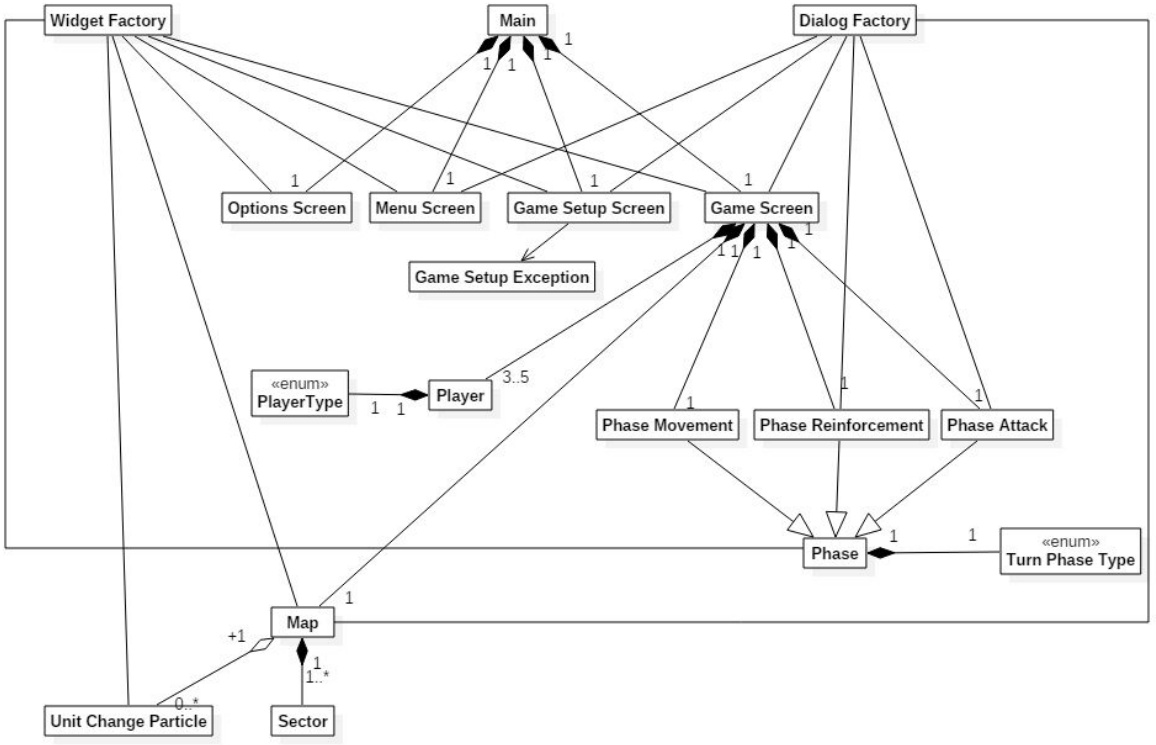
Architecture

Concrete Architecture

UML Diagram

Our full UML Diagram showing the concrete architecture is available at [1]. However due to the softwares complexity it is somewhat difficult to follow and therefore we have created a simplified version that simply shows the relations of classes to one another.



A full resolution version of the diagram is available at [2].

Language

The diagram is presented using a modified version of the UML 2.0 standard. The standard has been modified to omit the class variables and operators as due to the amount of them it made the diagram difficult to read and follow. Because it is the modelling language most commonly used in software engineering and we are therefore confident that other teams will be able to understand the design of our system.

Tools

The full UML diagram was produced using the IntelliJ IDE as it was able to automatically generate the full diagram from the source code. The advantage of this is that the diagram

was produced directly from the source code therefore minimizing mistakes in production, because of the complexity of the software such mistakes may be likely.
Additionally, the simplified diagram showing the class relationships was produced using StarUML.

Architecture Breakdown

Modifications To Abstract Architecture

This document references the updated statement of requirements available on our website[3].

When it came to implementing the game it became clear that our initial abstract architecture [4] had some large flaws. In addition to some of the impracticalities of the architecture we also removed some features we had initially designed as there was not time to implement them or we were not required to implement them.

The core concept of having a class that handles the flow of gameplay remain the same, Game in the abstract architecture and Game Screen in the concrete are analogous. However its behaviour has also been expanded to control the phase that the game is in. The three phases, (Reinforcement, Attack and Movement), can easily be switched between allowing the player to perform actions specific to that phase. This enables the software to satisfy requirement F18 as when in the Reinforcement phase the player has the choice to allocate new gang members.

Due to time restriction we removed having an AI player. This lead to us no longer needing the PlayerAI class. Furthermore the behaviour for handling how a Human and Neutral player act in a turn phase was moved to the Phase class. This lead to there being no difference in methods and data between the PlayerHuman and PlayerNeutalAI classes, they just have different constructors. Therefore those classes were also removed and replaced with static create methods in the Player class.

A screen system was added to the software in order to satisfy requirement F15. This is in the form of Main which then controls which of MenuScreen, Options Screen, GameSetupScreen or GameScreen are being shown.

