Edinburgh, 24/11/2016

Michal Bochenek  
Student ID: 40270585   
Software Engineering y.3

SET09101 - Software Development 3:

Report on coursework:   
 “Sky Wars”

Contents

[1. Introduction. 2](#_Toc468174412)

[2. Factory pattern. 2](#_Toc468174413)

[3. Observer pattern. 3](#_Toc468174414)

[4. Strategy pattern. 5](#_Toc468174415)

[5. Command pattern. 6](#_Toc468174416)

[6. GUI description. 7](#_Toc468174417)

[7. Use of threads. 8](#_Toc468174418)

[8. Programming techniques not taught. 9](#_Toc468174419)

# Introduction.

This document is a report on the Software Development 3 coursework assignment “Sky Wars”. It contains description of the design patterns used in project and explanation why and how the lack of the use of such patterns would have an influence on the design. There is also a mention about GUI use, programming technique not taught and a practical application of threads.

Please use java 1.8 and “Eclipse Java neon.1” version for maximum compatibility, also do not modify the folder structure and attached files to it.

# Factory pattern.

The practical use of factory pattern was quickly identified during planning stage for this project as the game required to spawn new ships, since each ship is an object and requires to keep unique properties and methods it would be troublesome to delegate the ship creation to the main game running class thus a factory pattern was applied to it, instead creating new object, main game class just holds an array list of ship objects.   
An external ship factory class, using a ship interface, is called whenever the creation of a new alien ship object is needed by simply telling which ship object to create (return as a new). It was also applied at the cloning of the array list of current turn ships to use it for turn history tracking and undo move procedure.

The example of use in code:

|  |
| --- |
| //using factory pattern:  **private** Ships spawnPlayerShip(**int** x, **int** y)  {  ShipFactory factory = **new** ShipFactory();  Ships ship = factory.getShip("player");    ship.setX(x);  ship.setY(y);  //for observer:  shipType = "Player";    **return** ship;  } |

# Observer pattern.

Observer pattern initially was intendent to be used as a game turn counter, as it would be much harder and less appropriate to implement a method inside GUI class which will constantly ask the game running class if the turn was updated by one, thus observer provides us with a good way of telling that the game running class has just incremented the turn counting variable by 1 without creating a new query from the GUI, since GUI should always be separated from program logics wherever possible.  
Later a new functionality was added to the observer pattern which was now observing if a new ship was spawned by the game running class and informing the user about it inside game history text area also another observer is notifying GUI about turn number.

The example implementation of an observer in code:

|  |
| --- |
| @Override  **public** **void** registerObserver(Board o) {  **this**.gameObserver.add(o);  }  @Override  **public** **void** removeObserver(Board o) {  **this**.gameObserver.remove(o);  }  @Override  **public** **void** notifyObservers(**int** latestTurn) {  **for**(Board tempObj : **this**.gameObserver){  tempObj.update(latestTurn);  }  }    @Override  **public** **void** notifyObservers(String shipType) {  **for**(Board tempObj : **this**.gameObserver){  tempObj.update(shipType);  }  }  //read current data  **public** **void** getTurnUpdate(**int** latestTurn)  {  setTurnUpdate(latestTurn);  notifyObservers(latestTurn);  }    **public** **void** getShipUpdate(String shipType)  {  setShipUpdate(shipType);  notifyObservers(shipType);  }    //set new data  **private** **void** setTurnUpdate(**int** latestTurn)  {  **this**.latestTurn = latestTurn;  }    **private** **void** setShipUpdate(String shipType)  {  **this**.shipType = shipType;  } |

# Strategy pattern.

The game requirements specified that the player ship would be using 2 different battle modes – an aggressive mode and defensive by default.   
This required to design a way of a dynamic ability to change the battle system behaviour in game without creating another instance of a player ship object which would be problematic otherwise. Without strategy pattern each time a battle was pending it would require game running class to query the player ship object about what status it has at that moment and use a battle solution algorithm depending on status of the ship.  
This would make the code of the game running class less readable and hard to extend or modify this feature in future releases, the strategy pattern comes handy in here as it separates the, rather enclosed, main logic from its additional dynamically created and used functionalities.

