The Concrete Model & Tools Used

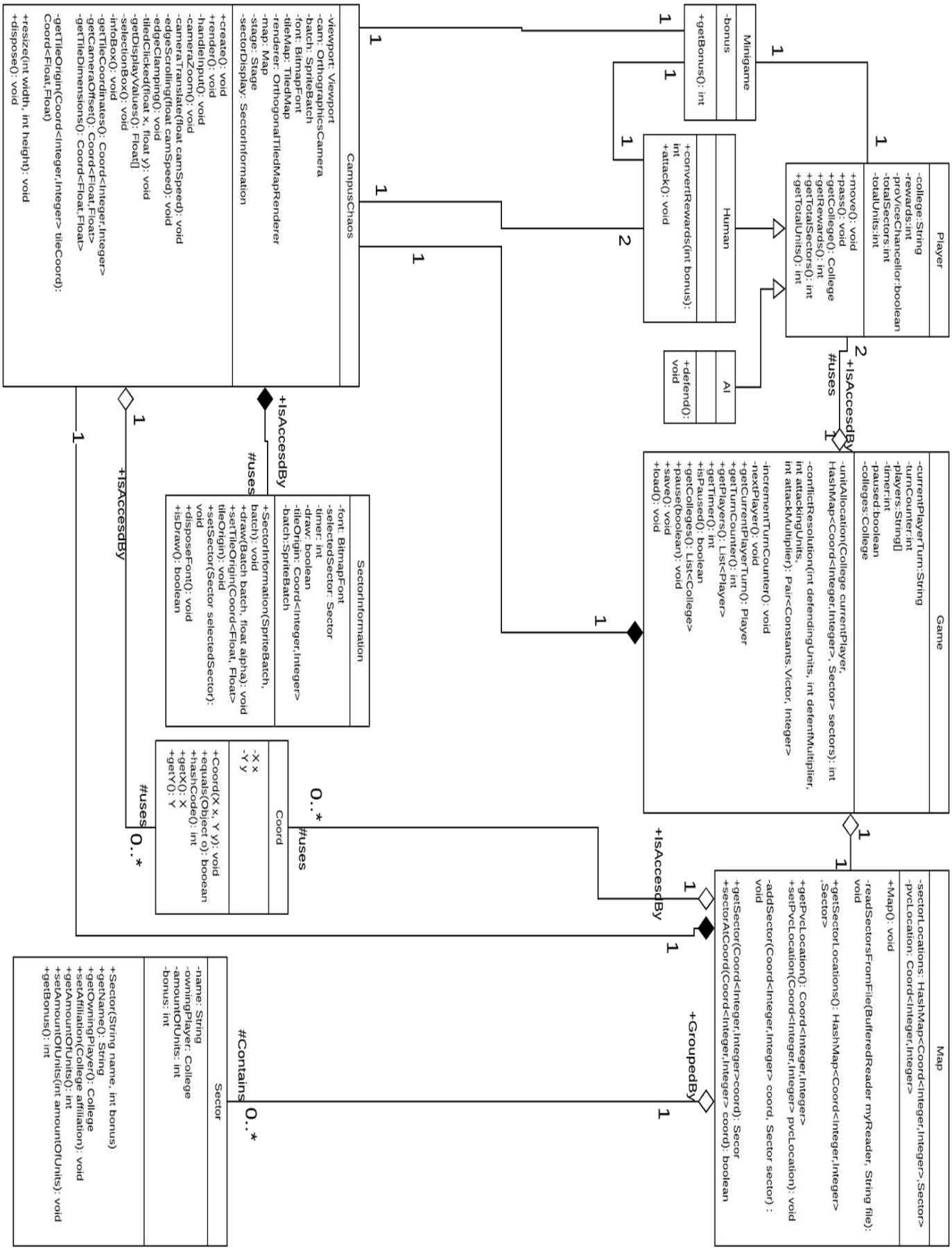
Creating a concrete representation of the architecture required the collaborative work of the entire team to move the project in the way that we envision the end product to be. UML 2.0 was chosen as the specification to be used, more specifically the Class Diagrams. Doing this will allow us to create a clear and detailed diagram of the game as close to code level as can be predicted. This will give a clearer picture of the route to take for the rest of the implementation of the game. We have still chosen to use Lucidchart as the software to build the diagrams with as it was the easiest to access with an appropriate outcome.

