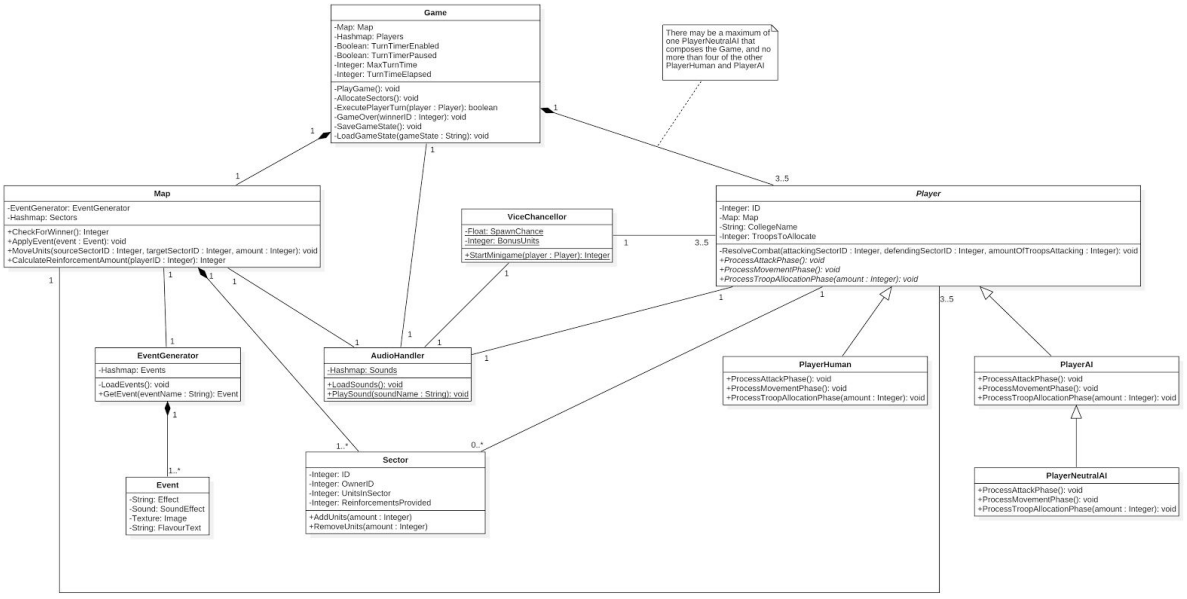
Architecture

Conceptual Design

Class Diagram

We have produced a conceptual model of our games class structure. The software architecture is represented in UML 2.x [1] and was drawn in StarUML. The diagram [2] details a proposed structure of the gameplay section of the software, the menu and options systems have been omitted. Furthermore, getter, setter and other low level implementation details have been omitted to ensure the diagram is easy to read.



Notation

The diagram shows a conceptual design of the software's classes. The structure was designed with a Java based implementation in mind. Each box represents a class which may contain attributes and operations. Attributes are in the format of 'Identifier: Type' and operations are in the form 'Identifier(Param1 : Type, Param2 : Type…): Return Type'. The links between the classes show how classes associate with each other. A line with no marking at either end shows a simple association. A black diamond at one end shows that the class with the diamond at the end is composed of the other class, this a strong relationship i.e. the class that is made up of the other one could not exist without the other one.Finally, an arrow head marking shows the class that is being pointed to is a parent class of the other one.

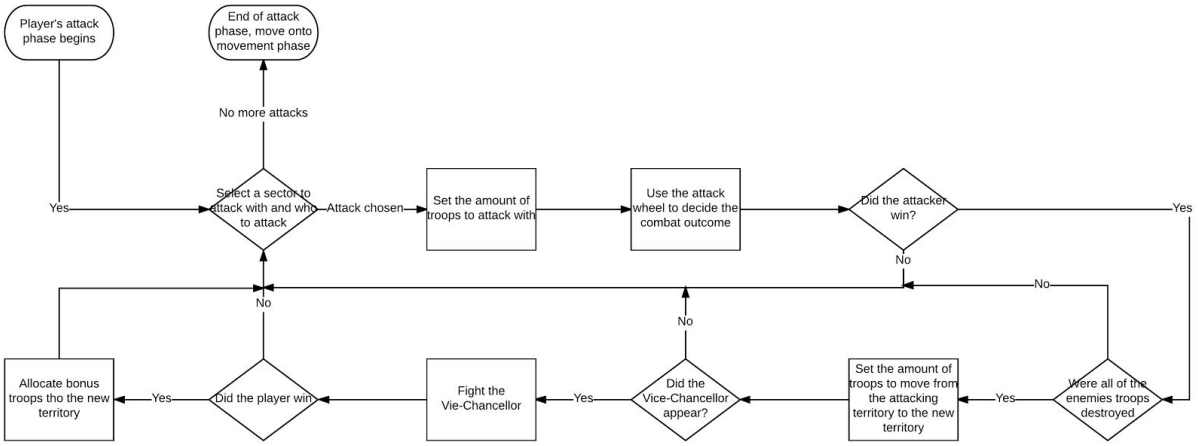
The numbers at each end of the links shows the multiplicity of the relationships. For example, the relation between Game and Player may be interpreted as one Game class is composed of between 3 and 5 player classes. 'n..*' denotes a multiplicity of n or more.

Variables or methods that are <u>underlined</u> are static methods and classes and methods that are in *italics* are abstract classes and methods.

Gameplay Flowchart

The gameplay is composed of four key phases: initial troop allocation; attack phase; movement phase and troop reinforcement phase. The initial troop allocation phase only occurs once for each player, at the beginning of the game, whereas the other phases occur each turn for each player. A full diagram showing the stages of gameplay can be found in document [3]. This document shows the order of processes from the start of the game; through each player's turn and finally checking at the end of each turn if a player has won.