Justification of Architecture

- **Main** - the Main class's purpose is to control the screen that is currently being displayed and apply the users options preferences. It is required as requirement F15 says that the user must be shown a setup menu before starting a game and therefore some way of changing screens is also needed.

- **Menu Screen** - this screen is what the user is presented with when the game is first started. It is used to access the other menu screens: Options Screen and Game Setup Screen.
- **Options Screen** - an Options Screen was required as the software must run on multiple systems with different display peripherals and this screen can be used for configuring the window resolution or enabling fullscreen mode.
- **Game Setup Screen** - the screen was necessary as F15 states that the user must be presented with a setup screen before starting a game. Additionally the screen is used to enable and disable the turn timer as required by F2. Finally, F10 and F11 state the possible player configurations. This screen is used to enforce that any game that is setup meets these requirements. If any of the configuration requirements are not met then a Game Setup Exception is thrown.
- **Game Setup Exception** - this exception was created so that if a game is attempted to be started with an invalid game setup then an error can be thrown. This exception is caught when an invalid game is started and a dialog with an error message is then shown to the user so they can correct the error in their setup.
- **Game Screen** - the Game Screen controls the main flow of gameplay and stores the Players, Map and Phases. It is key to ensuring that the game runs correctly as it. It controls which phase the game is currently in and whos turn it is. Additionally it checks if a player has been eliminated or the game is over.
- **Player** - the Player class stores data about the player which was collected from the setup screen. In addition to this it also stores the amount of bonus troops the player will receive to allocate at the start of their next turn, in order to satisfy F14. Our requirements state that the game can be played with between 2 and 4 players and a neutral player must be enabled if there is only 2 players F10, F11. Therefore the relation between Game Screen and Player has a multiplicity of 1:3..5 because there will always be 1 Game Screen to 3 Players, at least 2 human plus a neutral, and upto 5 as there can be 4 human players plus a neutral one.
- **Map** - the Map's main job is to store the sectors within the map. It keeps a HashMap of Integer IDs mapping to the Sector classes. When a Map is created it must allocate the sectors to each player, F17, this is done using the allocateSectors method. Finally as the game requires a combat mechanic, F5, this is going to lead to the number of troops within sectors changing. Therefore the attackSector method handles applying the results of combat to the amount of units in sector. Map has a list of Unit Change Particles in order to show to the player visually any changes in the number of units on a sector.
- **Sector** - sectors compose map as they are required in order to make up the map. They store how many troops are on that sector and who owns it. They also store the sector IDs that are adjacent to them so that when a player attempts to carry out an attack they it can be checked that the attack is on an adjacent sector. Using the Map and Sector system supports meeting requirement F13.
- **Unit Change Particle** - this class is used to store a particle effect showing the changes to the number of troops on a sector.
- **Phase** - the base phase class is used to set up the UI components that are consistent across the three phase types. By setting up these UI components in the

base class it removes needless code duplication by setting them up three times in the Movement, Reinforcement and Attack classes.

- **Phase Movement** - currently not implemented for this assessment, but the Phase Movement class will handle the input and rendering specific to the movement phase.
- **Phase Reinforcement** - the Phase Reinforcement class handles handles the player input specific to this phase. Meaning that when a player clicks on a sector they own, if they still have troops to allocate, they are shown a dialog asking them how many troops they would like to allocate. This is in line with requirement F18.
- **Phase Attack** - this class handles the player input and rendering specific to this phase. To support requirement F5 the class allows the player to choose a sector they own to attack from and an adjacent sector they do not own to attack. They then may select how many troops they want to attack with, the more troops you attack with the more you risk losing, introducing an element of skill meeting F6.
- **Widget Factory** - the class is is used for generating UI components that can be added to the GUI. A factory pattern was used as it allows to use reuse methods for generating similar components, i.e. buttons that look the same but contain different text. This reduces the amount of code and makes it more maintainable as to update the style of a set of buttons only one method needs to be changed rather than for each time a button is created.
- **Dialog Factor** - the class generates dialog boxes that display information to the player and also may take some input from them. The reasons for using a factory are the same as for a Widget Factory. We used a dialog system as its an intuitive way to display information to a player and therefore should help new players understand the game easily, NF1.

References

[1] SEPR “Concrete UML” Risky Developments [Online]. Available: http://www.riskydevelopments.co.uk/documents/concreteUML.png [Accessed: Jan. 21 2017]

[2] SEPR “Simplified UML” Risky Developments [Online]. Available: http://www.riskydevelopments.co.uk/documents/concreteArchitecture.png [Accessed: Jan. 22 2017]

[3] SEPR “Updated Requirements” Risky Developments [Online]. Available: http://www.riskydevelopments.co.uk/documents/UpdatedRequirements.pdf [Accessed 21 Jan. 2018].

[4] SEPR “Abstract Architecture” Risky Developments [Online]. Available: http://www.riskydevelopments.co.uk/documents/Arch1.pdf [Accessed: Jan. 22 2017]