SD3 – Game Report

Game Idea – My game idea expands upon the one in the specification. It is still a space game but instead it used a 7x5 grid and includes a major trading element. The player controls a spaceship which can fly one space per turn, if the player lands on the same space as an enemy spaceship its strength value will be compared to the cumulative total of the other spaceships. Who ever has the highest will win and this will result in the other being destroyed. The player can gain support from planets by being on the same space as one when they are attacked. The player can also trade with local civilian ships as well as the local planets for resources and money. The idea was to have an economy that was affected by purchases and with different purchase rates for each planet so you could buy and sell and make a profit but due to time constraints this was omitted. The player can also load/save a game that has been written to the file as well as starting a new one from the main form.

Threads – I have used individual threads for running each form that loads up as well as implementing a separate thread for updating the player map on the screen. When the application loads a separate thread is created which cycles every half second, this thread updates the user interface so that the player can see the position of everything on the map. The thread also attempts to call the moveEnemy() method which changes the enemy ships positions, this works in unison with the Observer pattern as the enemies will not move until they have been notified that the player has already done so. It also updates a list box with all the object positions. Without threads this would’ve become quite troublesome to do from my Game class whenever something is changed.

Object Serialization – I have used object serialization to save my main game class to disk. This allows a player to save the game’s state and reverts back to it whenever they wish. The benefit of this is that when the program is no longer running a player may return to a game they had previously started or if they make a mistake they may go back to a previous save point. Without using object serialization it would mean that every time the game is started an absolute brand new state would be created not allowing any continuity. A game would have to be played until the end.

Singleton - I have used the singleton pattern to manage my main game class. This class manages and controls all the player actions as well as interactions between the other classes in the game such as the enemies and the planets. The key benefits of this is that all the information can encapsulated in one centralised location and the game itself and all its functionality can be separated from the GUI. Also because it can only ever have one instance this means that if you access it from anywhere else you can access only what you need yet it being the same data as elsewhere. Without the singleton pattern this would’ve required a lot of parameter passing dependent on certain conditions within the game or a possible merger between the GUI and the Game classes.

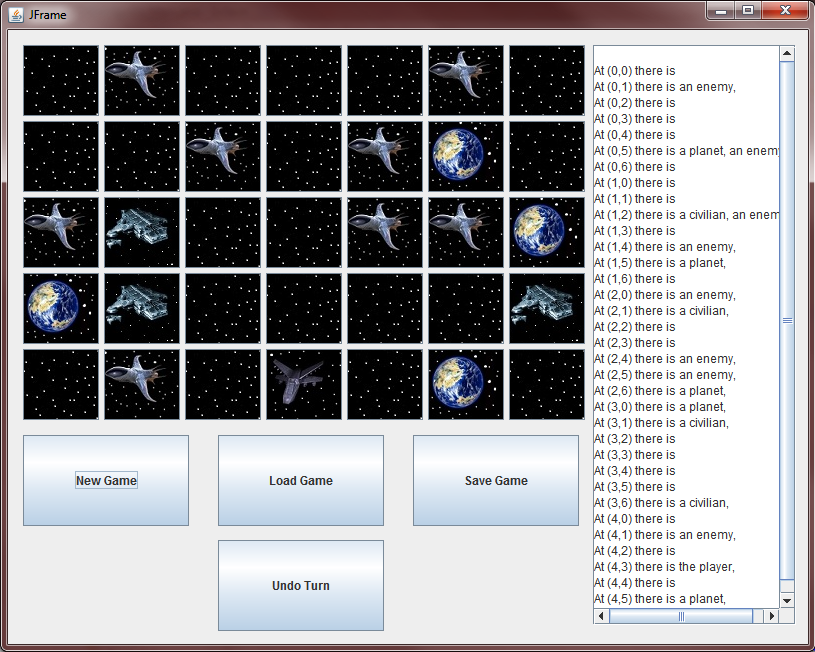
Factory – I have used the factory patten to manage the creation of different objects within the system. I have four different factories, one for the creation of ships, one for creation of planets, another for the creation of items and one for the creation of names. The main benefit of the factory pattern is that it makes it easier to organise many related objects together. Without the factory pattern this may have been difficult with related objects and would’ve exposed other classes to the instantiation of these objects unneccisarrily.

Observer – I have implemented the observer pattern with my enemies and player movement. The enemy ships movement runs on their own thread and wait until the player has moved before they update their position. When the player moves it calls the setPMoved() method of the enemy ships and sets it to true. Once all the ships are notified they will all begin moving. The Observer pattern in this case allowed me to stop an action from being done until a certain event or set of conditions had been met, giving me better control over the flow of the game itself.

Strategy – The strategy pattern makes controlling the behaviour of different entities within the system much easier. Being able to alter their methods dynamically by allowing different behaviours such as Attack, Trade and Defend would have given me interesting combinations. Such as planets who are hostile to you, pirates who trade stolen goods and passing good Samaritans joining your fight. I have implemented a basic strategy structure and classes but have not yet given it full functionality so the set mode of each of the ships is set to their default. Enemy – Attack, Civilian – Trade and Planet – Defend/Trade. With the player being able to do all three.

