**Preliminary client requirements**

The client desires a grand-strategy game, set in the time of Ancient Rome (around 200 BC), where the player can play as a faction of the time (such as the Romans, Carthaginians, Gauls, Celtic Britons, Spanish, Numidians, Egyptians, Seleucid Empire, Pontus, Amenians, Parthians, Germanics, Greek City States, Macedonians, Thracians, Dacians), with the overall goal of conquering all provinces in the game map (or succeeding at another grand victory objective).

The game should be a turn-based military/economic strategy game - each faction engages in actions during sequential "turns", where they perform all actions to manage their armies and provinces for a single year in their turn, before clicking "End turn", which will result in all other factions performing their turns sequentially.

The basic unit of currency in the game will be "Gold". Each building and unit of troops has a price which must be paid in gold upfront, without going into debt (unless implementing the entirety of the lending and bankruptcy mechanic extensions).

**Movement of Soldiers**

Factions should be able to move troops between regions and attack neighbouring enemy regions. The type of unit and the development of roads in provinces will affect the number of regions through which troops can move in a single turn.

Factions should be able to attack multiple enemy provinces in a single turn, with the following restrictions:

· To attack multiple enemy provinces in a single turn they must use different instances of units

· Factions cannot "attack through" a settlement they conquered in the current turn (i.e. if they conquered settlement A in the current turn, they cannot move troops to settlement A and use them to attack settlement B directly from settlement A)

If troops have been used to invade a province, or are moved into a province invaded in the current turn, they cannot be moved for the rest of the turn.

Movement points are expended for every province passed through from the starting province, to the destination province. The starting province is included in this calculation, whilst the destination province is not included.

Units cannot move through enemy territory. Units should automatically move by the shortest path between non-adjacent provinces (shortest path is defined as the path which will expend the fewest movement points).

If the player attempts to move units to a province further than the movement points allow the units to be moved in the turn, or is unable to move the units from the starting province to the destination province, the attempt to move the units should be rejected.

By default, the number of movement points available to a unit each turn depends on its category:

· Cavalry = 15 movement points

· Infantry = 10 movement points

· Artillery = 4 movement points

The number of movement points used moving across a province should depend on the level of roads, as follows:

· No roads = 4 movement points

· Dirt roads = 3 movement points

· Paved roads = 2 movement points

· Highways = 1 movement point

Only the Romans should be able to build highways.

The player should be able to select several units for movement from a province to another province. When multiple units are moved at the same time, the maximum distance they can be moved during the turn together should be limited by the number of movement points of the unit with the fewest movement points. Troops should lose movement points at the unit's normal rate regardless of whether they were moved with other units.

For example, if a unit of infantry with 10 movement points was moved with a unit of artillery with 4 movement points, and moving across 2 provinces used 4 movements points for both units, the artillery would have used all of its movement points and be unable to move for the duration of the turn. However, the infantry unit would be able to be selected independently and moved further with its remaining 6 movement points.

**Infrastructure**

Factions should be able to develop infrastructure in the following categories:

· Troop production buildings - factions should be able to recruit heavy infantry, spearmen, missile infantry, melee cavalry, horse archers, elephants, chariots, and artillery. Different categories of soldiers will require different "chains" of buildings.

· Wealth generation buildings - factions should be able to construct markets, farms, ports (if on a region bordering the sea), and mines. Each building adds a scalar value to the wealth of the region, and adds a scalar value to the before-tax rate of town-wealth growth of the region each turn, at different rates. In addition, the following additional effects occur:

o More advanced farms increase the rate at which troops can be produced each turn in the province

o Each port increases the before-tax rate of town-wealth growth for all sea regions owned by a province by an additional scalar value

o Each market reduces the construction cost of all buildings across the faction in a multiplicative fashion - i.e. if one market reduces construction costs by 2%, and another by 3%, and the current construction cost of buildings in the faction this turn is *C*, then the new construction cost of buildings in the faction this turn will be *C x (100%-2%) x (100%-3%)* = *C x 95.06%*

o Each mine reduces the initial cost of all soldiers in the province in a multiplicative fashion (similar calculation to markets). The highest level of mine will reduce the time taken to construct all buildings across the faction by 1 turn (the minimum time to build a building, even after applying this affect several times, is 1 turn). This bonus can be applied multiple times - e.g. building the most advanced mines in 2 settlements will reduce the number of turns taken to build all buildings across the faction by 2 turns, down to a minimum of 1 turn).

· Walls - these provide buffers to all troops defending a province. They have a random chance of causing damage to attacking troops if upgraded to have archer towers or ballista towers. Towers have infinite morale, and can only be damaged by artillery (but are instantly repaired without cost after the battle).

· Smiths - these provide battle bonuses to troops produced in the region. Where both scalar addition bonuses (e.g. +1 attack damage) and multiplicative bonuses (e.g. 20% loss in speed) apply due to receiving multiple armour upgrades, the scalar addition bonuses are applied first. Smith bonuses can include 1 or more of the following (these bonuses stack):

o Upgraded helmets - enemy unit attack damage reduced by 1 (to a minimum of 1 attack damage)

o Upgraded armour suit - enemy unit attack damage (ranged and melee) reduced by 50% (to a minimum of 1 attack damage), speed of the soldiers wearing the armour reduced by 20%

o Upgraded weapon - wielding units attack damage increase by 20%. Morale of the wielding unit increased by 10%

o Fire arrows - archers shoot arrows reducing morale of enemy soldiers in a skirmish with them by 20%, but lose 10% of missile attack damage (to a minimum of 1 missile attack damage)

· Roads - these reduce the rate at which movement points of soldiers are used up.

A building "chain" is a sequence of buildings, where each subsequent (more advanced) building requires the prior building to be built before it can be constructed. Later troop production buildings will allow more advanced troops to be recruited, later wealth generation buildings/walls/smiths/roads will amplify the effects of the building. Each province can only have up to 1 building per building chain at any time - so building a more advanced building will replace the existing building for it's building chain. For example, building a level 2 farm in a province replaces the level 1 farm existing in that province.

Factions should be able to recruit soldiers in each province each turn (if they have enough money). The number of soldiers they can produce in each province will be increased by more advanced farm buildings. More advanced troop production buildings will allow production of different types of soldiers.

If a province is conquered whilst constructing a building, construction is cancelled without refund of gold to the faction which paid to produce the building.

