ALMA MATER STUDIORUM
UNIVERSITY OF BOLOGNA

First degree course in Engineering and Computer Science

SeaRumble:
a battleship game

Object-oriented programming project of

Daniele Mazzotti
Michel Paoloni
Nicola Di Berardino

Course professors

Mirko Viroli
Andrea Santi
Danilo Pianini

Academic year 2013/2014
Last revision March 17, 2014

1 Problem Analysis

SeaRumble consists in the popular battleship game for two players where you try to guess the location of some ships your opponent has hidden on a grid. Players take turns calling out a row and column, attempting to name a square containing enemy ships.
To make the game more original and maybe even funnier to play, some modifications have been added: *special actions* and *weather conditions*.

Special actions, which are unlocked after some conditions are verified:

- **Super missile**: rocket which shoots a cluster of bombs, striking more than one square.
- **Radar**: discovers the chosen square and the ones next to it.
- **Shield**: protects every ships from one hit. Using it do not cause to lose the turn.

Wheather conditions, which affects game-play:

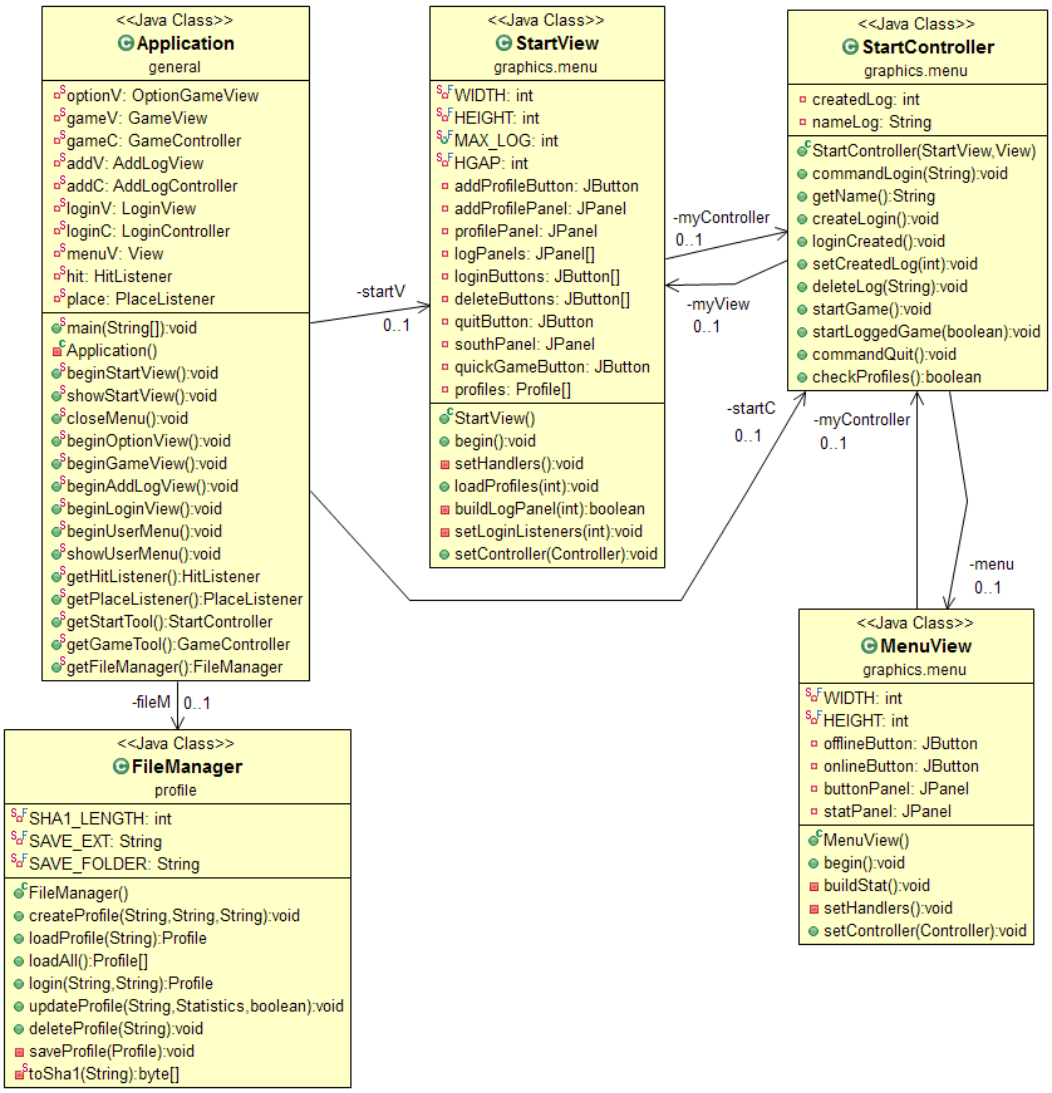
- **Sunny**: no effects.
- **Foggy**: every square is set as *undiscovered* until the fog is not disappeared. Players can't see the previous hit markers on the map, except for already sunk ships.
- **Stormy**: every turn a lightning hits a square. If a ship is hit, the square is set as *discovered* and the ship as *hit*.

