

Sky wars coursework report

Sebastian Schmitt 40409930
SET11103 2018-9 TR2 001 - Software Development 2

Introduction

The game consists of a board (the sky), one master ship and several enemy ships. The ships can move to the locations next to their current location (like the king in chess). The game starts with the master ship randomly allocated on the board. Every time the master ship moves there is a 1 to 3 chance that an enemy ship is spawning in the top left tile. After the master ship moves, all the enemy move as well. In the case that the master ship is on the same location than enemy ships, a battle is being fought. When the master ship is in the same location with less than 2 enemy ships (3 in offensive game mode) then the enemy ships are being destroyed and removed from the board, otherwise the master ship loses the battle and the game is over. The game mode can be switched during the game from defensive to offensive. The defensive mode is the default once the game is started.

Overall design and implementation

The board is an 4x4 grid consisting of 16 tiles on it. This is being realized with an array of objects of the type *Tile*. Every tile has a coordinate, starting from 0 in the top left and resulting in the bottom right tile with the coordinate 15. The board and also every tile record if a master ship is placed on it and also how many and which enemy ships are located on them at the moment. Every time a ship is moved the records on the board and also on the tile are being updated. The moves are realised with a *Move* class and the transition to a new state with a *Transition* class. The game logic itself is in the *JSkyWars* class. It uses all the methods from the *Move*, *Transition*, *Tile* and *Board* classes. In addition, the conflict / battle, the spawn, save game and load game methods are implemented in the *JSkyWars* class. Also the states of a game are being managed in this class (whose move is it, the game mode, round count etc.). The graphical user interface (GUI) has one instance of the *JSkyWars* class as a local instance variable which is updated, once the user interacts with the GUI (e.g. presses a button). Further details about the design and implementation are being explained later in the report.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	15	15

Figure 1: Logical representation of the board

Ship design

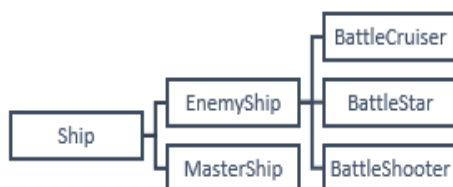


Figure 2: Ship inheritance

The “mother” class of every ship is the *Ship* class. The classes *MasterShip* and *EnemyShip* inherit all properties and methods from it. Additionally, the *EnemyShip* class has three child classes itself; the *BattleStar*, *BattleShooter* and *BattleCruiser* classes. Every child class of the *Ship* only differs in the name, the *ShipType* property and visual representation on the board.

Properties and methods

Every ship instance has a local class variable with the possible moves (*POSSIBLE_MOVES*) which are in this case -5, -4, -3, -1, 1, 3, 4, 5. These offsets result in the coordinates next to the current coordinate / location of the ship (visualised in Figure 4). The instance variables are the current coordinate of the ship and the ship type which is managed in the enum *ShipType*. The class also has three methods ensure that the ship moves

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 4: Possible moves visualisation

Ship	
uniqueID	UUID
POSSIBLE_MOVES	int[]
Ship(ShipType, int)	
calculateLegalMoves(Board)	Collection<Move>
moveShip(Move)	Ship
isFirstColumnLegalMove(int, int)	boolean
isFourthColumnLegalMove(int, int)	boolean
shipType	ShipType
coordinate	int

Figure 3: Overview Ship class

in a way as intended in the concept. The *calculateLegalMoves* method calculates all legal moves with the help of the *POSSIBLE_MOVES* array, *isFirstColumnLegalMove*, *isFourthColumnLegalMove* and *isValidCoordinate* (implemented in the *Board* class). The methods *isFirstColumnLegalMove* and *isFourthColumnLegalMove* check if the ship is currently located in the first column (position 0, 4, 8 or 12) or the fourth column (3, 7, 11, 15) and ensures that the ship can't move over the board edge (e.g. 15 → 12). The *moveShip* method takes the ship and gives it a new coordinate according to the passed *Move* object. Also the sub-classes of Ship have their own *toString()*

methods. The string representations are later transformed to the actual visualisation of the GUI. The single ships are represented by the following strings:

- Master ship → MS
- Battle shooter → SH
- Battle star → ST
- Battle cruiser → CR

The same applies to other classes which are visualised by the GUI like the *Board* and the *Tile* class.

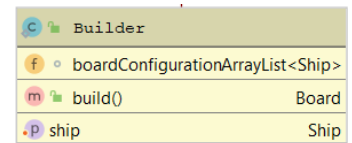
Every ship also has a unique ID, generated by the *UUID* class. Without the unique ID in the case that several enemy ships of the same ship type are located on one tile, they would look like equal objects, because they have the same coordinate and ship type. The enemy ships are managed by an *ArrayList*. Without the unique ID they would disappear when the board and tiles are generated again. When an object of the class *MasterShip*, *BattleCruiser*, *BattleStar* or *BattleShooter* are initiated then the ship type is being automatically assigned.