Each province can only work on constructing 1 building at any time. Each building type has a defined base number of turns which it requires to build (this can be reduced by building the most advanced mine building in a settlement.

**Soldiers**

All soldiers have a base-level initial training cost (which must be immediately paid by the player). If soldiers are killed in battle, or disbanded, the affected soldiers are removed from the game, without refund to the player. Soldiers do not incur upkeep costs.

Soldiers who have finished training appear at the *beginning* of the turn they are due to finish training. For example, if the user selects to recruit a unit of soldiers taking 1 turn to train at turn 2, the soldiers will be available at the start of turn 3.

If a province is conquered whilst troops are being trained, the troop production is cancelled and no refund is made to the player.

Each province can initially work on training 1 unit per turn, but upgrades to farms can increase the number of units which the settlement can train per turn. Some units (e.g. elephants) take 2 or more turns to produce (these units must occupy a training slot for the required number of turns - having spare training slots doesn't allow the training to be sped up).

All soldier types can be recruited as mercenaries (except artillery). The name of all mercenary soldiers is their normal unit type name, prefixed by "Mercenary " (e.g. "Hastati" as mercenaries would be "Mercenary Hastati"). Availability of mercenaries is determined by random-chance on a turn-by-turn basis, and resets every turn (i.e. mercenaries don't accumulate into an increasing pool of available units). If the user selects to recruit a mercenary unit, the mercenaries can be immediately deployed to any province on the map the user chooses (upon payment of the initial cost).

Mercenaries generally suffer from higher initial purchase cost, and penalties to combat ability. This differs per unit type as follows:

· Horse archer mercenaries incur double the initial recruitment cost. Each type of horse archers has a 3% chance of being available per turn.

· Melee cavalry mercenaries (including heavy cavalry, lancers, elephants, chariots) incur triple the initial recruitment cost. They suffer from a 20% loss of morale. Each type of melee cavalry has a 5% chance of being available per turn.

· All infantry mercenaries (including missile infantry, heavy infantry, and spearmen) incur double the initial recruitment cost. Each unit of mercenary infantry reduces the morale of all friendly units in the province by 10% (stacks in a multiplicative fashion). Each type of heavy infantry has a 10% chance of being available per turn.

Soldiers are usually restricted to certain factions for province-level recruitment, and available depending on the level of infrastructure. However, all mercenary units can be recruited by any faction, without requiring infrastructure to obtain them.

**Soldier Special Abilities and Combat Statistics**

The following special abilities should be available and implemented automatically.

· For all Roman legionary units: "Legionary eagle" - provides +1 morale to all friendly units in the province. For every legionary eagle lost to the enemy (by the unit being destroyed defending a province) all friendly units across all provinces suffer a 0.2 penalty to morale until the settlement is recaptured (down to a minimum of 1 morale)

· For all Gallic/Celtic Briton/Germanic berserker units: "Berserker rage" - unit receives infinite morale and double melee attack damage, but has no armor or shield protection, in all battles

· For all melee cavalry: "Heroic charge" - where the army has fewer than half the number of units as the enemy, this cavalry unit will double its charge attack damage, and have 50% higher morale

· For all pikemen or hoplite units: "Phalanx" - these hoplites or pikemen have double the melee defence, but half of the speed, as they are otherwise configured to have

· For all javelin-skirmisher units: "skirmisher anti-armour" - in ranged engagements, troops fighting these skirmishers only receive half the protection from armour they would receive otherwise

· For all elephant units: "Elephants running amok" - during any engagement with elephants, there is a 10% chance that the casualties inflicted by a unit of elephants will instead be directed at a random allied unit (as if the elephants were battling the allied unit directly)

· For all horse-archer units: "Cantabrian circle" - when enemy missile units engage this unit of horse archers, the enemy missile units will suffer a 50% loss to missile attack damage

· For all druid units: "Druidic fervour" - allied units in an army with druids enjoy a 10% bonus to morale, and enemy units suffer a 5% penalty to morale, whilst the druids haven't routed. The effect of this ability is amplified by scalar addition, and can be amplified up to 5 times (e.g. 2 druids results in allies receiving 20% bonus to morale and enemy units suffering 10% penalty, however 6 druids provides 50% bonus and 25% penalty respectively)

· For all melee infantry: "Shield charge" - for every 4th engagement by this unit of melee infantry per battle, the value of shield defense is added to this unit's attack damage value

Note that mercenaries receive the same bonuses as non-mercenaries.

These bonuses are applied similarly to smith bonuses - scalar addition bonuses (for all bonuses e.g. smith, special bonuses) are applied before all multiplicative bonuses.

**Wealth and Taxes**

Province wealth represents the collective wealth of the people and infrastructure in the province. It is distinct from the treasury; treasury is the gold you spend on troops/buildings. Province wealth is composed of town wealth (from the accumulation of turn-by-turn wealth generation) and building wealth (i.e. the base wealth provided by mines, farms, markets, ports).

Wealth generation buildings increase base wealth immediately, and also increase the turn-by-turn town-wealth generation rate.

Wealth cannot be spent; it is taxed to produce tax revenue for the faction's treasury. The tax revenue generated is always equal to *province wealth x province tax rate*

The minimum wealth of a province is equal to the building wealth of the province (i.e. town-wealth has a minimum value of 0). This means that the minimum wealth of any province is 0 (for a province with no farms, mines, markets, or ports, and 0 town-wealth).

Taxes are managed at the province level by the player.

Higher tax rates reduce the growth of town-wealth in a region (and can make it negative).

The effects of different tax levels are:

· Low tax = +10 town-wealth growth per turn for the province, tax rate = 10%

· Normal tax = No effect on per turn town-wealth growth, tax rate = 15%

· High tax = -10 town-wealth growth per turn for the province (i.e. 10 gold loss to wealth per turn), tax rate = 20%

· Very high tax = -30 town-wealth growth per turn for the province, tax rate = 25%, -1 morale for all soldiers residing in the province

**Campaign Victory**

Basic goals in the game should include:

· Conquering all territories (CONQUEST goal)

· Accumulating a treasury balance of 100,000 gold (TREASURY goal)

· Building all possible infrastructure across all settlements (INFRASTRUCTURE goal)

· Accumulating faction wealth of 400,000 gold (WEALTH goal)

When starting a campaign game, a logical conjunction(AND)/disjunction(OR) of basic goals, or a conjunction/disjunction of other conjunctions/disjunctions should be chosen randomly (a uniformly random choice from all possible goal expressions where each basic goal exists in the expression at most once). For example:

{ "goal": "AND", "subgoals": [ { "goal": "TREASURY" }, { "goal": "OR", "subgoals": [ {"goal": "CONQUEST" }, {"goal": "WEALTH" } ] } ] }

This example of a victory condition would allow victory in either of the following scenarios:

· TREASURY and CONQUEST goals achieved

· TREASURY and WEALTH goals achieved

Upon reaching the victory condition, the game should be automatically saved, and the user presented with an interface congratulating them on their victory. Re-loading the automatic save should allow the user to continue playing the game without further victory prompts.