Moving onto the concrete design for our game, as a group we collaboratively brainstormed as to how we want each method to interact with certain classes throughout the program. This gave us a basis at which, not only could we start to create the more detailed diagram, but also where to start implementing the game. We expanded each method to what we thought the parameters and the values returned should be. With our gained knowledge from research into libGDX, we could also expand the GUI class from the abstract diagram to contain all the elements necessary to create a intuitive GUI that would be easy to use. This class was the most changed between the abstract diagram and the concrete due to the amount of features which had to be added to create the display.

Through the implementation of the game so far, we often referred back to the concrete and abstract architecture to set out well defined structure for our game. Although inevitably we could not predict exactly how everything should be pieced together or what classes required methods we didn't even know we'd needed until implementation. This means that throughout the implementation so far we have updated and remade the class diagram to suit the knowledge of how the structure of the program should be built. This could be something as simple as adding a class to take care of a certain element of the game which could be modularised into a class.

Due to the diagram being so big, we have added a very cramped version of the image below which shows the basic relationships between classes. However the variables and methods can't clearly be read so you can download the image from:

http://www.sidmeiers.me/images/ConcreteArchitecture.png


Player
-college:String
-rewards:int
-proViceChancellor:boolean
-totalSectors:int
-totalUnits:int
+move(): void
+pass(): void
+getCollege(): College
+getRewards(): int
+getTotalSectors(): int
+getTotalUnits(): int
Minigame
-bonus
+getBonus(): int
Human
+convertRewards(int bonus): int
+attack(): void
AI
+defend(): void
Game
-currentPlayerTurn:String
-turnCounter:int
-players:String[]
-timer:int
-paused:boolean
-colleges:College
-unitAllocation(College currentPlayer, HashMap<Coord<Integer,Integer>, Sector> sectors): int
-conflictResolution(int defendingUnits, int defenfMultiplier, int attackingUnits, int attackMultiplier): Pair<Constants.Victor, Integer>
-incrementTurnCounter(): void
-nextPlayer(): void
+getCurrentPlayerTurn(): Player
+getTurnCounter(): int
+getPlayers(): List<Player>
+getTimer(): int
+isPaused(): boolean
+getColleges(): List<College>
+pause(boolean): void
+save(): void
+load(): void
Map
-sectorLocations: HashMap<Coord<Integer,Integer>,Sector>
-pvcLocation: Coord<Integer,Integer>
+Map(): void
-readSectorsFromFile(BufferedReader myReader, String file): void
+getSectorLocations(): HashMap<Coord<Integer,Integer>,Sector>
+getPvcLocation(): Coord<Integer,Integer>
+setPvcLocation(Coord<Integer,Integer> pvcLocation): void
-addSector(Coord<Integer,Integer> coord, Sector sector): void
+getSector(Coord<Integer,Integer> coord): Secor
+sectorAtCoord(Coord<Integer,Integer> coord): boolean
CampusChaos
-viewport: Viewport
-cam: OrthographicsCamera
-batch: SpriteBatch
-font: BitmapFont
-tileMap: TiledMap
-renderer: OrthogonalTiledMapRenderer
-map: Map
-stage: Stage
-sectorDisplay: SectorInformation
+create(): void
+render(): void
-handleInput(): void
-cameraZoom(): void
-cameraTranslate(float camSpeed): void
-edgeScrolling(float camSpeed): void
-edgeClamping(): void
-tiledClicked(float x, float y): void
-getDisplayValues(): Float[]
-selectionBox(): void
-infoBox(): void
-getTileCoordinates(): Coord<Integer,Integer>
-getCameraOffset(): Coord<Float,Float>
-getTileDimensions(): Coord<Float,Float>
-getTileOrigin(Coord<Integer,Integer> tileCoord): Coord<Float,Float>
+resize(int width, int height): void
+dispose(): void
SectorInformation
-font: BitmapFont
-selectedSector: Sector
-timer: int
-draw: boolean
-tileOrigin: Coord<Integer,Integer>
-batch:SpriteBatch
+SectorInformation(SpriteBatch, batch): void
+draw(Batch batch, float alpha): void
+setTileOrigin(Coord<Float, Float> tileOrigin): void
+setSector(Sector selectedSector): void
+disposeFont(): void
+isDraw(): boolean
Coord
-X x
-Y y
+Coord(X x, Y y): void
+equals(Object o): boolean
+hashCode(): int
+getX(): X
+getY(): Y
Sector
-name: String
-owningPlayer: College
-amountOfUnits: int
-bonus: int
+Sector(String name, int bonus)
+getName(): String
+getOwningPlayer(): College
+setAffiliation(College affiliation): void
+getAmountOfUnits(): int
+setAmountOfUnits(int amountOfUnits): void
+getBonus(): int
#uses
+IsAccesdBy
+GroupedBy
#Contains
0..*
1

Justification

We have decided to use a UML class diagram as it gave us the capability to go into express the relationships between each of the classes

Game

The game class is the one of the classes which hasn't changed very much from the abstract representation of our project. Like the rest of the classes to follow, each variable in the class has been been given a type that the variable should have in the program. Along with each method in the class should have the parameters it needs and their types, then whether the method returns a value or not, and its type.