The game has two modes: single-player and LAN multi-player. In single-player mode the user plays against an artificial intelligence, while in multi-player games can play against a friend over LAN. It is also possible to chat with the opponent after connection between the two players has been established. In order to play multi-player, users must necessarily create a profile. Every profile is protected with password and stores all the statistics of the player. They are saved on the local hard-drive in the `Profiles` directory and updated after a new game has been played. In the main window of the game there is the ranking of all the local profiles, ordered by the maximum score players have achieved during games. Main statistics are displayed and updated every round during games, whether single-player or multi-player mode has been chosen. When the game is finished, a summary page pops out.

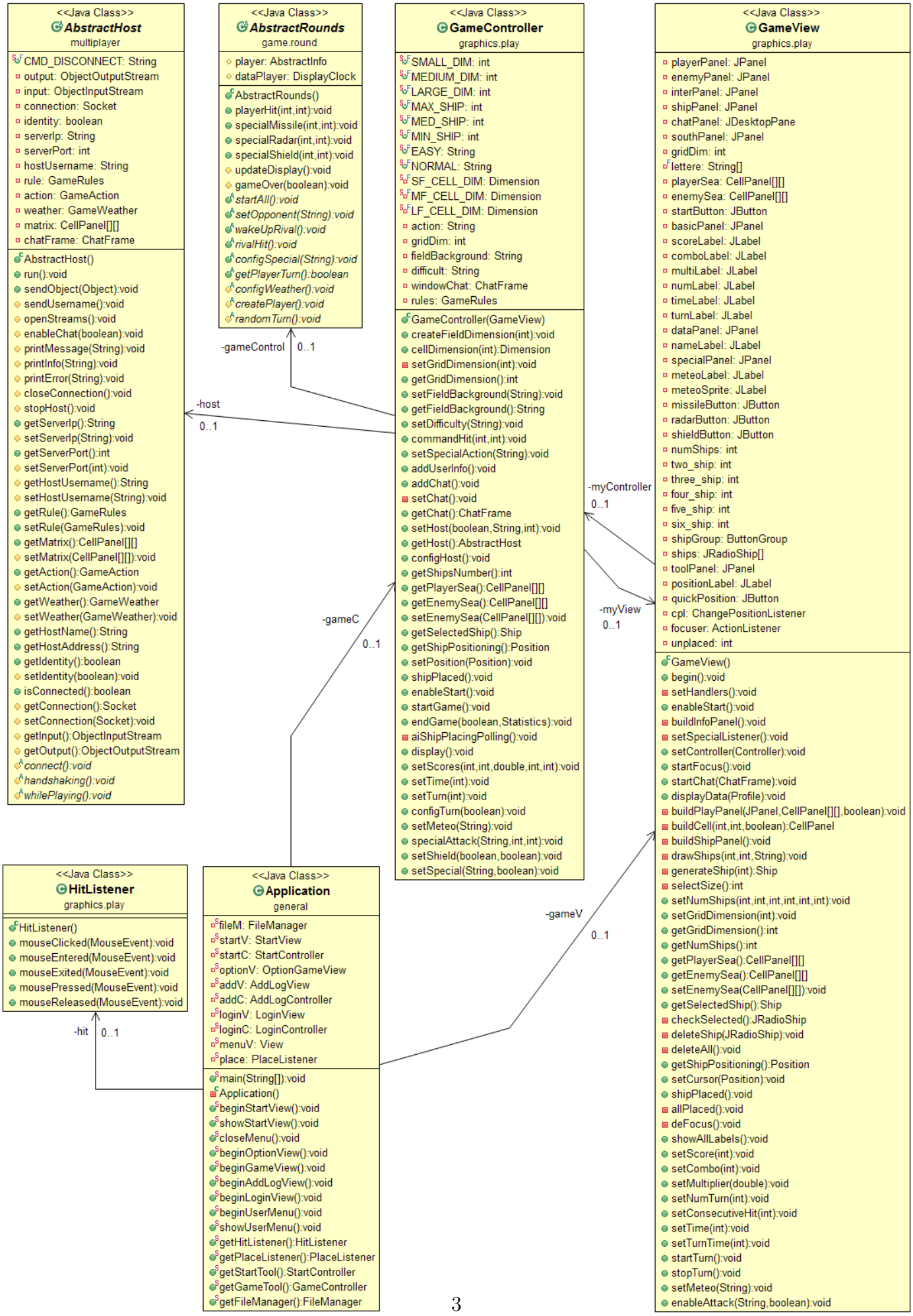
Since the battleship game itself doesn't require any particular interaction with the player or any complex graphics, the whole game it is based on a simple GUI.

