, etc**.).**

A separate 'game layout' is required: submit this as a draft in the Semester 1 Week 11 PDF Report and as a Final Version in the Semester 2 Week 11 PDF Report. The game layout should be very, very simple – like the example below, though yours may have more squares. 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. 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 *Technopoly*, 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.

Investment Fund Collect X_I points	Square X Colour Brown Sq costs X_X Dev costs Y_X Maj Dev costs Z_X	Square Y Colour Brown Sq costs X_Y Dev costs Y_Y Ext Dev costs Z_Y	Square A Colour Blue Sq costs X_A Dev costs Y_A Ex Dev costs Z_A	Square B Colour Blue Sq costs X_B Dev costs Y_B Ex Dev costs Z_B	Square C Colour Blue Sq costs X_C Dev costs Y_C Ex Dev costs Z_C

Your game will have many of the features of a board game. 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'. 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), 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' *Technopoly* 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 *Technopoly*.]

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

Even a relatively simple *Technopoly* game is likely to be underpinned by quite a complex object mechanism, a suite of interacting software objects that have attributes and behaviour. The game has up to eight players, and their names should be entered.

The players take turns, they throw 2 virtual dice, and they land on squares.

Players are told where they have landed and what their obligations or opportunities are. Where appropriate, they may indicate their choice of action.

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' 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 – whether people or money, for example – 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 a number of 'fields'. Some 'fields' have three 'areas' and some have only two 'areas'.

A number of related areas form a 'field' in *Technopoly*. You decide what the fields are called and what they represent. For example, Artificial Intelligence (whether as an academic discipline or an area of product development) might be a 'field'. 'Fields' are equivalent to the colour groups in Monopoly. Different areas make up a field: Dialogue Systems (again, whether it represents a research area or families of commercial products) might be an area in the field of Artificial Intelligence (say it out loud, and see if it makes sense!) You decide what the technology-related fields are in your game, and what areas they will include.

One of the two-area fields is the most expensive field on the board to manage and resource; another two-area field is the least expensive field to manage and resource.

As soon as you acquire an area, other players must give you a contribution if they land on it. However, before you can develop an area within a field, you must own/manage/'be in charge of' (you decide what 'ownership' means!) the *whole* field – and on your turn you can develop an area in a field that you already own even if you are not positioned on that area.

You develop an area by building the equivalent of a 'house' on it: you decide what a 'house' is called, what it represents and how much it costs in your game. It might be something in the physical world or something from the 'knowledge economy'; others might have to pay money to use it, or provide people to help develop it further. Again, you decide the nature and the significance of the transactions between players.

Three 'houses' are needed before you can…

establish (and pay for, or otherwise 'resource') the *equivalent* of an 'hotel' – again you decide what this represents and what it costs. (Remember – we're using the 'house/hotel' terminology from Monopoly only so that, if you've

[2] For example, if in a demo you are asked 'Show me how managing and resourcing an area has a cost', and you have called areas 'research specialisms' and your cost 'staff', then you should be able to show how managing a 'research area' requires 'staff'.

played Monopoly before, you can more easily understand the underlying principle: your game will not be dealing with building domestic accommodation or accommodation for tourists!).

Players taking a turn are told where they have landed and what their obligations or opportunities are.

Where appropriate, they may indicate their choice of action.

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' or 'people' that are still available).

Not only is there a cost associated with acquiring and developing areas within fields: when you land on an area, but do not 'own' it yourself, you have to give up some of your resources for it – the more developed the area, the greater the resource required.

Decide what happens when one player runs out of resources. Are they the winner or the loser? It's your choice. If one player runs out of resources, or if one player no longer wants to play, the game ends. In both cases (no resources left; not wanting to play), the final state of play is shown. Find forms of words that express the outcome of the game in a manner appropriate to the overall style of your version of *Technopoly*. Is winning a matter of vanquishing the opposition, or might you be 'taking over' from fellow players, giving them the opportunity to 'take a well-earned break'; do you win by amassing resources from other players, or should our *Technopoly* hero be aiming to give all that they have (and more) to support the worthy ventures of a fellow philanthopist? You decide.

