Implementation Report

References to requirements are given in **[ ]** and can found here

Changes to Code

Capture the Chancellor Mode

The *capture the chancellor* mode was fully implemented into the game, in accordance with the assessment 4 requirement extensions **[13]**. A new 'Chancellor' class was created and was organised into the 'entity' folder. The class initialises all associated variables and implements the following methods:

- **move()** - moves the chancellor to a random tile and random point within that tile **[13.b]**.
- **activate()** - sets the chancellor to become active (visible) and begins task to continually move the chancellor every given period. This allows one chancellor to be set to active each time it is needed rather than having a new chancellor initialised every time.
- **deactivate()** - sets the chancellor to become inactive (invisible) and sets its current tile to null.
- **captured()** - deactivates the chancellor and gives the current player a reward of +50 money **[13c]**.

The class also implements mutator methods for the current player and the horizontal and vertical location of the chancellor; and accessor methods for the current tile, activity (whether the chancellor is currently active) and, again, the horizontal and vertical location.

The modularity and organisation of classes was preserved, keeping the code's readability and maintainability; To allow the chancellor mechanic of the game to function, the *GameEngine* will invoke the *Chancellor.Activate* method within the *Chancellor* class at a specified time. When this has commenced, the *GameScreen* will draw the chancellor on the screen through the use of the *drawer class.*

In the *GameEngine,* each transition to phase 3 (in the *nextPhase* method) calls a *beginChancellorMode* method which activates the chancellor and passes the current player to the chancellor object. Upon a transition to phase 4 the *stopChancellorMode* method will deactivate the chancellor **[13.a]**. In the *GameScreen*, the render method checks whether the chancellor is active; if active, the *updateChancellor* method is called - this determines whether the chancellor should be rendered depending on whether the phase is in its last 15 seconds **[13.a]**. This method also initiates the captured method if the *selectedTile* is equal to the chancellor's current tile **[13.c]**.

Maximum Players Reduction

In the project we inherited it was possible to play games with 9 players. A large amount of players was excessive considering the game only has 16 tiles (and finishes after all tiles have been claimed). We reduced this number to 4 to allow players to have at least 4 turns each **[3]**.

The reduction of players was implemented through a simple change of an integer. Within the PlayerSelectScreen class the predicate that stated '`if(AIPlayerAmount + playerAmount < 9)`' was changed to '`if(AIPlayerAmount + playerAmount < 4)`' so now the add player button can no longer be clicked once there are 4 players taking part.

Tile Resource Count

The way tiles initialise their resource counts has been changed. This was done to cater to an original requirement to give tiles resource counts representative of their graphics **[15.b.i]** and to give landmark tiles a 'bonus' **[2.a.ii]**. Tiles containing mostly trees were given more food count, tiles containing large portions of the lake were given more energy and tiles that contained certain landmarks were given more ore. The increase of the resources is given by a multiplier of 150% and is processed within the Tile class in a switch statement.

Commented [1]: Code: the code was uniformly well done. Teams made good (minimal) changes to existing architecture, and also focused on efficiency.

Commenting on new code was extremely helpful, and some teams even copy-edited the comments on previous code to ensure some degree of consistency of style.

Summary of changes: the summary of changes made to the code was a mixed bag. The best solutions had a concise list of changes (with some fine-grained detail) and explicit indexing and cross referencing to requirements and architecture.

The weaker solutions dwelled on changes to the design and design decisions; we were interested in what changes have been made to the code, not the motivation for those changes.

Commented [2]: 3. Implementation [15 marks]:

a) Provide documented code and (in a header comment in the code) the URL of the executable on the team website, for a working implementation of the game that meets the remit, requirements for Assessment 3, and the concrete architecture and changes introduced in Assessment 4. You code comments should highlight new or extended sections of code. Code can be submitted in the zipfile, or via a link to a repository with a verifiable date before the hand-in deadline. (5 marks)

b) Summarise how the software (code and GUI) was modified to incorpor- ate the required changes, and any other necessary changes made to the software, relating each change to the revised final requirements and architecture, and clearly justifying each change. Explain how and why you changed the software and GUI that you inherited. Explain and justify any ex- tra features that have been included in the software. (10 marks, ≤ 2 pages)

Changes to GUI

Few changes were made to the game's GUI and it predominantly retains the form that it was in when we inherited it. It wasn't quite left untouched, however: elements of the primary *GameScreen* class were more thoroughly distributed across the game's internal architecture, and the in-game market's interface was split away from the functionality that had once been interwoven through it. We even found the time to port the random effects **[6]** that we designed and implemented for our last submission into this one, thereby reinstating the complementary overlays indicating when and how they may be applied.

The most significant changes of this kind saw the compartmentalisation of many side-hand " actors" in the main *GameScreen* class across numerous new " table" classes. These new classes are now responsible for declaring and spawning the primary UI elements that collectively comprise *GameScreen's* appearance: the *PhaseInfoTable* class contains the *GameTimer* object **[9.b.i] [9.c.ii]** and the " NextPhase" button that control and shift through the game's phases **[9]** (along with the labels that identify them), whereas the *PlayerInfoTable* class now contains the *Label* and *Image* objects that identify active players' representative colleges **[14.b]** and inventories **[17.a]**. The *GameScreen* class was split up in this way because we felt that it was particularly monolithic - even after having been partially split off into the *GameEngine* class - and, appreciating the use that we got out of porting the original game's *Overlay* and *TTFont* classes in our last development period, felt that it was somewhat important to make more of our UI feel portable and maintainable in practice. There also exists the *SelectedTileInfoTable* **[18.a]** and *MarketInterfaceTable* **[7]** classes that do exactly what you would expect them to, although the mere existence of the latter does raise another noteworthy change to have come out of this assessment period.

The original version of this game (and the version released by *Fractal* after assessment 3) incorporated a singular *Market* class that combined the core functionality of an actual market (from which resources could be bought and sold on the terms of supply-and-demand economics) with a structured interface that facilitated interactions with this market **[7]**. Thus, it was built to extend *LibGDX's Table* class - the same one which provides the spatial framework for everything that appears in the game - and, unsurprisingly, really suffered for it as a component of the game to be maintained over time. It wasn't possible to spawn a functional *Market* object without also spawning an accompanying interface, so it was impossible to unit-test (as *JUnit* - the framework that we adopted to unit-test our game - can't load anything that derives from *LibGDX's Actor* class). It also couldn't be ported over to other projects without necessitating a whole lot of refactoring, and it was even bordering on being monolithic despite only governing a single component of the game. The visual and functional components of the *Market* class clearly needed to come apart, so they did: the aforementioned *MarketInterfaceTable* class now declares the buttons, labels and spatial specifications that embody the game's market on-screen, whereas the *Market* class now only exists to direct the market's core operational logic.

For the first time since they were introduced during the second stage of this assessment, the tooltips bound to the game's 16 tiles were finally extended to display their yields. The requirement necessitating the implementation of this feature (**[1.c.iv] [18.a]**) had been set at the very beginning of the project, so it was high time for it to be met: and it's fortunate that it was, as it significantly lessens the game's unintentional ambiguity and consequently helps players to make more informed decisions about what kinds of resources they should be aiming to acquire.

The only other change to be noted here regards the game's primary font, which was changed from *Big Noodle* to *Montserrat*. The latter font was also worked into our previous submission; it was chosen to be used again here because, as we had found when rounding that submission off, it's a far more readable font that looks cleaner than the alternative and suits the game's (light) " academic" theme just a little bit more.