New Features and Changes

In Assessment 3 there were some significant new features that were required to be implemented , these were:

- The minigame (Requirements F7, F8)
- Bonus exchange/screen - functionality and user interface (Requirement F9)
- Background music and sound effects (Requirement NF4)
- Pause menu (Requirement F3)
- Save and reload - functionality and user interface (Requirement F12)
- Movement

We decided that in order to make these additions we would have to add three new screens for the mini game, save/load and bonus exchange. I order for the implementation of these to be consistent with other screens, a template was made using one of the existing screens, that could then be extended as needed.

Movement

As one of the first additions following the changeover, we felt movement was an important part of the game to implement in order to allow the game to be played through fluidly. Though not explicitly defined in the requirements, the movement phase is a critical part of the way the game plays. Since there was no instructions for how the movement phase should be implemented, we were able to be flexible in the way we worked with this. Consequently we ended up with an implementation that was very similar to the way in which the attack phase works. A user selects a sector that they control, then clicks on another adjacent sector that they control, with an arrow being drawn from the initial sector to their cursor. The user then selects how many troops they would like to move using a dialog box with slider (as used in the reinforce and attack phases). The maximum troops that may be moved from a sector is one less than the total units in that sector (as one unit must always remain).

Due to the architecture we inherited already including a class for the movement phase, as well as an abstract phase class, implementation was relatively straightforward. By making use of previous implementations for the reinforce and attack phases, we were able to quickly implement a fully functioning movement phase. This meant that the generateArrow() method from the attack phase was copied across to the movement phase. Methods getMovedUnits() and moveUnits() were added and the abstract methods defined in the class phase were updated.

Although we experienced some issues with adjacent sectors not recognising that troops can be moved between them, these issues were ironed out relatively quickly, leaving a feature that functions effectively.

Sound and Music

As per requirement NF3, the game was required to implement sound effects and background music. Since the architecture we inherited did not have the required structure to account for this, a new class *Sounds* was added to the solution. This dealt with all of the functions related to sound effects, and ensures that sounds and music are played at the volume defined in the options menu.

When selecting what sound effects to use we took care to observe all influencing factors. With theme, context, clarity, and appropriateness for the users (as per requirements C2 and NF3) all playing a role in the selection of appropriate sounds. Overall we are reasonably happy with the sounds we have used, though given a larger time frame it may have been possible to create sounds that are slightly more in line with the theme of the game.

Since the architecture we inherited did not account for the addition of sound effects, we needed to add a new class *Sounds* to deal with the use of this functionality. The class initialises all the sound effects and music used by the game, and also provides accessors for setting the volume of the music and sound effects. Furthermore, a function was added to allow sounds to be played, which ensures the correct volume is used.

To ensure minimal impact on the architecture, the *Sounds* class is only instantiated as part of *Main*. This however did not provide the ability to add a full range of sound effects, and consequently there are some notable omissions in this regard, mainly with the use of dialog boxes, which proved to be a complication throughout the project. Aside from this, sound effects are implemented where appropriate throughout the project.

Minigame

Requirements F7 and F8 were not specific as to the exact minigame that the game should contain, which left us with lots of flexibility. We made the design decision as a group quite early on that we wanted to make a slot machine for the minigame and production of the assets was undertaken before any new code was written.

Once this was completed, the UI was implemented using the template screen created earlier as a starting point. Once we were happy with the appearance, the spin logic was the next thing to add. This works using a random number generator to pick which image to display in each slot every time a new frame is rendered. The first slot updates 50 times per spin, the second 100 and the third 150 to mimic the operation of a real slot machine. The current state of the spin cycle is tracked using global flags inside the class, which are then reset after each spin. This allows for multiple spins from the same instance of the screen although we have currently limited the mini game to one spin each time you win access to it, and a new screen is instantiated each time to ensure that everything is reset properly.

Access to the minigame is through a dialog box that was added to the Dialog Factory. This has a one in five chance (again implemented using a RNG - random number generator) of being called by the attack success dialog boxes called by attackSector() in Map. When the user clicks “Play” on this dialog box it calls the game screen method that starts the minigame.

Pause and Resume

One of the biggest challenges we faced when implementing pause and resume was getting it to work with the game timer. This was partly due to the fact that the timer works by finding the difference between the current system time and the time when the timer was started to get the time elapsed. This meant that to pause the timer we have to similarly count the amount of time since the timer was paused and add it as an offset when the timer is resumed again.

We also had to refactor a lot of our pause methods when we realised that there were times when the timer needed to be paused without the pause menu being displayed, for example when the user leaves the game screen to play the minigame or visit the bonus exchange screen. This meant that we added two global boolean flags “gamePaused” and “timerPaused” that track the current pause state. There are methods to pause and resume the timer that are called when the game is paused but also when switching to the other screens. The render method will draw the pause menu based on the current value of gamePaused.

Enum EntryPoint

In order to be able to access the OptionsScreen and LoadScreen from the main menu and the in game pause menu it was necessary to follow where the screen was accessed from in order to return to the correct screen on exit. In order to achieve this we created a new enum that could be passed to the new screen.

Save and Reload

A key aspect in most games, this one included, is the function to both save and load a game. This is so that the users playing the game can pause and save the game at anytime and whenever they feel like it, they can come back and pick up the game where they left off.
We were given a requirement (F12) stating that the game should be able to save and load and that is should have at least one save game. The project we picked up had no existing framework for saving and loading. Due to the nature of the programming language we are using (java), a lot of the variable types and primitives we used are easily savable.
The saving method we used, is to collect all of the necessary information into a new class. This class was aptly named Data. It was then made to implement serializable which allowed the writing of the class directly to file. This made it easy to save all the data as there was no need to convert any of the information to something that we could read and write from file, other than the object itself.

The information that is saved is primarily, the sectors, the players, the current players turn as well as the current time elapsed on that turn. The sectors contain the information about who owns what sector as well as the amount of units on that sector. The player stores the college that they picked along with the name of that player. Then the other information stored is necessary to reset the functionality of the game to that of the same game. For example whether or not the turn timer was enabled for that game.
Although saving was the simple process of collecting all of the data and sorting out all of the objects that were serializable and those that were not, converting those that were not into a similar object that was, or a collection of primitives that were, then serializable. Loading was a whole nother ball park. Many issues popped up from the object not being saved correctly, to trying to reset variables to the values they were when they were saved. After some creativity and thinking outside the box, methods were implemented that allowed the Load class to reinitialize the necessary variables to the values they were when the game trying to be loaded was saved.
To display the data we created another screen class, LoadScreen, which allows the user to view all of the saved games that have been saved, up to a limit of four. This allows the user to pick the game they want to continue as it will give a picture of all of the colleges in play along with the names of the players associated with those colleges. In this screen the user can select the save slot they wish to save on/overwrite or to load from.
All of this completes the requirement F12 by allowing the user to save a game which they can then reload at another time.

Bonus Exchange

The bonus exchange screen referenced in requirement F9 was the last of the new screens that we added. Again we created the UI elements first and then started implementation from the template screen. Once the UI was completed it was just a case of implementing the logic. This mainly consisted of accessing the data from the player class about the bonus amount held and writing the changes to the bonus and amount of troops to allocate back when the convert button is pressed.