**Basic Campaign Game Interactions**

Upon starting/loading a game, the user should be presented with a game map and a menu/menus to perform actions. The game map should display a map of the world, split into historically accurate, clickable provinces (as in the starter-code). Each province should be allocated to a faction, and display the flag of the faction, the province wealth, the number of soldiers, and the faction name.

The user should be able to view the current game year and turn on the screen whilst playing the game. The user should be able to select to finish the turn by clicking an "End turn" button.

The user should be able to save their campaign game at any time (so that they can load it and resume gameplay later).

**Basic Campaign AI**

In this campaign AI, enemy factions should first try to spend up to 50% of their treasury balance on infrastructure. This AI will initially prioritize the cheapest buildings. If there are multiple buildings of the same price, it will prioritize by category of building in the following order:

· Wealth generating buildings, in the following order:

o Ports (if on a sea province)

o Markets

o Farms

o Mines

· Roads

· Troop production buildings

· Smiths

· Walls

No further ordering will be necessary, because buildings in a building chain will always have different prices (the later buildings are always more expensive).

After purchasing infrastructure, the AI will try to spend the remaining treasury balance on recruiting units. The AI will first prioritize recruiting recruitable units in the category with the fewest units across all provinces (with the categories being horse archers, cavalry, heavy infantry, spearmen, missile infantry, artillery). Where this is equal between categories, the following order of priority will be applied (from highest to lowest priority):

· Spearmen

· Heavy infantry

· Missile infantry

· Cavalry

· Horse archers

· Artillery

After choosing a unit category, the AI will prioritize recruiting the recruitable unit with the highest initial cost. Where this is equal, it will randomly choose a unit (uniformly random).

After choosing a unit, the AI will prioritize recruiting from the province closest to an enemy province (fewest adjacent regions to cross in the shortest path). Where this is equal, it will randomly choose this enemy province (uniformly random).

The AI will move its troops not currently at a border province to the border province with the fewest soldiers. If there are multiple border provinces with the fewest soldiers, the province with the most enemy soldiers in adjacent regions is prioritized. Where this is equal, the AI will move it's troops towards a randomly chosen province (uniformly random).

When moving soldiers, the AI will move all soldiers in a province at once to the aforementioned closest province (it doesn't split the army up). The AI will first move the troops from the non-border province closest (fewest adjacent regions to cross in the shortest path) to this destination border province (note, during different moves in a turn the border province with fewest troops may change, and the AI should update it's destination province as such). If there are multiple closest origin provinces, the AI will randomly choose a province to move troops from (uniformly random).

Similarly to the player, the AI automatically sends the units on the shortest path to the destination border province. Once the AI has picked a destination border province for some units, they will move to this province (potentially across several turns) until reaching the destination.

The basic campaign AI will first attack a neighbouring province with the fewest units. Where this is equal or it can attack with multiple armies, it will attempt to attack with its army with the most units. Where this is equal, it will attack from a random province (uniformly random).

The basic campaign AI never tries to recruit mercenaries, or disband troops. The basic campaign AI always applies a normal tax rate for all provinces.

**Basic Battle Resolver**

Battles are composed of a sequence of "skirmishes" between a single unit from each of the armies, which are run until an army is eliminated or routed entirely (towers are not counted for the purposes of checking if a defending army has been defeated). The army left standing is victorious. If the attacking army in an invasion is victorious, the province is conquered by the invaders.

Walls double the melee defence of all troops defending a settlement (except when fighting artillery), reduce the missile attack damage of attacking archers by 50% (to a minimum of 1), and reduce the missile attack damage of attacking horse archers to 1. ADDED TO INFRASTRUCTURE

At the beginning of a skirmish, a unit is randomly chosen from each of the armies (uniformly random). When an army is defending walls with towers (archer towers or ballista towers), the towers can be selected as a unit (and will have infinite defence, except against artillery). The number of towers which can be included as units will depend on how developed the walls are. Towers have infinite morale (they cannot route).

During a skirmish, both units engage in a sequence of engagements against each other until a unit **successfully** routes (runs away from the battle permanently) or is defeated. However, if an attacking unit skirmishes against a tower, the skirmish may end before the attacking unit routes or is defeated (by random chance).

In all engagements (with or without walls), if both units are melee units, there is a 100% chance of a melee engagement. In all engagements (with or without walls), if both units are missile units, there is 100% chance of a missile engagement.

If a battle lasts longer than 200 engagements, the outcome should be a draw. The invading army in a draw should return to the province it invaded from.

**The result of all engagements should be presented to the user via visual or text information.** A JavaFX TextArea is included in the starter-code UI to make this easier.

**Chance of ranged/melee engagement where there is a ranged and melee unit in the engagement**

In a battle **without** walls, if there is 1 ranged unit (artillery, horse archers, missile infantry) and 1 melee unit, there is a base-level 50% chance for the engagement to be a ranged engagement, and a base-level 50% chance for an engagement to be a melee engagement.

In a battle **with** walls, if there is 1 ranged unit (artillery, horse archers, missile infantry) and 1 melee unit, there is a base-level 90% chance for the engagement to be a ranged engagement, and a base-level 10% chance for an engagement to be a melee engagement.

The base-level chance of engagement to be a melee engagement (where the engagement has both a melee and missile unit) is increased by *10% x (speed of melee unit - speed of missile unit)* (value of this formula can be negative)

Where the engagement has both a melee and missile unit, the maximum chance for an engagement to be either a ranged or melee engagement is 95% in either case (except where a tower is in an engagement - all battles involving a tower will be ranged engagements).

**Tower engagements**

Against artillery, towers have 100 health points, and suffer a loss in health points for every engagement with an artillery unit (defined by the configurations for the artillery unit).

Note that the configurations for artillery units include a missile attack damage value for engagements with troops, and a missile attack damage value for engagements against towers.