The most complex stage in the gameplay loop is the attack phase. The diagram below illustrates an attack phase, extracted from the main gameplay loop.



The flowcharts were produced using Lucidchart.

The player is able to attack with any sector they wish as many times as they like, as long that sector has troops on it to carry out the attack. They must choose the amount of troops they would like to attack with and then the combat mechanism is used to decide the outcome of battle. If the player successfully destroyed all of the opponent's troops then they must choose how many troops to move onto the newly acquired tile, from the tile they attacked with. Additionally, when a territory is conquered their is a chance the Vice-Chancellor will appear and challenge the player to a mini-game. If the player then wins this challenge then the tile they conquered is given some bonus reinforcements.

Architecture Justification

This system architecture was designed with the client's requirements in mind. These requirements may be found in document [4] and specific requirements are referenced using the ID numbers, as defined in the Requirements document. The conceptual design of the system is justified below:

Game

The Game class is the main controller of the gameplay. It sets up the game initially, allocates the Sectors to each Player as needed by F17; contains the main game loop; manages the players' turns; saves games and loads games from save files. The saving and loading system is required by F12 and performed using the methods SaveGameState and LoadGameState.The class contains the boolean variable TurnTimerEnabled so that the turn timer can be toggled on and off, this satisfies requirement F2. Additionally to meet requirement F3, the boolean TurnTimerPaused is used to pause the timer if the mini-game is in progress.

One game is composed of one Map and between three and five Players. It is also associated with the Audio Handler so that the Game can control the music currently playing.

Map

The map is responsible for storing each sector in the game map. It must also be able to check if a player has won yet; apply the effects of a random event to the relevant sectors; move units between Sectors and calculate how many reinforcements a Player should receive at the end of their turn, based off the Sectors they own as required by F16.

The Map class is composed of at least one sector. It is associated with an Event Generator so that if an event occurs at the start of a turn then one can be fetched to apply and is also associated with the Audio Handler so that sound effects from events can be played. Finally, the Map is associated with between three and five Players so that the Player's actions in their turn can be carried out, e.g. the Player can move units between Sectors.

Sector

This class stores who owns this Sector, the OwnerID, and how many units are currently in this Sector. It has methods for adding and removing units from the Sector.

A Sector is part of the composition of a Map and a sector is associated with one player. It is associated with the player that owns it.

Event Generator

The Event Generator class must load the possible random events that can occur from a file and create event objects so that these events can be applied to the map.

The event generator is composed of one or more Events. When GetEvent is called, by a Map which it is associated with, one of the Events is returned so that it may be applied to the Sectors in the Map.

Event

The class stores what effect the Event has on a Sector when the Event occurs; the sound effect that should be played when the event occurs; some flavour text to tell the player what the effect was and an image to show the player, visually, what has happened.

An event is part of the composition of the Event Generator.

Player, PlayerHuman, PlayerAI and PlayerNeutralAI

PlayerAI and PlayerHuman are children of Player, and PlayerNeutralAI is a child of PlayerAI. Player contains abstract methods for the logic of what to do in each phase of a player's turn. PlayerHuman will be implemented so that a user can decide what they want to do. PlayerAI should contain logic so that a human could play against a computer and PlayerNeutralAI is a special case of PlayerAI where the AI will only defend, not move or attack.

The Game is composed of between three and five Players, where there may be a maximum of one PlayerNeutralAI and there can be a maximum of four PlayerAI and PlayerHuman, i.e. there is a max of 4 normal Players and then additionally there may be a neutral player. This is required by F10.

Player is associated with Map so that the Player's actions in a turn may be executed, i.e. attacking another Sector; moving troops between Sectors and allocating reinforcements to Sectors. Attacking another sector triggers the conflict resolution system, as required by F5. Player associates with Audio Handler so that sound effects may be played when the Player makes an action. In case of a Player conquering a Sector after having attacked it and the Vice-Chancellor appearing the Player is associated with the ViceChancellor class so that the mini-game may be triggered.

Vice Chancellor

This class contains a static method that when called triggers the mini-game. The method returns an integer value of how many bonus reinforcements the player should be awarded, if they do not win this will be zero as required by F9. It is not guaranteed that the Vice-Chancellor will appear when a Sector is conquered, the probability is determined by the SpawnChance variable.

The class is associated with the Players so that if they the Player conquers a territory then the StartMiniGame method can be called. It also associates with the AudioHandler so that sound effects within the mini-game can be played.

Audio Handler

The Audio Handler is used to load audio files and play them when PlaySound is called. This meets requirement NF4. PlaySound is a static method so that it can be called at any point during the game from other classes as there are many occasions where sound may need to be played, e.g. during the mini-game; when troops attack each other and when a random event occurs.

Audio Handler associates with Game, Map, Player and ViceChancellor so that they may play music and sound effects.

References

[1] Bell, D. (2004). The class diagram. [online] Ibm.com. Available at: https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/index.html [Accessed 3 Nov. 2017].

[2] SEPR "UML Class Diagram" Risky Developments [Online]. Available: http://www.riskydevelopments.co.uk/documents/UMLClassDiagram.png [Accessed 3

[3] SEPR "Gameplay Flowchart" Risky Developments [Online]. Available: http://www.riskydevelopments.co.uk/documents/FullGameplayFlowchart.png [Accessed 3 Nov. 2017].

[4] SEPR "Req1" Risky Developments [Online]. Available: http://www.riskydevelopments.co.uk/documents/Req1.pdf [Accessed 7 Nov. 2017].