■■

Functionality that satisfies the requirements between the dashed lines (■■■■■■) represents a good **basic** *Technopoly* game. Within the constraints of the text user interface, **value-added** features may include but are not limited to:

- delivering the required functionality in a manner that demonstrates excellent usability, including clear, timely and engaging interaction with the players;
- a novel and coherent interpretation of the *Technopoly* theme that is very different from buying/selling/property trading;
- many and varied 'areas' and 'fields';
- special squares relevant to the theme;
- 'surprise' costs and gains (and remember, these may not necessarily involve money!);
- suitable celebrations when the game ends and a player is honoured by his/her peers.

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!

Deliverables

(See the table on p. 10 and the separate Activity Plan for detailed delivery dates and times, and see the General Instructions at the end of this document!)

Once GitLab repositories are allocated, teams should use appropriately named folders in GitLab to co-ordinate their work and share their contributions to the deliverables. Finished deliverables are uploaded to Canvas for assessment.

Semester 1 Week 11: PDF Report 1 – The Problem and the Early Solution; Interim Demo; Peer Assessment 1 (Total: 20%)

The main body of the report (excluding 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.

This report covers ***Use Case Requirements Specification and Planning***, ***System Analysis, and Peer Assessment 1.*** It should include…

1. **Use Case Requirements Specification and Planning (10%)**

a) **A use case diagram** representing the main sets of sequences of user-system interaction. (**See the important footnote (1) previously.**)

b) A corresponding set of **written use case descriptions** (**See that important footnote (1) again.**). Make sure that the name of a use case (in exactly the same form as it appears in an ellipse in the use case diagram) is placed at the top of each description – see the example below.

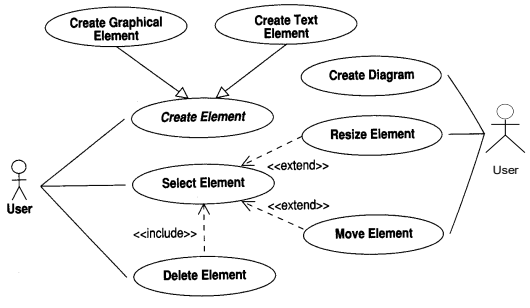
c) A **Gantt chart** indicating the main development strands and deliverables over the whole project lifetime. This should be easy to read at-a-glance.

*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 description and diagram shown below are working samples only and do not represent a full solution.* ***Choose use case names and write use case descriptions that suit the software you are developing – which will be different from the example below!***

From the 'Requirements' chapter of the Module Notes:

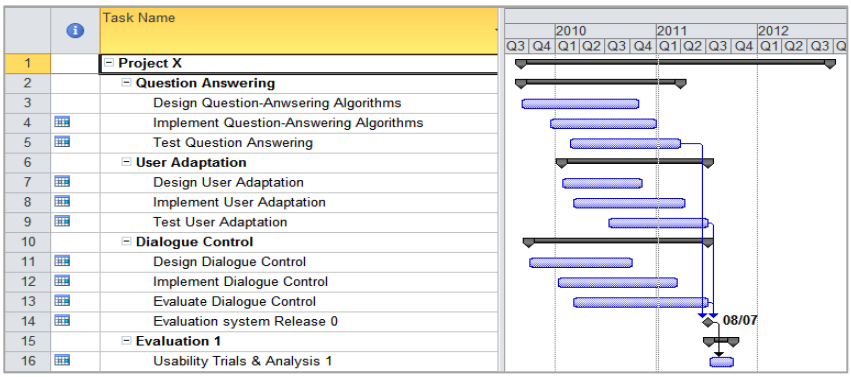
Flow of Events for the *Select Element* use-case	
Objective	***To select an element in the workspace***
Precondition	*There is an active diagram containing at least 1 element*
Main Flow	*1. The user selects the selection tool (if necessary)* *2. The user moves the cursor over an element* *3. The user 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 sample use case description