If a tower loses all health points, it is disabled for the duration of the battle.

Against infantry or cavalry, towers inflict damage against the enemy unit in the same fashion as missile infantry (according to their configured statistics) without suffering damage.

Units fighting towers do not route, and have a chance of escaping from each engagement (returning to the army for another skirmish) of:

· minimum(50% + (speed x 10%), 100%) \*

**Inflicting of casualties in an engagement**

Melee units cannot inflict damage in a ranged engagement.

Ranged units in a ranged engagement inflict casualties against the opposing unit equal to (up to a maximum of the entire enemy unit, with a minimum of none of the enemy unit):

· (size of enemy unit at start of engagement x 10%) x (Missile attack damage of unit/(effective armor of enemy unit + effective shield of enemy unit)) x (N+1) \*

Where *N* is a normally distributed random variable with mean 0 and standard deviation 1 (standard normal distribution).

**HINT: check Random.nextGaussian() to obtain a random value from a random normal distribution:**

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html#nextGaussian()>

You should ensure the ranged attack damage above incorporates the effect of any bonuses/penalties (e.g. the 10% loss of missile attack damage from fire arrows).

Cavalry/chariots/elephants attacking a province with walls do not receive any melee bonus from their charge statistic. Melee cavalry/chariots/elephants defending a province with walls, or attacking a province without walls, will have an attack damage value in all engagements equal to their *melee attack damage + charge value*. Infantry and artillery do not receive a charge statistic (only cavalry/chariots/elephants do).

Units in a melee engagement inflict casualties against the opposing unit equal to (up to a maximum of the entire enemy unit, with a minimum of none of the enemy unit):

· (size of enemy unit at start of engagement x 10%) x (Effective melee attack damage of unit/(effective armor of enemy unit + effective shield of enemy unit + effective defense skill of enemy unit)) x (N+1) \*

Where *N* is a normally distributed random variable with mean 0 and standard deviation 1 (standard normal distribution).

Note that all effective attributes in the formula should incorporate the effect of any bonuses/penalties (such as blacksmith upgrades, formations such as phalanx formation, charge bonuses where applicable for cavalry/chariots/elephants).

Note that a Gaussian distribution can return negative values, therefore it is important that you apply the specified bounds to the minimum and maximum values of these formulas (maximum casualties is entire enemy unit, minimum casualties is none of the enemy unit). This is to prevent damage causing "healing", or the appearance of units with a negative number of troops.

**Breaking a unit**

Following the inflicting of casualties during each engagement, random chance may result in a unit "breaking" and attempting to flee the battle.

The base-level probability of a unit "breaking" following an engagement is calculated as:

*100% - (morale x 10%)*

For example, a unit of peasants with 3 morale would have a base chance of breaking after each engagement of: 100% - (3 x 10%) = 100% - 30% = 70%

Note that the morale value in the above formula should be morale after applying all campaign/battle morale adjustments (such as morale loss due to very high tax in the province, fighting an enemy unit with fire arrows, morale loss due to scary units in the enemy army).

The chance of fleeing the battle is increased by (a scalar addition):

*(casualties suffered by the unit during the engagement/number of troops in the unit at the start of the engagement)/(casualties suffered by the opposing unit during the engagement/number of troops in the opposing unit at the start of the engagement) x 10%*

For example, if our above unit of peasants suffered casualties equal to half of the units size at the start of the engagement, and the opposing unit suffered casualties equal to a quarter of its size at the start of the engagement, then the chance of fleeing the battle is increased by: (1/2) / (1/4) x 10% = 2 x 10% = 20% And thus the unit of peasants would have a 70% + 20% = 90% chance of breaking.

i.e. if your unit loses a larger proportion of it's soldiers than the opposing unit in an engagement, the chance of your unit to break is increased by a larger value than than the enemy unit from the base fleeing chance.

However, for any engagement, the minimum chance of fleeing the battle is 5%, and the maximum chance of fleeing the battle is 100%, after these calculations/adjustments.

If both units break, they both successfully flee the battle without inflicting further casualties upon each other.

**Routing**

If a unit "breaks" (and the opposing unit does not), it will repeatedly attempt to flee the battle (each failure resulting in an engagement in which the fleeing unit suffers casualties, without damaging the pursuing unit), until it is successful in doing so or is destroyed (units which break will not attempt to return to the battle).

If a unit breaks, there is a base-level 50% chance of it successfully routing for every attempt. The minimum chance a unit can have to successfully route is 10%, and the maximum chance it can have is 100%. The speed of the units in the engagement affect the chance of successfully routing, according to the following formula (but is superseded by the 10%/100% rule outlined above):

*chance of routing successfully = 50% + 10% x (speed of routing unit - speed of pursuing unit)*

For example, if the speed of the routing unit is 8, and the speed of the pursuing unit is 5, the chance of routing successfully in a single engagement is 80%. If the speed of the routing unit is 1, and the speed of the pursuing unit is 7, the chance of routing successfully in a single engagement is 10% (rather than -10%, since the minimum chance for a unit to escape is 10% per attempt).

Note that even if towers are destroyed by artillery, walls/towers will be repaired immediately after the battle at no cost to the faction owning (or conquering) the towers.

Note that if a unit successfully routes from the battle, if it is in an attacking army which loses the battle, it will return to the province from which it attacked, but if the attacking army wins the battle, the routing unit will join the army in moving into the new province. The entirety of a defending army which is defeated/routed is destroyed.

**Main Menu**

Upon starting the game, the user should be presented with a main menu.

From here, the user should be able to choose to start a "Campaign Game". The user should be able to select an option for a "Campaign AI", and a separate option for a "Battle Resolver", which will apply throughout the duration of the campaign. The user should at least be able to select the "Basic Campaign AI" and "Basic Battle Resolver", details of which have been outlined above. If you develop different campaign AIs/battle resolvers, the user should be able to select their preferred option. After selecting any/all relevant settings, the user should be able to start the campaign game.

The user should be able to load a previously saved game from the main menu (or a follow-on-menu).

**Miscellaneous Points**

All calculations, unless otherwise specified, should be rounded to the nearest integer in the result (e.g. if a market reduces the price of a building costing 70 gold pieces by 3%, the price of the building will be 68 gold pieces, rather than 67.9). However, the minimum training cost for troops, and minimum purchase price for a building, is 1 gold piece.