Other classes

Board

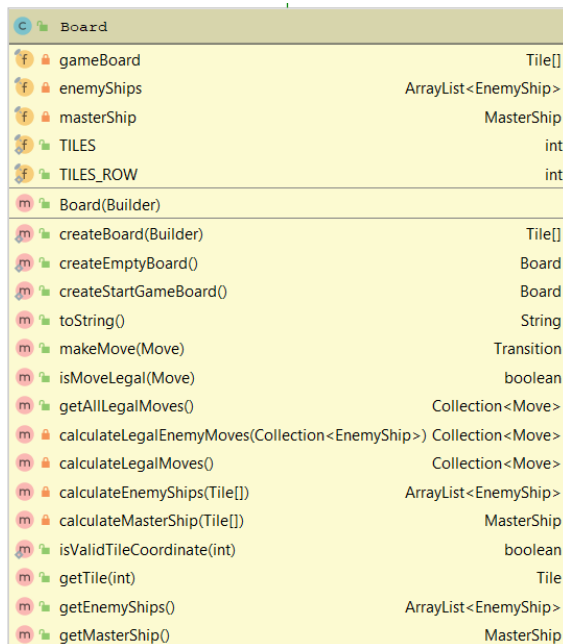
An essential part in enabling to manage and create board is the Builder sub-class. Every builder holds a list with ships which then are put on a new Board instance when the build() method is being called.

As already mentioned earlier, the board is consisting of 16 tiles (recorded in the TILE variable) resulting in a 4x4 grid (number of rows recorded in the variable TILES_ROW). Every board also holds a record with all ships on it (the instance variables enemyShips and masterShip). There are three different ways of creating a board:



Builder	
boardConfigurationArrayList<Ship>	
build()	Board
ship	Ship

Figure 5: Overview Builder class



Board	
gameBoard	Tile[]
enemyShips	ArrayList<EnemyShip>
masterShip	MasterShip
TILES	int
TILES_ROW	int
Board(Builder)	
createBoard(Builder)	Tile[]
createEmptyBoard()	Board
createStartGameBoard()	Board
toString()	String
makeMove(Move)	Transition
isMoveLegal(Move)	boolean
getAllLegalMoves()	Collection<Move>
calculateLegalEnemyMoves(Collection<EnemyShip>)	Collection<Move>
calculateLegalMoves()	Collection<Move>
calculateEnemyShips(Tile[])	ArrayList<EnemyShip>
calculateMasterShip(Tile[])	MasterShip
isValidTileCoordinate(int)	boolean
getTile(int)	Tile
getEnemyShips()	ArrayList<EnemyShip>
getMasterShip()	MasterShip

Figure 6: Overview Board class

1. Create an empty board:

This gives back a board with just empty tiles with no ships on it. This is used when the game is started and the GUI appears.

2. Create a start game board:

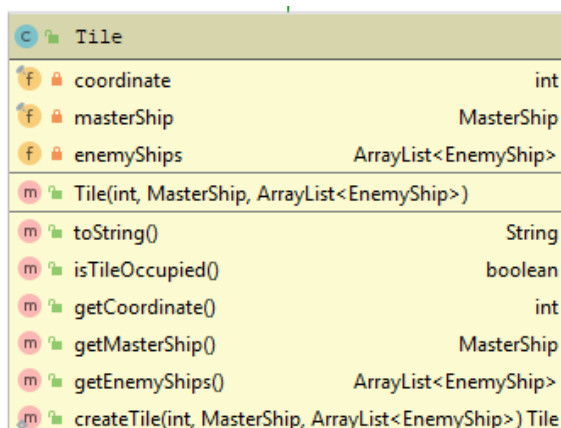
This gives back a board with just the master ship randomly placed on it. This is used when the game is started and “Start new game” button in the GUI is pressed.

3. Create board:

This method is being used, whenever the state of the board changes. This can be triggered by a ship moving or ships being removed due to a conflict. Other than the other two methods, a builder is handed to this method with the ships which then later exist on the new board.

Next to these methods, also a set of methods exist which calculate legal moves of the ships on the board. For example, the isValidCoordinate method ensures that the ships only move to tiles that actually exist. The makeMove method takes a Move object and transforms it with the help of the Transition class into a new board. Next to the already covered methods several methods to access the board properties exist.

Tile



Tile	
coordinate	int
masterShip	MasterShip
enemyShips	ArrayList<EnemyShip>
Tile(int, MasterShip, ArrayList<EnemyShip>)	
toString()	String
isTileOccupied()	boolean
getCoordinate()	int
getMasterShip()	MasterShip
getEnemyShips()	ArrayList<EnemyShip>
createTile(int, MasterShip, ArrayList<EnemyShip>)	Tile

Figure 7: Overview Tile class

Like a board, a *Tile* object also holds a record with all ships on it. In addition, it also has a coordinate assigned to it.