A sample use case diagram

From the 'Software Project Management' chapter of the Module Notes – a sample Gantt chart:



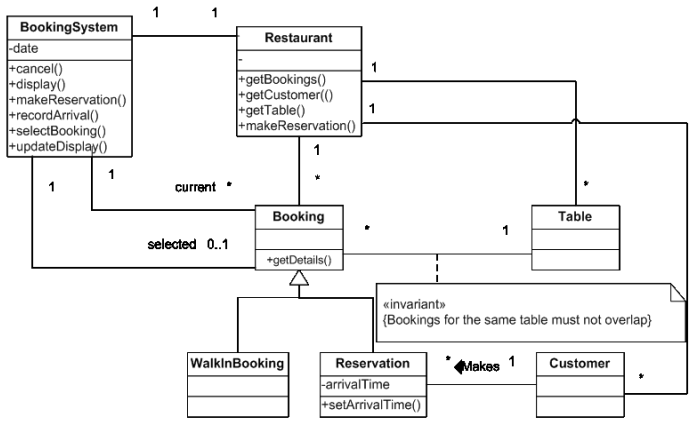
2. System Analysis (10%)

a) An initial **class diagram** (analysis model) representing the most important concepts in the application domain. These concepts may also be regarded as candidate classes for the subsequent design and implementation. They will be derived from the Use Case Requirements Specification. Before you use a particular style of line or arrow in your diagram, make sure that you understand what it means in the UML (in general, follow the examples in your module notes rather than simply use default line or arrow styles provided by a drawing package). Include a brief written commentary that describes the main features of your analysis model.

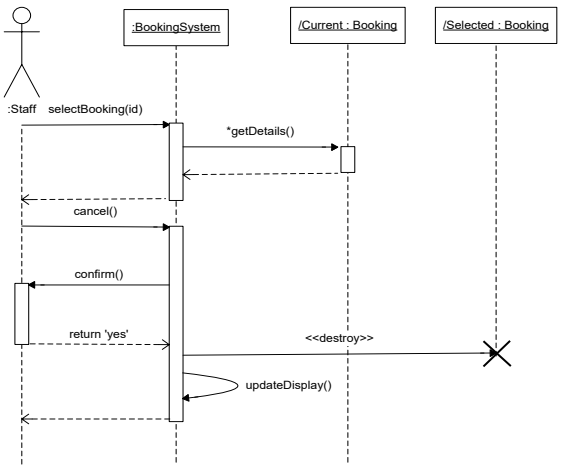
b) **Use case realisations** in the form of sequence diagrams that show the main sets of sequences of interaction identified in the Use Case Requirements Specification. In the title of each sequence diagram, make very clear which use case(s) – give the exact use case name(s)! – are realised by a particular sequence or set of sequences. Do not attempt to show the realisation of the whole system in a single diagram. Each sequence diagram must clearly match a named use case or use cases. Include brief written commentary on the use case realisations.

c) Your **draft game layout** (see above) – this is not assessed separately, but if you don't include it, it will make your Requirements Specification, Planning and System Analysis harder to understand and your scores for these sections will be adversely affected.

*e.g. From the 'Analysis' Chapter of the Module Notes (again, look at the **full module notes** and **recommended texts** in order to appreciate the variety of ways in which the UML notation can be used to represent relationships between classes and sequences of calls between instances):*



Class diagram (analysis model)



Use case realisation (sequence diagram)

3. Interim Demo (-5% if no demo is submitted)

In Semester 1 Week 11 all teams must submit a 5-minute video demonstration (with commentary) of the main features of their evolving solution. This should **not** be a video that simply shows and comments on the diagrams in the PDF report. **Ideally the video will show simulated outputs or working coded fragments in action.** At this stage fully implemented components and complete systems are not expected. **At very least, teams should be able to present the main concepts behind their documented solution so that the reviewers have a strong impression of the form that the eventual solution will take.** This video demonstration will not be formally assessed. It is, however, an opportunity for teams to obtain feedback on how their work is progressing. **A penalty of -5%** will be applied to teams who fail to submit a video demonstration.