Each turn takes 1 year. Thus, in this spec, the terms "year" and "turn" are synonymous.

Bankruptcy is not possible in the game outlined in the basic specification because there are no recurring costs, and players can only spend money in the treasury (they cannot raise debt). Implementing an extension for upkeep would also require you to consider how bankruptcy would be handled.

When a faction loses all of its provinces, it should be removed from the game. If the player loses all of their provinces, the game should present a "Game lost" screen, from which the player can return to the main menu.

**The Client's Inspiration**

The client has gained some inspiration from the classic board game "Risk" (but has developed different rules and extensions to this):

<https://en.wikipedia.org/wiki/Risk_(game)>

The client is also a massive fan of the RTS strategy games "Rome Total War" and "Empire Total War":

<https://en.wikipedia.org/wiki/Rome:_Total_War>

<https://en.wikipedia.org/wiki/Empire:_Total_War>

The client has also gained some ideas from the game "Civilisation Revolution":

<https://civilization.com/en-AU/civilization-revolution/>

**Source of Map Data**

The map data provided in the starter code was obtained from: <https://github.com/klokantech/roman-empire>

The project uses the following files in this repository:

· *data/provinces.geojson* (province borders and province names)

· *data/provinces/label.geojson* (province label points and province names; the locations to place images/text for each province in the starter code are generated from here)

You can interact with an online version of the map at: <https://klokantech.github.io/roman-empire/#4.01/41.957/19.646/0/8>

The GeoJSON files downloaded from the repository were converted to the newer format for GeoJSON by applying the "Right hand rule" using: <http://mapster.me/right-hand-rule-geojson-fixer/>

This tool was then used to convert the GeoJSON files to GeoPackage files: <https://mygeodata.cloud/converter/geojson-to-geopackage>

GeoJSON files have the advantage of being JSON data and therefore easy to read using a text editor, or process with Java code using the provided geojson-jackson library (or you could use a standard JSON library such as JSON-Java (also provided, this is the JSON library you used in assignment 1)). However, it is inconvenient to load GeoJSON data directly into ArcGIS FeatureLayers because the ArcGIS runtime SDK for Java lacks a class to directly load in GeoJSON files. You can also view a GeoJSON map visually by using the VSCode extension "VSCode Map Preview".

Whilst GeoPackage files are hard to read using a text editor, it is easy to load GeoPackage files directly into ArcGIS using the GeoPackage class. This has been done in the starter code. The documentation for the GeoPackage class is at:

<https://developers.arcgis.com/java/latest/api-reference/reference/com/esri/arcgisruntime/data/GeoPackage.html>

You may apply different map data if you find a more complete digital map of the ancient layout of provinces, but we recommend to first focus on completing all required features.

**Province Adjacency Matrix**

An adjacency matrix of adjacent provinces has been provided in the file:

*province\_adjacency\_matrix\_fully\_connected.json*

Running the script *detect\_adjacent\_provinces.py* in the same directory as the file *provinces\_right\_hand\_fixed.geojson* produces a file *province\_adjacency\_matrix.json*. This file is a JSON file containing a nested dictionary structure, mapping all permutations of 2 province names to whether the 2 provinces are adjacent. Provinces have been set as not adjacent to themselves.

*detect\_adjacent\_provinces.py* uses the python3 shapely library (pip install shapely) to detect adjacent *MultiPolygons* (representing provinces) and stores the resulting adjacency matrix in the format:

{province1name: {province2name: whether\_adjacent}}

Where *province1name* and *province2name* are the names of provinces (strings) and *whether\_adjacent* is a boolean representing whether the provinces are adjacent.

We manually edited the file *province\_adjacency\_matrix.json* using a text editor, so that the provinces graph is fully connected. Particularly, we added that:

· Britannia connects to Lugdunensis

· Cyprus connects to Cilicia

· Sardinia et Corsica connects to VII

· Sicilia connects to III and Africa Proconsularis

· Baetica connects to Mauretania Tingitana

Running the file *detect\_adjacent\_provinces.py* performs a check at the end that the provinces adjacency matrix in *province\_adjacency\_matrix\_fully\_connected.json* represents a fully connected graph and all adjacency connections are bidirectional. If the file *province\_adjacency\_matrix\_fully\_connected.json* does not exist, it will raise an error.

**Landlocked provinces list**

The file *landlocked\_provinces.json* contains a manually-generated JSON list of strings of landlocked province names. Since you can infer the list of all province names from files such as *province\_adjacency\_matrix\_fully\_connected.json*, and have this list, you can therefore infer which provinces border the sea...

**User interface**

The UI component of this project will be implemented in JavaFX and ArcGIS. The starter code contains a very basic UI (missing many features), with the world split between the Romans(the player) and Gauls, where the player can attack the enemy. In this basic form of the game, there are no turns, the enemy does not attack the player, and the chance of winning/losing is 50%, regardless of the number of troops involved! In this basic form of the game, 60% of attacking troops are lost in a defeat, and 40% are transferred to the conquered province (and no losses are incurred by the player) in the case of a victory. In the basic game, troops cannot be moved between friendly provinces, and infrastructure/troops cannot be built/recruited.

A collection of example images has been provided in the folder images. You are free to use different ones. You can find them elsewhere or even create your own. Some examples came from [here](http://opengameart.org/), most were developed internally by COMP2511 tutor Sam O'Brien (thanks Sam!).

Where the specification does not define how to build the GUI, you can choose how to construct this (keep in mind good design practices, such as Nielson's 10 Usability Heuristics - you will be marked on the quality of your UI/UX in milestone 3).

**Requirement analysis (Milestone 1)**

For this initial milestone, you are to model the requirements of the client as user stories on the issue board in GitLab. You will demonstrate these user stories to your tutor in the Week 7 lab, where they will ask you questions and assign marks based on your answers to these questions (and also through offline marking when appropriate).

It's important that you and your partner meet and collaborate early in order to develop a shared understanding of the project and how you intend to approach it. In developing the stories you will need to consider the requirements as given by the client in this document, but also make your own judgements about the expectations of potential users. Epic stories should be broken down into user stories and each story should have its own card.

The default columns that GitLab provides are sufficient for this project.

You are expected to produce:

1. High-level epic stories from the problem statement. Each epic should have its own card/issue and a corresponding tag used to mark user stories that fall under it.