2 Design scheme

Now follow the main UML schemes of the application. The UML diagrams have been drawn using *ObjectAid UML explorer* tool for Eclipse IDE.



There are several View classes that are associated with one Controller which contains some methods called by the View according to the *Observer pattern*. Additionally, these controllers classes allow others to access the View. The **Application** class provides all the statics methods for the Views managing, hence from every class it is possible to call an Application method to open a certain window.
The first window is **StartView** which shows the lists of all profiles. From here, users can play a quick game, create a new profile (until the limit is not reached) or log in, switching to the **MenuView** with all the statistics of the player. From this window, they can still play single-player, in which case their statistics will be saved, or choose a multi-player game, challenging another player.


<<Java Class>>
AbstractHost
multiplayer
CMD_DISCONNECT: String
output: ObjectOutputStream
input: ObjectInputStream
connection: Socket
identity: boolean
serverIp: String
serverPort: int
hostUsername: String
rule: GameRules
action: GameAction
weather: GameWeather
matrix: CellPanel[][]
chatFrame: ChatFrame
AbstractHost()
run():void
sendObject(Object):void
sendUsername():void
openStreams():void
enableChat(boolean):void
printMessage(String):void
printInfo(String):void
printError(String):void
closeConnection():void
stopHost():void
getServerIp():String
setServerIp(String):void
getServerPort():int
setServerPort(int):void
getHostUsername():String
setHostUsername(String):void
getRule():GameRules
setRule(GameRules):void
getMatrix():CellPanel[][]
setMatrix(CellPanel[][]):void
getAction():GameAction
setAction(GameAction):void
getWeather():GameWeather
setWeather(GameWeather):void
getHostName():String
getHostAddress():String
getIdentity():boolean
setIdentity(boolean):void
isConnected():boolean
getConnection():Socket
setConnection(Socket):void
getInput():ObjectInputStream
getOutput():ObjectOutputStream
connect():void
handshaking():void
whilePlaying():void
<<Java Class>>
AbstractRounds
game.round
player: AbstractInfo
dataPlayer: DisplayClock
AbstractRounds()
playerHit(int,int):void
specialMissile(int,int):void
specialRadar(int,int):void
specialShield(int,int):void
updateDisplay():void
gameOver(boolean):void
startAll():void
setOpponent(String):void
wakeUpRival():void
rivalHit():void
configSpecial(String):void
getPlayerTurn():boolean
configWeather():void
createPlayer():void
randomTurn():void
-gameControl
0..1
-host
0..1
<<Java Class>>
GameController
graphics.play
SMALL_DIM: int
MEDIUM_DIM: int
LARGE_DIM: int
MAX_SHIP: int
MED_SHIP: int
MIN_SHIP: int
EASY: String
NORMAL: String
SF_CELL_DIM: Dimension
MF_CELL_DIM: Dimension
LF_CELL_DIM: Dimension
action: String
gridDim: int
fieldBackground: String
difficult: String
windowChat: ChatFrame
rules: GameRules
GameController(GameView)
createFieldDimension(int):void
cellDimension(int):Dimension
setGridDimension(int):void
getGridDimension():int
setFieldBackground(String):void
getFieldBackground():String
setDifficulty(String):void
commandHit(int,int):void
setSpecialAction(String):void
addUserInfo():void
addChat():void
setChat():void
getChat():ChatFrame
setHost(boolean,String,int):void