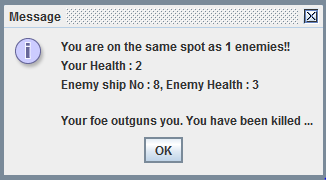
Command - I have used the command pattern by creating a Mover class which takes in all the instances of enemyShips and civilianShips and adds them to a list which is to be sent off and have their move methods activated. I have also used the Mover class to undo a single turn, it loads all the ships and causes them to move back one turn. The command pattern makes it easy to undo a large number of moves, because it stores a list of all the ships to be dealt with and allows it to be done in one go. The main game acts as the Client sending a list of the ships to be moved to the Mover class. Here the mover checks to see whether they are allowed to move. If they are the Mover class calls their moveShip() method but if not it just ignores it and leaves it until later. The Client, Invoker, Receiver model. The command pattern made it easier to create a general component that executes methods at a certain time without needing to know the owner of the method or the method parameters.

Game Programming Techniques – Due to the nature of the programming project and my background on the Games Development Degree multiple games programming techniques were used. The “Main Game Loop” is a simple technique in which the game is constantly performing a set of actions regardless of user input. This is in contrast to the GUIs event driven nature where it doesn’t do anything UNLESS the user inputs. It works together with it though by being performed on an individual thread so that other actions in game logic can be performed whilst the player is waiting. I also used Dynamic Programming to breakdown the task into more manageable subsystems . These techniques helped make programming the logic behind the game a lot easier and help control the flow.

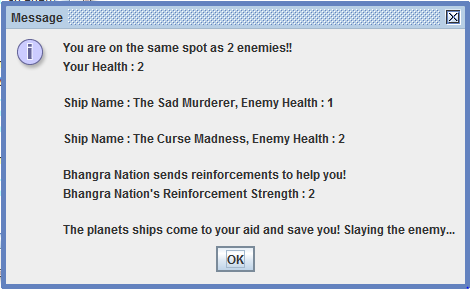


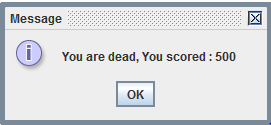
The main game screen appears as above. The player can click on any of the positions on the grid to attempt to move their ship there. Their ship is at the bottom of the screen. They can only move the ship 1 space in any direction, any attempts to move further will fail and the player will lose a turn! There are only three buttons, new game, load game and save game. New game resets the game state and creates an entirely new instance of the game. If there is a version of the game already saved, using the Load Game will restore that and all the enemies, inventories and player positions will be set to those. The Save Game button stores the current game state to a file so that it may be loaded back in via Load Game. The undo turn button does exactly that, it undoes one turn, setting each ship back to its previous position.

The list box on the right of the screen shows the XY coordinates of the grid and each positions contents. Since the game has a layered system but can only display one image at a time this is quite useful to make sure nothing is hidden. An image of a single spaceship means a single enemy is on that space whilst an image of more than one spaceship means that there is more than one. The more blocky spaceship is a civilian spaceship.

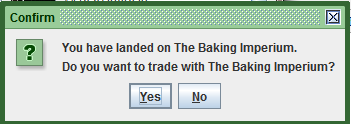


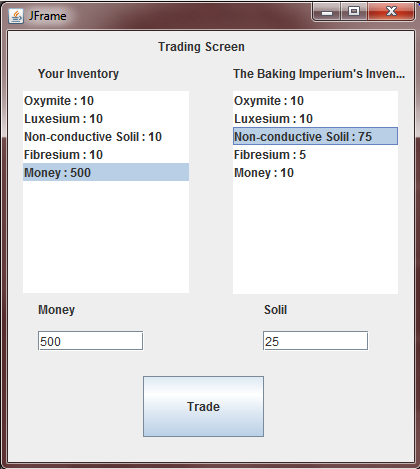
When the player moves onto a space containing an enemy or an enemy moves into a space containing the player the game detects a conflict and each players stats are shown. A message pops up telling you the number of enemies you are facing and your health value which acts as your strength. It also shows you the details on the opposing ships you are facing as well as if you have any planetary assistance as seen in the next picture. This may result in the death of the player or, if the player is stronger, the gain of resources!





If the player lands upon the same space as a planet and there are no enemies present, they will be given the option to trade with them. The player will still have the option to decline this though.





Above is the trade screen between the player and the planet that they have visited. At the top right it displays the planets name and below shows their inventory. Across from that it shows the players inventory. To trade the player must choose which items it would like to trade for what. In the above case the player wishes to buy “Non-conductive Solil” for “Money”. Each material has a different worth and this will be factored into the transaction. In this case Solil is worth 20 per piece so the player can only afford 25 in total. The player is also able to trade with Civilian spaceships that float around the universe and they show the same trading screen and have the same functionality.