2. User stories, each containing:

o a short description of the feature based on the Role-Goal-Benefit (or Role-Feature-Reason) template (Refer to the RGB model from COMP1531 if unsure)

o an estimate for the implementation of the user story in user story points (e.g. 4 points).

o a tag indicating the priority

o acceptance criteria for each user story as a checklist in the issue (Refer to material from COMP1531 if unsure)

There should be a "Minimum Viable Product" tag indicating features which form part of a minimum viable product (i.e. absolutely necessary to be implemented, and therefore should be developed first).

As you progress through the rest of the project, you will keep your board and issues up to date: checking off acceptance criteria that have been satisfied and moving stories from **To Do** into **Doing** and finally into **Closed**.

**IMPORTANT**: You must add your user stories to the Gitlab issue board for them to be considered. Work in a different format (such as a PDF or word document) will not be assessed during marking.

**Domain modelling and backend implementation (Milestone 2)**

Based on your requirements analysis, and all feedback you have received, you will produce a domain model for the backend component of your project in the form of a conceptual UML class diagram, implement it in Java and write JUnit tests to test its functionality.

In deciding on your design and writing your implementation, you should follow the practices and design principles covered in the course. You are expected to apply at least 3 of the design patterns covered in the course. It is up to you where they are applied, but you will be expected to justify how and why you used them to your tutor during demonstration.

Your class diagram only needs to be conceptual (showing the general structure of the classes and their relationship), but it needs to be consistent with the code and clearly indicate where you're using design patterns (use labels if necessary).

Your JUnit tests should be rigorous to ensure your backend functions as expected. In addition to basic unit tests, you need to have tests based on your acceptance criteria.

These JUnit tests should be placed into the "src/test" directory, using package "test". The dryrun will check your tests pass. It is important you follow this structure, since we will run automated coverage checking on your program.

In the week 8 lab, your tutor will ask you questions, and assign marks based on your answers to these questions (and also through offline marking when appropriate).

**Running coverage checking**

To run coverage checking, on a CSE machine (or on the Virtual Machine) in the root directory of your repository:

$ gradle test -b test.gradle

The coverage checking report will be in: *build/reports/jacoco/test/html/index.html*

The test report will be in: *build/reports/tests/test/index.html*

Your tutor will receive a report generated using this command, generated from the master branch, from your latest submission as of the deadline.

Please do not push the *build* directory to your Gitlab repository. The gitignore has been configured to ensure this in the most recent version of the *final\_project* repository.

**UI design and extensions (Milestone 3)**

For this milestone you are to design and implement the user interface component of the application. A very basic UI can be built with minimal changes to the starter code, so that is where you should start. Fancier UI features can be added once you have something that is at least usable. You should apply the ideas from user-centric design and consider the usability heuristics covered in the lectures.

Additionally, for this milestone, you also have the chance to extend the project with your own ideas. Note that, to get high marks for these extensions, you will need to consider how they impact the user. **Extensions that are technically complex, but do not provide the user with any real benefit are not considered good extensions**. You can, and should, create additional user stories to model the requirements of these extensions. Note that the number of marks which can be received for extensions is capped at 3 marks (even if the sum of all mark indicators below is above 3 marks).

Possible extensions include but are not limited to:

· Multiplayer campaigns (either on the same computer (0.5 marks), or online (1.5 marks))

· Allowing the player to destroy infrastructure after being built to recoup costs or engage in a "scorched earth" policy (0.5 marks)

· Allowing the player to cancel infrastructure or orders of troops to recoup funds (at a discount, particularly if several turns into production) (0.5 marks)

· Implementing "raids" - allowing the player or AI to attack without trying to conquer the settlement, to possibly damage enemy infrastructure, reduce town-wealth, and kill enemy troops (without needing to assult enemy walls) (1 mark)

· Add turn-by-turn unit upkeep for all units, and implement a bankruptcy mechanic (results in a portion of troops disbanding every turn whilst bankrupt and a loss to morale for all troops). Mercenaries should have higher upkeep (1 mark)

· Define better Campaign AI or Battle Resolvers (0.5-1.5 marks for this category depending on the level of improvement)

· Implement a diplomacy system - alliances, trade between neutral or allied factions, payment/tribute between factions, protectorates, trade embargos, swapping of territory, diplomatic relations based on the actions of each faction, etc. (1.5 marks)

· Public order system - including town halls/inns to increase public order, whilst provinces with low public order have a random chance of a rebellion every turn, and potentially losing the province to the enemy/rebels (1 marks)

· Generals - characters with traits/retinues which build up over time based on their administrative actions or victories/losses in battles (0.5 to 1.5 marks depending on how extensive)

· Agents - to attempt to spy on otherwise hidden enemy forces, or to sabotage enemy walls, towers or infrastructure. Agents engaging in missions should have a chance of failure and execution. Agents should be able to develop traits upon succeeding at missions, increasing their likelihood of success (0.5 to 1.5 marks depending on how extensive).

· Add a "bank" building which allows secure storage of more gold (and with this extension, the maximum treasury size is either limited or has a chance of being stolen by default, with banks reducing the chance of gold being lost and increasing the ability to store gold) (1 mark)

· Add money-lending functionality - where a faction can go into debt to fund wars or develop infrastructure. The interest rate paid on debt should be a monotonically increasing, continuous function of the ratio of *debt/total province wealth* (reflecting that over-leveraged entities must pay higher interest rates due to higher risk borne by the counterparty). Factions should be able to lend to other factions in the same fashion. Note that this will require implementing a bankruptcy mechanic, which should result in a portion of troops disbanding every turn whilst bankrupt and a loss to morale for all troops (1.5 marks)

· Implement terrain for different regions, which have bonuses/penalties to different types of soldiers (e.g. desert regions could provide buffers to desert soldiers and penalties to soldiers from colder climates). Additionally, implement a random chance of rain, snow or fog in battles with different effects (rain should remove the fire-arrow ability of archers, snow should reduce the speed of units by 40%, and fog should reduce the missile attack damage of archers/horse archers/artillery by 50% (to a minimum of 1)) (0.5 to 1.5 marks for this category depending on how extensive)

· Implement the ability for players to customize their victory objective when starting a campaign game by inputting a logical expression representing the campaign objective (a basic goal, or expression composed of logical conjunctions/disjunctions of basic goals and other logical conjunctions/disjunctions) (0.5 marks)

· Add immersive music for all events (0.5 - 1 marks depending on how well this is done)

· Implement visual battles (replacing the TextArea output representation of engagements). This feature should include units with fighting animations such as sword-swings, drawing and firing arrows, etc. (1-2.5 marks depending on how extensive).