The *toString* method generates an “-” string for empty tiles and a concatenated string of the ships on a tile for non-empty tiles (e.g. MSST for a tile with a master ship and a battle star).

No setters exist and the getters just return the attributes of an object.

Outside of the class, *Tile* Objects are created by the static method *createTile* instead of the constructor. This is a general design pattern, also used in the other classes. It has the following advantages:

- Clearer name
- Better control of inheritance patterns
- Hide implementation details

Move and Transition

The *Move* class is an abstract class. It has two sub-classes *NormalMove* and *NullMove*. Again moves are only being created by the *createMove* method for the reason explained earlier. When a move is illegal (moves out of the grid or wants to skip a tile) then a *NullMove* is being created. This means the ship is not being moved. In case the move is legal a *NormalMove* object is being created. In the class also an enum can be found with the types “DONE” and “ILLEGAL_MOVE”. These are later used in the *Transition* class.

The *Transition* class is more or less an interstep between a move and a resulting board. This could be solved different, but appeared cleaner like this when I thought of the concept. So I just kept it like this.

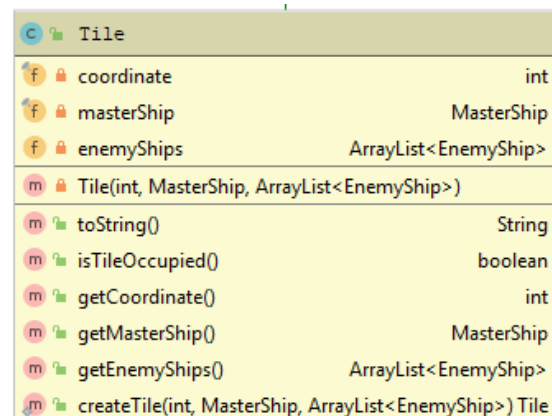


Figure 8: Overview Move class

The transition class has no other methods than the constructor, getters and setters.

Log

Like the *Transition* class, the *Log* class just consists of an arrayList where the destroyed ships are recorded in and some setters and getters. This could be done in the *JSkyWars* class itself, but it is cleaner to separate it and helps to better understand what is done.

JSkyWars

JSkyWars		
f	board	Board
f	isGameStarted	boolean
f	POSSIBLE_MOVE	int[]
m	main(String[])	void
m	getRandomMoveOffset	int
m	startNewGame()	void
m	moveShip(Ship)	void
m	moveMasterShip()	void
m	moveEnemyShips()	void
m	spawnEnemyShip()	void
m	conflict()	void
m	newRound()	void
m	saveGame()	void
m	loadGame()	void
m	changeMasterShipModeDefensive	void
p	enemyMove	boolean
p	log	Log
p	gameOver	boolean
p	gameStarted	boolean
p	masterMove	boolean
p	roundCount	int
p	board	Board
p	masterShipModeDefensive	boolean

Figure 9: Overview JSkyWars class

The JSkyWars class is the place where all the methods and properties of the other classes are brought together. The class is then used as a single association by the GUI.

All the game states are managed in the JSkyWars class as well.

The different game states are:

- Is the game started
- The log of the destroyed ships
- Is the game over
- Is it the master ships move?
- Is it the enemy ships move?
- Which round is it currently
- Is the master ship mode offensive or defensive?
- The current board

Also the methods for a conflict / battle, to save or load a game, spawn an enemy ship, start a game, start a new round, change the master ship mode or move any ships are implemented here.

All these methods are later then triggered by an event listener (in this case button clicks) within the GUI. Also the GUI is updated once the state of the board changes (implemented in the GUI with *ActionListener*).

Graphical user interface



The graphical user interface (GUI) consists of the following components with its functions:

Menu

Consists of a “Save Game” and “Load Game” button. With The “Save Game” button these you can export the current game state with all the ships on the board whose turn it is, the current round, the master ship mode and the destroyed ship log. These information can then later be imported again with the “Load Game” button.