Example of a use of the strategy pattern in code (dynamical change executed by switching the radio button):

|  |
| --- |
| //set ship into offensive mode  rdbtnOffensiveMode = **new** JRadioButton("Offensive");  rdbtnOffensiveMode.setBackground(**new** Color(255, 255, 224));  rdbtnOffensiveMode.setEnabled(**false**);  rdbtnOffensiveMode.addActionListener(**new** ActionListener() {  **public** **void** actionPerformed(ActionEvent e)  {  battle.setShipBattleBehaviour(**new** AggressiveBehaviour());  }  });  //set ship into default defensive mode  rdbtnDefensiveMode = **new** JRadioButton("Defensive");  rdbtnDefensiveMode.setBackground(**new** Color(255, 255, 224));  rdbtnDefensiveMode.setEnabled(**false**);  rdbtnDefensiveMode.addActionListener(**new** ActionListener() {  **public** **void** actionPerformed(ActionEvent e)  {  battle.setShipBattleBehaviour(**new** DefensiveBehaviour());  }  }); |

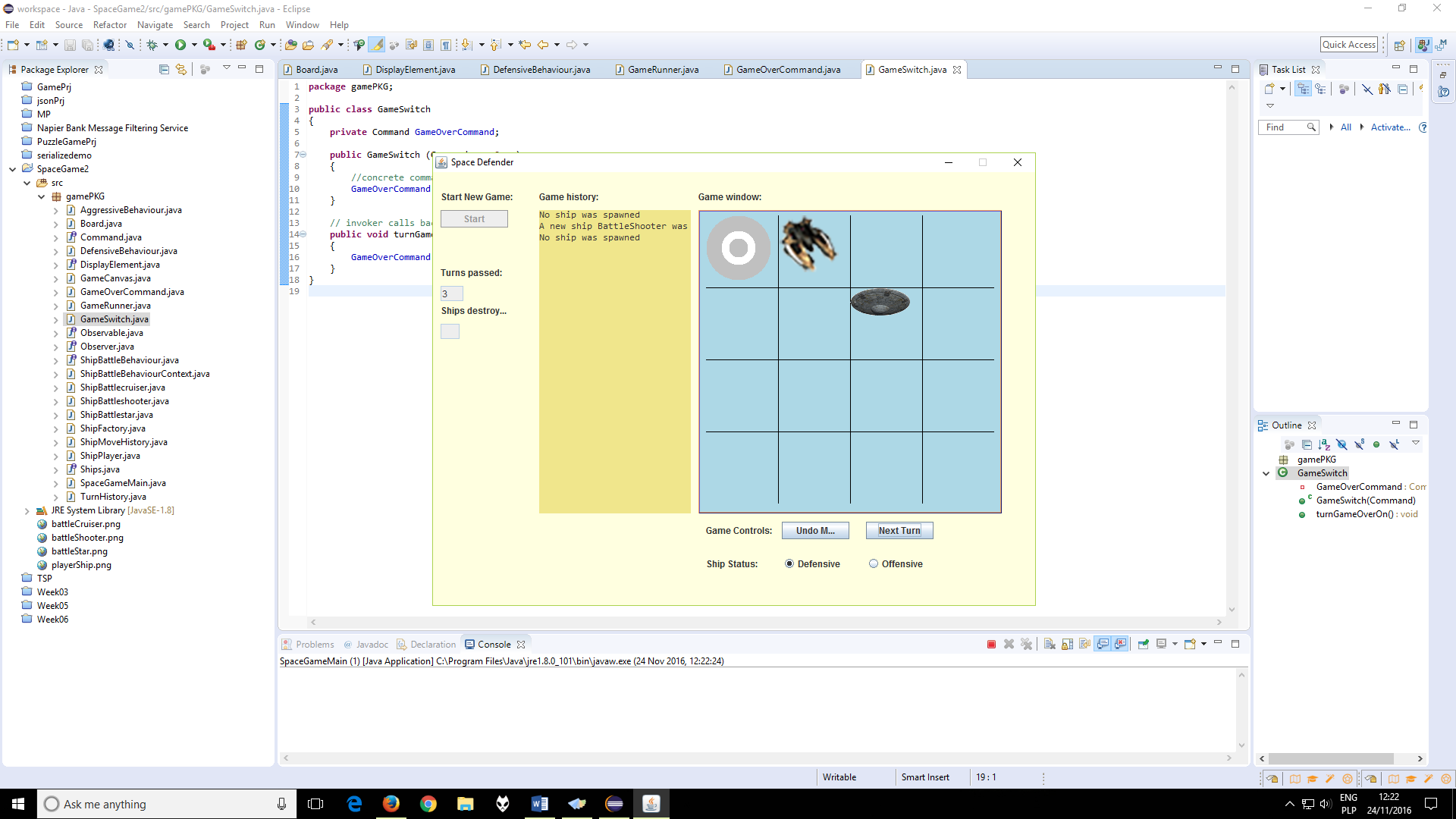
# Command pattern.