· Implement UI double-arrowhead widget dynamically indicating the valid sea crossings between provinces in the game (for example, Britannia connects to Lugdunensis, so the ability to cross the English channel should be visually apparent to the user). This should be produced programmatically, so that if more sea connections are added later, it will update automatically (1-1.5 marks depending on quality)

This final milestone will be a culmination of all the work done in the previous milestones. You have the opportunity to improve on your design based on feedback from your tutor. Marking of the design will be harsher for the final milestone as you have already had the opportunity to receive feedback.

In the week 10 lab, your tutor will ask you questions, and assign marks based on your answers to these questions (and also through offline marking when appropriate).

**Assessment**

You will be assessed on your ability to apply what you have learnt in this course as well as your ability to produce a significant piece of software.

In cases where the client has not been explicit in their requirements, you will need to make your own design decisions with your partner. However, this does not mean you can ignore whatever requirements the client has given you. You may be asked to justify any assumptions you have made during marking.

You are expected to use git appropriately by committing regularly with meaningful commit messages and using feature branches to manage significant changes. Similarly, you should use the task board to coordinate work with your partner. You will need to take the principles you learnt from COMP1531 and apply them here.

While it is up to you how to divide the work between you and your partner, both of you are expected to contribute code. Just creating diagrams and documentation is not sufficient contribution.

**Hints**

· When developing this project, following appropriate design patterns is critical! The project may appear to be a lot of work due to the volume of requirements, but applying the design patterns and principles taught in the course will make the application scale more effectively to different types of soldiers, infrastructure, AI/battle behaviours, and to structures in the game entities.

· The first two milestones do not require a working UI. First determine how you are going to model the game and its entities before considering the UI. A well designed back-end will require minimal change to connect to the UI.

· Up until JavaFX has been covered, you may find some of the starter code to be hard to understand.

· You should use JavaFX Properties where applicable between your backend and the JavaFX UI components. This implements the observer pattern to ensure the frontend and backend are in a consistent state and that they are not tightly coupled.

· The majority of marks available (see below) are for having a well designed application that meets the requirements. Avoid adding extra complexity and extensions till you have something that meets the most basic requirements.

· You will need to engage in research to learn parts of JavaFX yourself. This is expected, and an important learning process mirroring real-world development. You may also have to research how to configure VSCode to work with JavaFX to make some functionality work - this is similarly an important learning process. You should not expect to be taught all of the necessary JavaFX functionality.

· The starter-code has been setup so that you should not have to perform significant research into the ArcGIS mapping library - the starter code already shows:

o A functional historical map of provinces from the Roman empire (use these provinces as as your entire game-world; split them among the factions)

o How to respond to clicking regions

o How to show an image and text above a province

o How to refresh the image and text above a province based on game changes

o How to react to JavaFX events

o How to paste information into JavaFX widgets

· For JUnit in VSCode - it can be useful to run CTRL-SHIFT-P and run "clean the Java language server workspace" if you are having issues with packages/tests recognition.

· For JUnit tests - for this to work in VSCode, it is important to make the test classes public and named after the file they are inside.

· Useful JUnit5 documentation/tutorial: <https://junit.org/junit5/docs/current/api/> <https://junit.org/junit5/docs/current/user-guide/>

· You can sample a normal distribution using *Random.nextGaussian()*, and you can sample a uniform distribution using *Random.nextInt()* for a discrete value or *Random.nextFloat()* for a continuous value. Both of these are within the *Random* class: <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html>

· Terminology summary for battles: **campaigns** contain a sequence of turns (in the same order by different factions, repetitively), **turns** contain battles, moves and orders for construction/recruitment, **battles** contain a sequence of skirmishes, **skirmishes** contain a sequence of **engagements**.

· A unit contains soldiers/troops. The number of soldiers in each unit will vary (infantry units are likely to be larger than cavalry units in number of soldiers).

· Where values are not precise in this spec (e.g. the different individual types such as Hastati/Principes or the precise attack/defense statistics of units) you should develop configuration files for different units/buildings/other entities.

· You will need to develop a format for saved games.

**If you cannot complete all functionality - Minimum Viable Product**

It is standard in software development to think about what constitutes a "minimum viable product". Agile software development revolves around the notion that a minimum viable product is first developed, followed by further sprints which incrementally expand the functionality of the software (at the end of each sprint, the software should at least perform the functionality of a minimum viable product, and gradually expand and improve in functionality).

Planning around constructing a minimum viable product minimises the risk that you fail to provide a baseline of working software in time - which is likely to occur in waterfall software development, by failing to produce multiple versions of working software, and by working directly towards constructing the final product without considering the chance of failure to meet projected milestones in time.

If you cannot complete all functionality in time, you should prioritize more fundamental requirements over the more detailed requirements.

For example, you could implement a purely random campaign AI performing purely random moves, and a battle resolver which makes an immediate calculation from the weighted sum of the attack damage and defense values of the units in the armies. However, this will not achieve as many marks as the more detailed implementation specified in the spec.

The following is a suggestion for the priority of features in a possible minimum viable product (from most important to least important):

· Player is able to play the game

· Player is able to attempt to invade provinces from an adjacent province (and some form of battle resolution occurs)

· Player is able to move troops between owned provinces

· Player is able to recruit troops

· Enemy Campaign AI engaging in random moves, randomly deciding to invade provinces, randomly deciding to recruit troops

· Player is able to construct buildings

If you are unable to complete all functionality, your tutor will make a judgement on how many marks the **working functionality** you have completed is worth. It is important to stress that a minimum viable product will not receive a high mark. This, however, will be worth more marks than software which cannot run!

Do not lose sight of the fact that design and testing is worth many marks - do not purely focus on finishing the full implementation at the cost of design and testing!

**In general, you should prioritize breadth of functionality over depth of functionality for this project, where you cannot achieve both.**

**Important Technical Notes**

**Submission sizes**

Maximum submission sizes apply (for milestones 2 and 3, since a submit command is used). For both milestones 2 and 3, a maximum repository size of 4MB applies. Exceeding this limit may result in you being unable to submit. Please email your tutor if you are unable to submit due to the maximum submission size being exceeded.