getHost():AbstractHost
configHost():void
getShipsNumber():int
getPlayerSea():CellPanel[][]
getEnemySea():CellPanel[][]
setEnemySea(CellPanel[][]):void
getSelectedShip():Ship
getShipPositioning():Position
setPosition(Position):void
shipPlaced():void
enableStart():void
startGame():void
endGame(boolean,Statistics):void
aiShipPlacingPolling():void
display():void
setScores(int,int,double,int,int):void
setTime(int):void
setTurn(int):void
configTurn(boolean):void
setMeteo(String):void
specialAttack(String,int,int):void
setShield(boolean,boolean):void
setSpecial(String,boolean):void
-myController
0..1
-myView
0..1
<<Java Class>>
GameView
graphics.play
playerPanel: JPanel
enemyPanel: JPanel
interPanel: JPanel
shipPanel: JPanel
chatPanel: JDesktopPane
southPanel: JPanel
gridDim: int
lettere: String[]
playerSea: CellPanel[][]
enemySea: CellPanel[][]
startButton: JButton
basicPanel: JPanel
scoreLabel: JLabel
comboLabel: JLabel
multiLabel: JLabel
numLabel: JLabel
timeLabel: JLabel
turnLabel: JLabel
dataPanel: JPanel
nameLabel: JLabel
specialPanel: JPanel
meteoLabel: JLabel
meteoSprite: JLabel
missileButton: JButton
radarButton: JButton
shieldButton: JButton
numShips: int
two_ship: int
three_ship: int
four_ship: int
five_ship: int
six_ship: int
shipGroup: ButtonGroup
ships: JRadioShip[]
toolPanel: JPanel
positionLabel: JLabel
quickPosition: JButton
cpl: ChangePositionListener
focuser: ActionListener
unplaced: int
GameView()
begin():void
setHandlers():void
enableStart():void
buildInfoPanel():void
setSpecialListener():void
setController(Controller):void
startFocus():void
startChat(ChatFrame):void
displayData(Profile):void
buildPlayPanel(JPanel,CellPanel[][],boolean):void
buildCell(int,int,boolean):CellPanel
buildShipPanel():void
drawShips(int,int,String):void
generateShip(int):Ship
selectSize():int
setNumShips(int,int,int,int,int,int):void
setGridDimension(int):void
getGridDimension():int
getNumShips():int
getPlayerSea():CellPanel[][]
getEnemySea():CellPanel[][]
setEnemySea(CellPanel[][]):void
getSelectedShip():Ship
checkSelected():JRadioShip
deleteShip(JRadioShip):void
deleteAll():void
getShipPositioning():Position
setCursor(Position):void
shipPlaced():void
allPlaced():void
deFocus():void
showAllLabels():void
setScore(int):void
setCombo(int):void
setMultiplier(double):void
setNumTurn(int):void
setConsecutiveHit(int):void
setTime(int):void
setTurnTime(int):void
startTurn():void
stopTurn():void
setMeteo(String):void
enableAttack(String,boolean):void
-gameC
0..1
<<Java Class>>
HitListener
graphics.play
HitListener()
mouseClicked(MouseEvent):void
mouseEntered(MouseEvent):void
mouseExited(MouseEvent):void
mousePressed(MouseEvent):void
mouseReleased(MouseEvent):void
-hit
0..1
<<Java Class>>
Application
general
fileM: FileManager
startV: StartView
startC: StartController
optionV: OptionGameView
addV: AddLogView
addC: AddLogController
loginV: LoginView
loginC: LoginController
menuV: View
place: PlaceListener
main(String[]):void
Application()
beginStartView():void
showStartView():void
closeMenu():void
beginOptionView():void
beginGameView():void
beginAddLogView():void
beginLoginView():void
beginUserMenu():void
showUserMenu():void
getHitListener():HitListener
getPlaceListener():PlaceListener
getStartTool():StartController
getGameTool():GameController
getFileManager():FileManager
-gameV
0..1