Through the thought of how the game should run during execution we added a few variables to the class. The variable turn from the abstract architecture, was transformed into currentPlayerTurn, A string containing the name of the current player to be displayed so the users know whose turn it is, and turnCounter, which keeps track of the current turn of the game.

Also we added colleges of type College as this holds all of the possible playable colleges in the game. College is an Enum type which are all initialised with the name of the college. When a college is picked, it will then represented by the enum for the rest of the game. This prevents any potential errors occurring from a mistyped string. (F5)

The methods which have been added to the class are mainly the getters and setters for all of the variables in the class. This is set up this way so that none of the variables may be directly changed and would then have some safety conditions so that the values are never out of usable range.

The specific methods which are unique to the game class are the pause, save and load methods these are used as the name suggests, to pause/unpause the game, save and load the game.(F2.1, F2.2, F2.3)

We have also developed two algorithms for unit allocation and conflict resolution. These will be under the game class as it's generic to each player so it should be easily accessible. (F7.1) (F9) (F10)

Map

The map class has for the majority, stayed the same as the abstract diagram. The sector amount has been removed as it has been transformed into sectorLocations. SectorLocations is now a HashMap with the coordinates of the sector as the key and then the instance of the sector as the value. There is also now the pvcLocation which is just a set of coordinates which can be compared against to see if the sector is the pvc sector or not. (F3)

Again the class comes with the getters and the setters for the values above except for sectorLocations which doesn't have a setter except there is a method in the class that reads all of the sectors and their coordinates from a file and initialises the map when the constructor is called. There is also a methods named sectorAtCoord which checks to see whether or not there is a valid sector at the coordinate given.

Sector

The sector class has been stripped of the pro-vice chancellor sector variable as that is now in the map class to be checked against from elsewhere in the program. However everything else is pretty much the same as the abstract diagram except for values being given to each variable.

Player

The player class has been given two extra variables named totalSectors and totalUnits, both are ints. These are so that during any calculation for the player (namely conflict resolution) the variables can be used instead of having to do some form of maths to get the same result.

The Player has also been given a number of methods which allow the player to interact with the game. These are moves that the player would perform during gameplay as well as methods which display information to the player which is needed to inform the player of their valid moves.

Human

As Human inherits from Player, not much was needed to be added to the Human class. The only difference is that the humanplayer needs to be able to convert the rewards it has into units and be able to attack.

AI

The AI class hasn't changed very much as until the implementation of the class not all of the methods or variables can be foreseen. We realise that there should be a method for the AI to defend itself but under that method, it could call on several others.

Minigame

We know that the minigame will be much larger than what is described on the concrete diagram. The reason for it being so bare is that the methods and variables that the class would use depends entirely on what the minigame is to be. We have left this minigame to be decided upon later and as such, the diagram should be changed accordingly. We only know that the minigame has a bonus and completing that game gives a bonus to the player. (F4)

CampusChaos (GUI)

CampusChaos is the GUI class, the reason for the rename of the class is that we have realised it is the main class which calls all other classes in our project. This is due to the desktop launcher provided by libGDX launching the GUI class first.

GUI has been given a number of variables and methods all related to displaying the GUI. These include: The map; The font for the text; The camera itself; The renderer; and a few others. (F11)

All of these are used in a number of methods. Important methods are: The create class which is used to set up all of the variables needed for the GUI to run; The render class which is called sixty times a second, this keeps the camera where it is and then calls functions to update the GUI based on input from the mouse; Finally the dispose method which removes all of the unnecessary variables at the end of the program.

There are a number of other methods which are all used to perform mathematical functions to update all of the elements of the GUI.

SectorInformation

Sector Information is a new class which is an actor for the stage in the GUI. This class contains all the information from a sector which has been selected by the mouse. The variables include the location of the sector, the font for the information to be displayed and a boolean draw to inform the GUI whether or not the information should be drawn.

Methods for this class are all based around the the draw method which is called by the stage from the CampusChaos class. This function draws all of the information in the bottom right hand corner of the stage.

Coord

Coord is a custom class to act like the Pair class for java. This was implemented so the amount of information needed to give to the class was just the x coordinate and the y coordinate. Methods are just the constructor and getters for the variables.