A likely reason for you exceeding the memory limit is uploading Java class files, or JavaFX/JUnit/ArcGIS binaries in the *lib*, *bin*, *resources* or *jniLibs* folders. Additionally, this could be caused by pushing the results of running gradle (the "build" folder). You should not push these to your Gitlab repository - particularly since it is configured to your Operating System, which might not match your partner's configuration or the CSE machine. The **.gitignore** file has been configured to prevent pushing the contents of your *lib*, *bin*, *build*, *resources* or *jniLibs* folders - please do not adjust this.

**Directory Structures**

Please adhere to the provided layout of the directories, and filenames and filepaths of the starter files. Modifying this may break tests we run on your submission and may result in loss of marks.

You may add additional Java files with new Java classes in the *src/unsw/gloriaromanus* folder, or new JUnit test files in the *src/test* folder.

**Equal Constribution**

While it is up to you how to divide the work between you and your partner, both of you are expected to contribute code. Just creating diagrams and documentation is not sufficient contribution.

You are expected to contribute equally to your partner every week. At the end of every week, if the levels of contribution were unequal between team members, you are required to email your tutor to discuss and resolve this. If you fail to email your tutor, no redistribution of marks will occur due to the lack of contribution. No further consideration will be granted in this regard.

We will automatically assume that the author of work is the author of the git commit pushing it to your repository for assessing contribution levels - you should not allow you partner to push your work.

We will only consider work pushed to the Gitlab repository in determining contribution levels.

In instances of unfair disparity of contribution, marks may be re-distributed between project partners.

**Submission**

We require your submission to execute without modification after your repository is cloned on our copy of the Virtual Machine, by opening the root repository directory in VSCode, and clicking the *"Run"* link in the **GloriaRomanusLauncher.java** file. You should test this before submitting.

Failure of your repository to work by these specifications may result in a loss of marks. Your tutor will not attempt to fix your code when marking your work (no matter how minor the necessary modificaton).

Only one person in each group should submit. We will assess the latest submission as of the due-time by either partner.

**Milestone 1**

You should have all your user stories entered into the issue board on your GitLab repository. You may continue to use the board between the deadline and your tutor's assessment, but they will be looking at the dates issues were modified to make sure you did the work that was required of you prior to the deadline.

You **do not** have to run a submission command for this to be submitted - we will simply check the issue board, and only mark issues and modifications of issues occurring before the deadline.

**Milestone 2**

Submit the contents of your GitLab repository with the following command:

$ 2511 submit milestone2

Your UML class diagram should be a PDF file at the root of your repository named design.pdf.

**Milestone 3**

Submit the contents of your GitLab repository with the following command:

$ 2511 submit milestone3

You will demonstrate your application to your tutor in Week 10. You may be asked to justify your design decisions and explain how you worked with your partner.

**Marking criteria**

The marks are allocated as follows:

· Milestone 1 (5 marks)

· Milestone 2 (12 marks)

· Final milestone (18 marks)

Below is a *rough* guide on how you will be assessed for each milestone.

**Milestone 1**

| Criteria | Mark |  |
| --- | --- | --- |
| Stories | 0 | No user stories |
|  | 1 | User and epic stories not in a valid format and/or vague or ambiguous |
|  | 2 | User and epic stories in a valid format, but with unclear benefits, goals or acceptance criteria |
|  | 3 | Unambiguous and clear user stories with concrete acceptance criteria |
| Planning | 0 | No user stories have points or priorities |
|  | 1 | Only some user stories have points or priorities |
|  | 2 | User stories have appropriate story point values and priorities |

**Milestone 2**

| Criteria | Mark |  |
| --- | --- | --- |
| Completeness | 0 | No or largely incomplete backend |
|  | 1 | Backend implements very few entities |
|  | 2 | Backend implements some of the entities |
|  | 3 | Backend implements most of the entities |
|  | 4 | Backend implements almost all entities |
|  | 5 | Backend implements all of the entities |
| Testing | 0 | No JUnit tests |
|  | 1 | JUnit tests for behaviour of a few entities |
|  | 2 | Rigorous JUnit tests for behaviour of almost all entities |
|  | 3 | Rigorous JUnit tests for behaviour of all entities |
| Design | 0 | No apparent consideration for design |
|  | 1 | Messy design and diagrams and/or design inconsistent with code |
|  | 2 | Clear design and diagrams with partial adherence to design principles and patterns |
|  | 3 | Clear design and diagrams with strong adherence to design principles and patterns |
|  | 4 | Clear design and diagrams fully adhering to design principles and patterns and conforming to code |

Marks will be deducted for poor git and GitLab usage. For example, meaningless commit messages, large commits, issue board out of date, etc.

**Milestone 3**

| Criteria | Mark |  |
| --- | --- | --- |
| Completeness | 0 | No or largely incomplete project |
|  | 1 | The game can be played with some of the entities |
|  | 2 | The game can be played with most of the entities |
|  | 3 | The game can be played with almost all of the entities |
|  | 4 | The game can be played with all of the entities |
| Design | 0 | No apparent consideration for design |
|  | 1 | Messy design and diagrams and/or design inconsistent with code |
|  | 2 | Messy diagrams and/or poor application of design patterns |
|  | 3 | Moderately clear diagrams and moderate application of design patterns |
|  | 4 | Clear design and diagrams with moderate adherence to design principles and patterns |
|  | 5 | Clear design and diagrams with strong adherence to design principles and patterns |
|  | 6 | Clear design and diagrams fully adhering to design principles and conforming to code, and correct application of design patterns |
| Interaction | 0 | Very basic user interface |
|  | 1 | Interface that makes it possible to play game, but is slow, awkward, or buggy |
|  | 2 | An interface that is mostly usable but with little consideration for usability heuristics |
|  | 3 | Interface that is easy to use |
|  | 4 | Interface that is easy and intuitive to use |
|  | 5 | A product that is engaging, intuitive and fun to use |
| Extensions | 0 | No extensions or only very basic extensions |
|  | 1 | Some implementation of extensions |
|  | 2 | At least 2 extensions of moderate consideration |
|  | 3 | Three or more extensions that represent some technical as well as design and user interaction consideration |

Note that for extensions, you should examine the estimates for marks to be allocated to each extension under the heading "UI design and extensions". The marking guide for extensions above is an estimate, you can achieve full marks with 2 very extensive extensions (each worth 1.5 marks), or with 6 very basic extensions (each worth 0.5 marks), for example.

Marks will be deducted for poor git and GitLab usage. For example, meaningless commit messages, large commits, issue board out of date, etc.