Figure 10: The menu with the save and load function

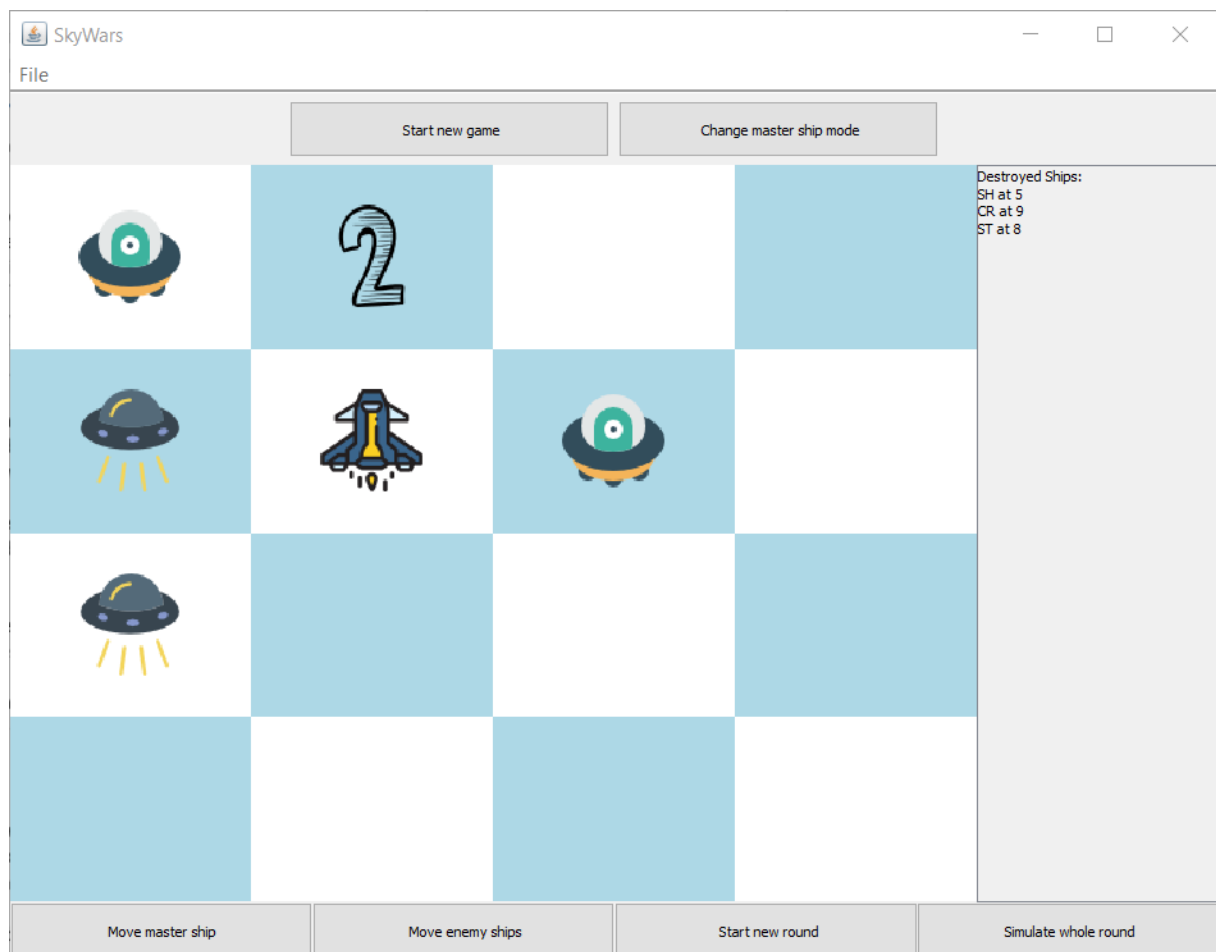


Figure 11: The graphical user interface

Start new game button

When the GUI is loaded the board is empty. With this button the master ship is randomly placed on the board.

Change master ship mode button

With this button the master ship mode can be changed to offensive and vice versa. The user is notified over the change with a pop up window.

Move master ship button

With this button the master ship is being moved to a random position surrounding its current position.

Move enemy ships button

When pressing this button all enemy ships currently on the board are being moved.

New round button

When the master ship is currently located on a tile where enemy ships are also located then a conflict is being fought. This results in either removing the enemy ship / ships or in ending the game. Removed ships are being logged in the right are of the GUI. Also a new round starts.

Simulate whole round button

A whole round is simulated. This means that all ships are being moved, the conflict is being fought and also a new round is being started. '

Destroyed ships log








Here the destroyed ships with the location where they were destroyed are listed.





The board

The board consists of 16 tiles and the ships currently in the game. After every move the board is being updated.

Ship representation

When creating the tile representation. The GUI takes the string representations of the tiles and then prints the following on the tiles (in case the tile is not empty):

Ships on tile	Graphical representation
Master ship only	
Battle cruiser only	
Battle star only	
Battle shooter only	
Master ship and one enemy ship	 1
Master ship and two enemy ships	 2
Master ship and three enemy ships	 3

Master ship and more than three enemy ships	
Two enemy ships	
Three enemy ships	
More than three enemy ships	

All the icons are retrieved from www.flaticon.com. The creators of the icons are:

- Pixel Buddha
- Smashicons
- Darius Dan
- Freepik