3 Packages organization

- **game** contains some general classes used during game-play such as the *Coordinates*, *PlaceShips* and *WeatherManager* classes.
  - **game.ai** contains the classes relatively at the Artificial Intelligence.
  - **game.player** contains all the classes relative to players.
  - **game.round** contains all the classes for the turn management for both single-player and multi-player modes.
- **general** contains the *Application* class within the *main* method, the *View* and *Controller* interfaces, the *AbstractView*, *User* and *SeaSound* classes.
- **graphics** contains the classes for every application's window.
  - **graphics.login** contains all the classes and controllers for the login view.
  - **graphics.menu** contains all the classes and controllers for the menu view.
  - **graphics.play** contains all the classes for the actual game views such as the options and the game window with all its controllers.
- **profile** contains all the classes used to handle profiles, statistics and data saving.
  - **profile.exception** contains a couple of Exceptions used in the profile management.
- **multiplayer** contains the classes needed for the communication between two hosts.
  - **multiplayer.packet** contains all the type of messages/packets that can be exchanged between the two hosts.
  - **multiplayer.time** contains a class used to start a countdown timer in multi-player games.

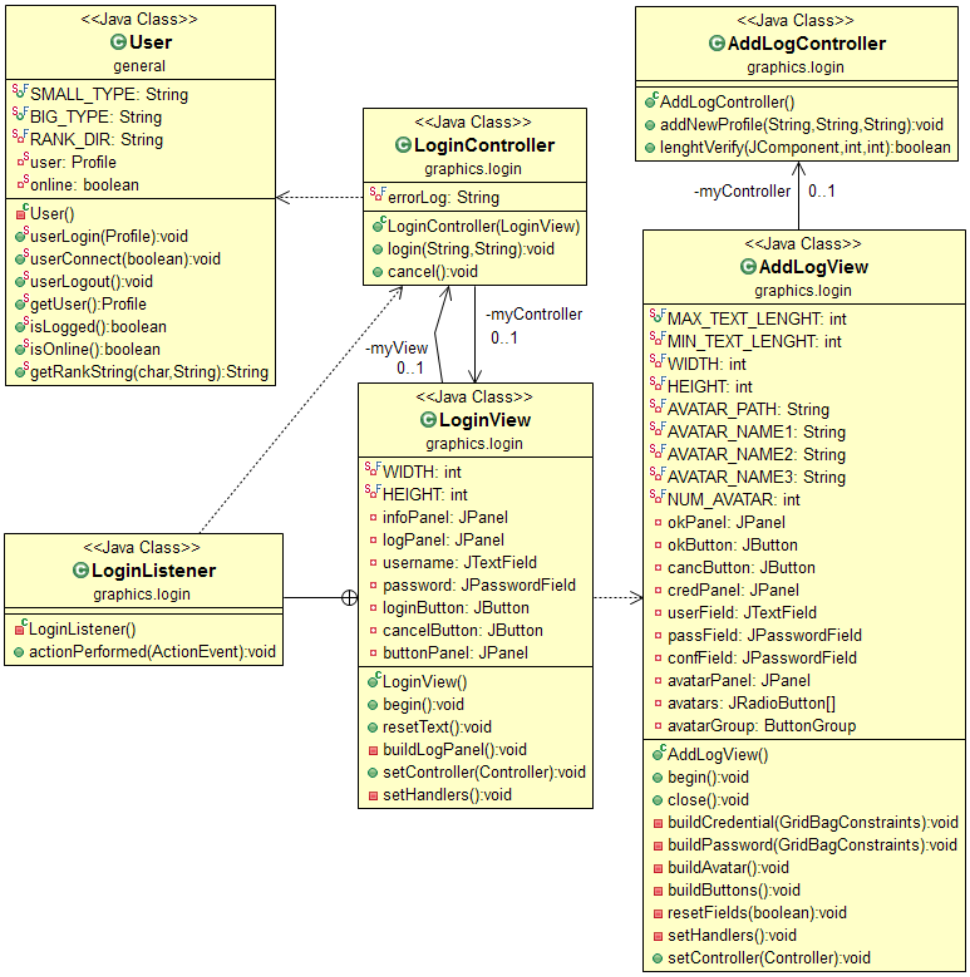
4 Division of work

- Daniele Mazzotti made the whole graphics and GUI part, including the chat frame for multi-player (**graphics**, **general** packages and sub-packages).
- Michel Paoloni made the data saving, the statistics management and the multi-player part. He also drew some of the sprites of the game (**profile**, **multiplayer** packages and sub-packages).
- Nicola Di Beradino made the core of the game, the turns implementation and the Easy difficulty of the Artificial Intelligence in single-player mode (**game** package and sub-packages).

The whole team developed the Normal difficulty (**game.ai.NormalAI**).

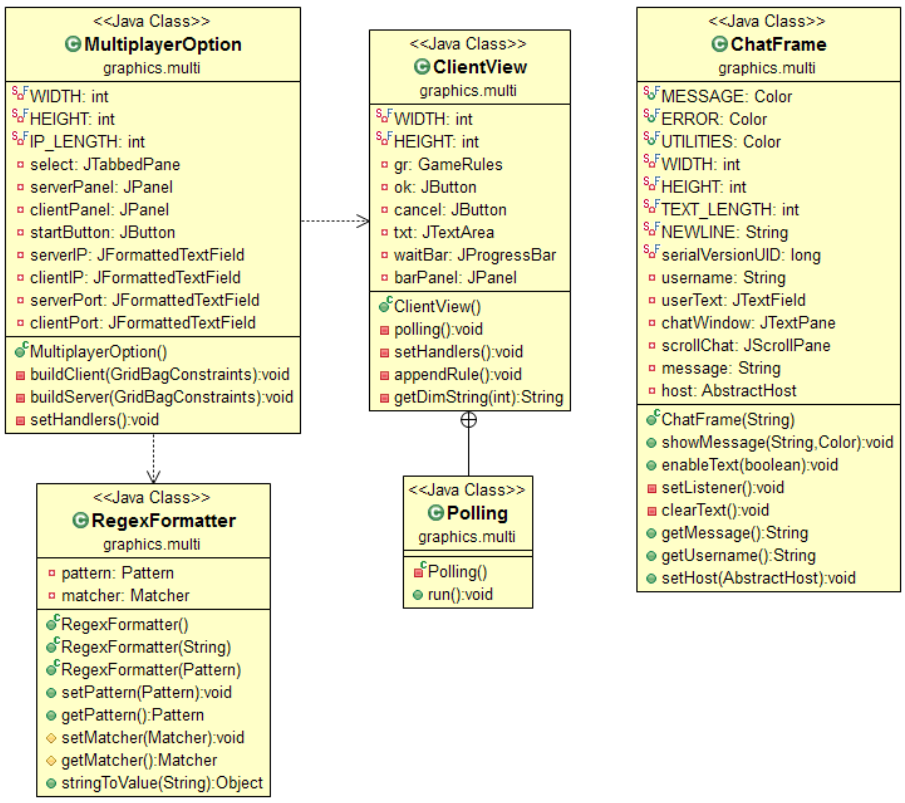
4.1 Division of work: Daniele Mazzotti

Diagram related to the login management (packages: **graphics.login**, **general**):