Keep agreed team minutes each week and include these as an Appendix to the Semester 1 Week 11 PDF report that you submit to Canvas. A Weekly Team Minutes template is available. Groups may submit to Canvas any prototype code that they have developed in the course of compiling their report.

4. Peer assessment for the Problem and Early Solution

Teams should meet and agree the content of their Peer Assessment 1 Form prior to the submission deadline and include this at the front their Semester 1 Week 11 PDF report. The Peer Assessment 1 form is available under Activity Plan and Project on Canvas. Please read the instructions on the form carefully.

The Peer Assessment 1 scores are used to calculate each individual student's mark from the raw mark that the assessor gives to this deliverable. Teams should meet (in person if it is safe to do so, or via Microsoft Teams) in good time in order to conduct the Peer Assessment exercise.

(P.T.O. for Semester 2 Week 11 deliverables…)

Semester 2 Week 11 – The System and Video Demo; the Final PDF Report – Design, Implementation and Process; Peer Assessment 2 (Total 80%)

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.

A: The Working System (20%)

The working system is to be developed in Java. **A zip file representing the full working system must be uploaded to Canvas**. A **5-minute video demo (with commentary)** of the working system **must also be uploaded to Canvas**. Plan your video, if necessary recording it on different computers at different stages of the game, so that you show all the basic working functionality and your most important value-added features in action.

a) A system that delivers *basic* functionality similar to that described between the broken lines (■ ■ ■ ■ ■ ■ ■ ■) in the *Group Product Description* will attract up to **10%**. As a rule of thumb, a system whose functionality is significantly less rich than the basic functionality described might be awarded 3%; a good basic system with slightly less complexity than was described might receive 6%; and a system whose basic behaviour is closely comparable in complexity to the functionality described would be worth 10%. Marks for basic functionality may be adjusted in the light of usability.

b) *'Value-added'* features (features that were not covered by the description of basic functionality) will attract up to 10 further percentage points - **10%** of the module mark. Some suggestions for value-added features are indicated in the Requirements section above. Again, by way of guidance, 'not many' extra features might be awarded an additional 3%; 'a good selection' of extra features might receive 6%; while 'lots' of extra features would be worth 10%.

B: PDF Report 2 – The Final PDF Report

1. Design Documentation (20%)

a) A documented design of the **text user interface** comprising sample prompts, messages, status summaries, etc. Include a brief written commentary describing the purpose of each of your examples and why you gave it the form shown.

b) A **class relationship model** and **sequence diagrams** (see also Semester 1 Week 11 above) to represent the most significant features and behaviours of the **final** system. This is likely to be an **update to** or **revision of** the material you submitted in Semester 1 Week 11). It should reflect the changes to the design that you made in the course of the project. Include brief written commentary. (You are **not** required to submit a new set of use cases and descriptions for the Semester 2 deliverable. However, if, in the course of implementing your system, your original use cases have changed significantly, or if you have developed important new use cases, you may include an updated use case diagram and descriptions as an Appendix to your report.)

c) Your **final game layout** – again, this is not assessed separately, but if you don't include it, it will make your Design Documentation harder to understand and your score for this component will be adversely affected.

2. Implementation-Related Documentation (20%)

A **Test Plan** based on the original Use Case Requirements Specification. The Test Plan should describe tests for all key elements of system functionality (both 'basic' and 'value-added') and should show how the system deals with valid and invalid input. The Test Plan should describe the context in which each test is performed, the data that are input for the test, the precise output that is expected from the test, and the actual output that the test produces, so that it is clear that the test has been applied and that the system has passed. If the

system fails a test but later passes after corrective action, link the initial and subsequent tests in your documented testplan.

Testing may include a combination of black box testing (from a customer's perspective), white box testing (incorporating the developers' knowledge of conditional paths through the code) and automated unit testing (see also Adherence to Process below). If your full test plan would cause the main body of your report to exceed 20 pages, include a sample of your test plan in the main body of the report, and document the rest of your testplan in an Appendix.