This pattern was implemented in code as a mean to inform the game that the player ship was destroyed and it should execute the game end now. Initially, while planning, it was hard to tell when or where the game end command should be placed in and how it should be able to communicate with the game logics to not create a strong association with an external class.   
A command pattern delivers us this method of creating a command interface which then can be implemented by the concrete commands and accessed and used by external classes with a command switch class. The client class has no idea how the called command is being executed it just knows how to create a call to it through the switcher.  
Since the game battles are being resolved inside game running class but the actual ship behaviour decides of the battle success or fail the game over execution switch was implemented inside the aggressive and defensive battle behaviour classes which then informs games logic that the game end has happened which proceeds it to the GUI to inform the user and stops the game.  
Post development it was also discovered that this method could have been used in as well as: start game, next turn, undo turn and battle mode command buttons, to further separate as much logics from the GUI as it was possible.

Example of the use of a command and switcher:

|  |
| --- |
| **public** **class** GameSwitch  {  **private** Command GameOverCommand;    **public** GameSwitch (Command gameOver)  {  //concrete command registers itself with the invoker  GameOverCommand = gameOver;  }    // invoker calls back concrete Command, which executes the Command on the receiver  **public** **void** turnGameOverOn()  {  GameOverCommand.execute();  }  } |

# GUI description.



a) Start button – starts new game, unlocks more GUI buttons  
b) Undo move button – repeats the last turn ship setup, but will execute new moves on the next turn, do not execute undo move button beyond player ship pawn spawn it contains a late discovered bug.  
c) Next turn button – executes next turn, redraws the board.  
d) Turns passed – counter based on observer of the game runner class, it increments by 1 with each next turn press.  
e) Game history – follows ship spawn process, based on observer of the game runner class.  
f) Game window – creates a 4x4 grid board for pawns to spawn at, UFO pawn represents player, other 3 types of images represent battlestar, battleshooter and battlecruiser ship types.  
g) Ship status defensive/offensive radio button – dynamically changes ship battle behaviour.  
h) Ships destroyed - unimplemented yet, intended as an observer for the number of enemy ships objects destroyed.

# Use of threads.

The game simplicity limited the thread use, there was a simple thread created inside the Board.java class for checking if the game has ended and it executes a command which blocks the GUI from further user next and previous turn input.  
This thread is quite useful as it doesn’t slow down the game progress and it is constantly aware of the possible game end conditions to become true at any time as it runs “in background” of the main game logics.  
Without this thread the GUI would have to query the game logics if the game is over and it would be possible only once the full turn process has ended, which could cause problems such as player ship has moved, was destroyed in battle but another ship was still created and the user could press undo turn etc.

Example of use of threading in code:

|  |
| --- |
| Runnable r = **new** Runnable()  {  **public** **void** run()  {  **boolean** flag=**false**;  //run this thread while game is not finished:  **while**(flag==**false**)  {  **boolean** gameOverFlag = game.getFlagState();  Thread.*currentThread*().getState();  **if**(gameOverFlag==**true**)  {  btnNextTurn.setEnabled(**false**);  btnUndoMove.setEnabled(**false**);  JOptionPane.*showMessageDialog*(**null**, "The ship was destroyed by overwhelming alien forces!!!");  flag = **true**;    Thread.*currentThread*().interrupt();  }  }  }  }; |

# Programming techniques not taught.

The only technique which had to be learned by the student was identified as cloning of java objects.   
The game undo turn functionality could be achieved by creating an array of ship objects arrays which would gather up the ship status on the game board each turn.  
During early implementation it was discovered that the despite putting ships list array on the turn array it was producing no effect while pressing undo button, thus a research was conducted and it provided an answer to the question why such thing happens. Each array of objects was just a reference to the original one and it was changing its status together with the original array list each turn.  
Cloning was required to copy the objects from the array list and save them as new instances to remove the memory referencing to the original ship object.  
Ship factory pattern was also being useful in the cloning process to shorten it overall.

|  |
| --- |
| //create a ship clone  **private** Ships cloneShip(Ships realShip)  {  **int** x, y;  String type;    x = realShip.getX();  y = realShip.getY();  type = realShip.type().toLowerCase();    ShipFactory factory = **new** ShipFactory();  dummyShip = factory.getShip(type);  dummyShip.setX(x);  dummyShip.setY(y);  **return** dummyShip;  }  //get original ship list  **public** **void** cloneShipList(ArrayList<Ships> shipsList)  {  **for**(**int** i=0; i<shipsList.size(); i++)  {  //get existing ship and make a copy, put it on new arraylist, return list:  clonedShipList.add(cloneShip(shipsList.get(i)));  }  } |

Example of use of the object cloning in code:

The other technique worth mentioning was painting the game graphics using Java Canvas which represented each turn current status on screen for users view.