You'll find a Sample Acceptance Plan in the 'Test – Software Verification' chapter of the module notes. This is particularly suitable for documenting black box testing and may be adapted to suit your team's particular style of testing. For documentation of automated (JUnit) testing, see Adherence to Process below.

3. Adherence to Process (20%)

Keep agreed team minutes each week and include these as Appendix 1 of your Final PDF Report – please use the Weekly Team minutes template provided on Canvas under Activity Plan and Module Deliverables.

<u>Brief</u> material representing further evidence of adherence to process (see below) should be included <u>as additional Appendices</u> of your Final PDF Report. (Any tests not documented in the main body of the dissertation may be documented in an Appendix.)

Adherence to process will typically include (but is not limited to):

- good project management
- well implemented classes that match the documented design
- version-managed code that indicates participation across the project team;
- appropriate JUnit testing with appropriate code coverage.

Evidence will typically include (but is not limited to):

- the submitted, implemented code;
- a populated Gitlab repository

and, as Appendices to the Final PDF Report:

- copies of team minutes (Appendix 1);
- a representative sample of screen dumps from GitLab's Activity record;
- screen dumps of key JUnit test runs and screen dumps showing code coverage (all implemented JUnit tests should be included in your submitted code; the code for the JUnit tests does **not** have to be included in your report also);
- comments (with reasons) on the development processes selected or indeed on approaches to development that were rejected.

Credit will also be given to teams that have implemented secure system features (in an Appendix you may give the relevant code snippet(s)); or that have identified appropriate future opportunities to implement secure system features (in an Appendix you may describe what they would be). For example, if your system were to become available to download online, or perhaps was used to support an online community of players, what secure features would you expect to implement? You don't have to discuss this extensively. However, this is an opportunity for you to show that you are aware of the security issues that might have to be tackled in a system like yours – either now, or in the future. Check the security-related lecture notes, and reliable on-line resources for guidance on best practice.

4. Peer assessment for the System, the Final PDF Report and the Process

Teams should meet and agree the content of their Peer Assessment 2 Form prior to the submission deadline and include this at the front their Semester 2 Week 11 PDF report. The Peer Assessment 2 form is available under Activity Plan and Project on Canvas. Please read the instructions on the form carefully.

The Peer Assessment 2 scores are used to calculate each individual student's mark from the raw mark that the assessor gives to this deliverable. Teams should meet (in person if it is safe to do so, or via Microsoft Teams) in good time in order to conduct the Peer Assessment exercise.

General Instructions for PDF Reports

Reports are submitted in Portable Document Format (PDF). Upload a PDF of the final version of each report to Canvas.

Ensure that the group name and the group number (Group 1 etc.) are clearly shown on the cover page of your report. Each report should be in A4 format, except where a diagram or chart requires a larger page size for greater legibility. For the main text, use at least point-size 10 and a standard typeface such as Calibri or Times New Roman. Do not hand-draw charts or diagrams. Choose content carefully to convey the most important aspects of your system and its development process: excluding the peer assessment form and the appendices requested above, **each report should not exceed 20 pages**.

In summary…

What is required?	*What is assessed?*	*Marks*	*When?*
Semester 1 PDF Report	Use Case Reqs. Specification and Planning	10	15:00, Friday 3rd December 2021
	System Analysis (incl. draft game layout)	10	
Semester 1 Interim Video Demo (optionally code may also be submitted)	Simulated outputs or working coded fragments	(-5 if not submitted)	
Semester 1 Peer Assessment	Participation in group work		
	Subtotal	*20*	
Semester 2 PDF Report	Design Documentation	20	15:00, Friday 25th March 2022
	Implementation-Related Documentation (Test Plan)	20	
	Adherence to Process (Team Minutes; screen dumps of GitLab Activity records, JUnit test runs, coverage analysis; etc.)	20	
Semester 2 Video Demo & Code	The working system	20	
Semester 2 Peer Assessment	Participation in group work		
	Subtotal	*80*	
	Total	**100**	