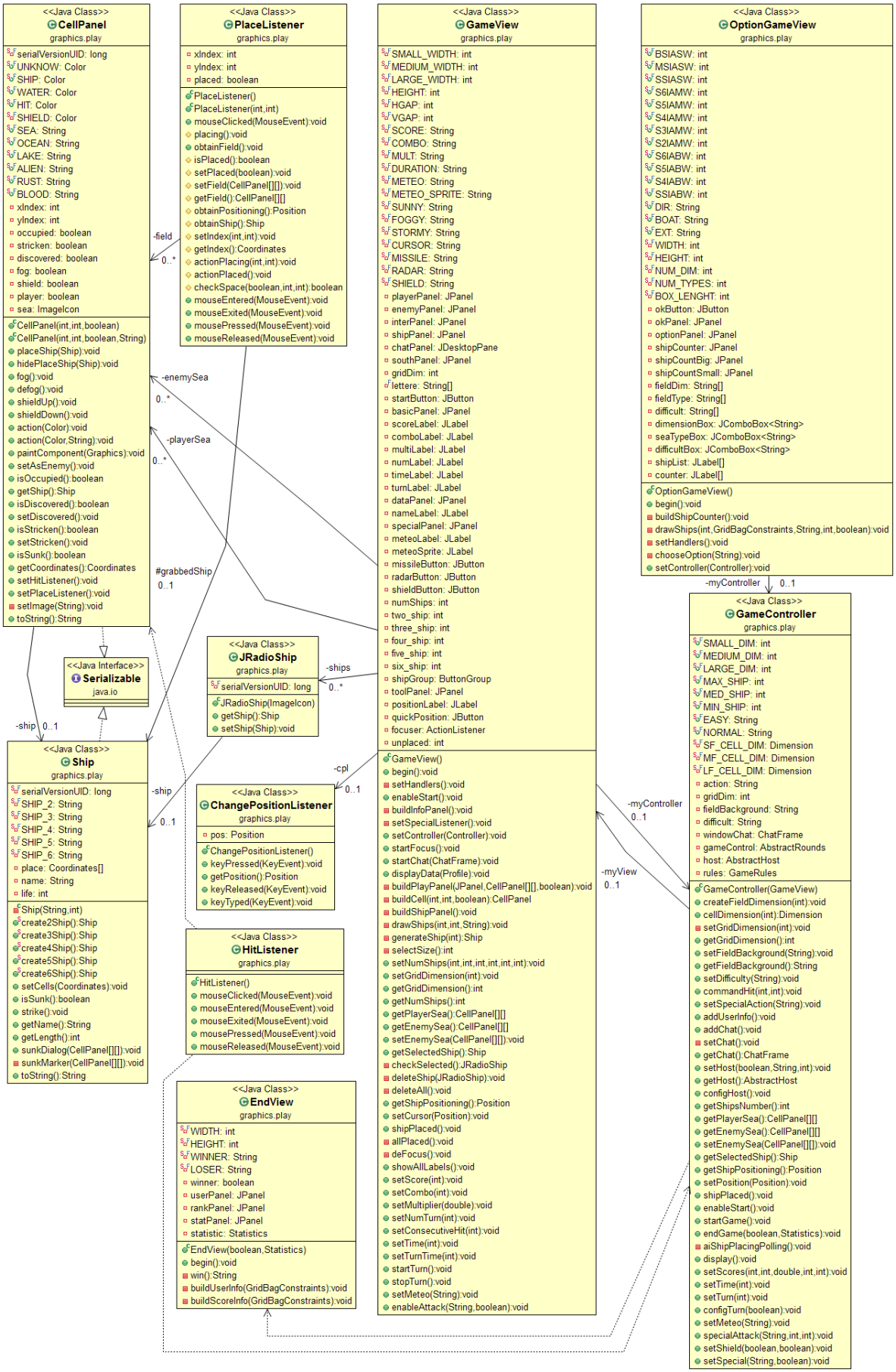
From the main window users can create a new profile or logging in with an existing one. The **AddLogView** class draws the window to create a new profile, with user-name and password fields and an avatar to choose among the predefined ones. On the other side, the **LoginView** class draws the window to log in. If the login fields are incorrect, an error message pops out and the user has to put the credentials again. When he/she is logged, his/her profile is registered using the User class which has some static fields to get at any time the information about that user (i.e. if he/she's logged).

Diagram related to the multi-player view (package: **graphics.multi**):



When the multi-player mode is selected, its options windows is shown, created by the **MultiplayerOption** class. Using a *JTabbedPane* the player can choose to be a server and select the options for the game or a client. The IP is checked if valid using a *Regular Expression* of a *Formatter* of Sun Microsystem found on the Internet. The chat frame present during multi-player games extends from *JPanel* and has a *JTextPanel* where the chat history is shown and a *JTextfied* to write messages.

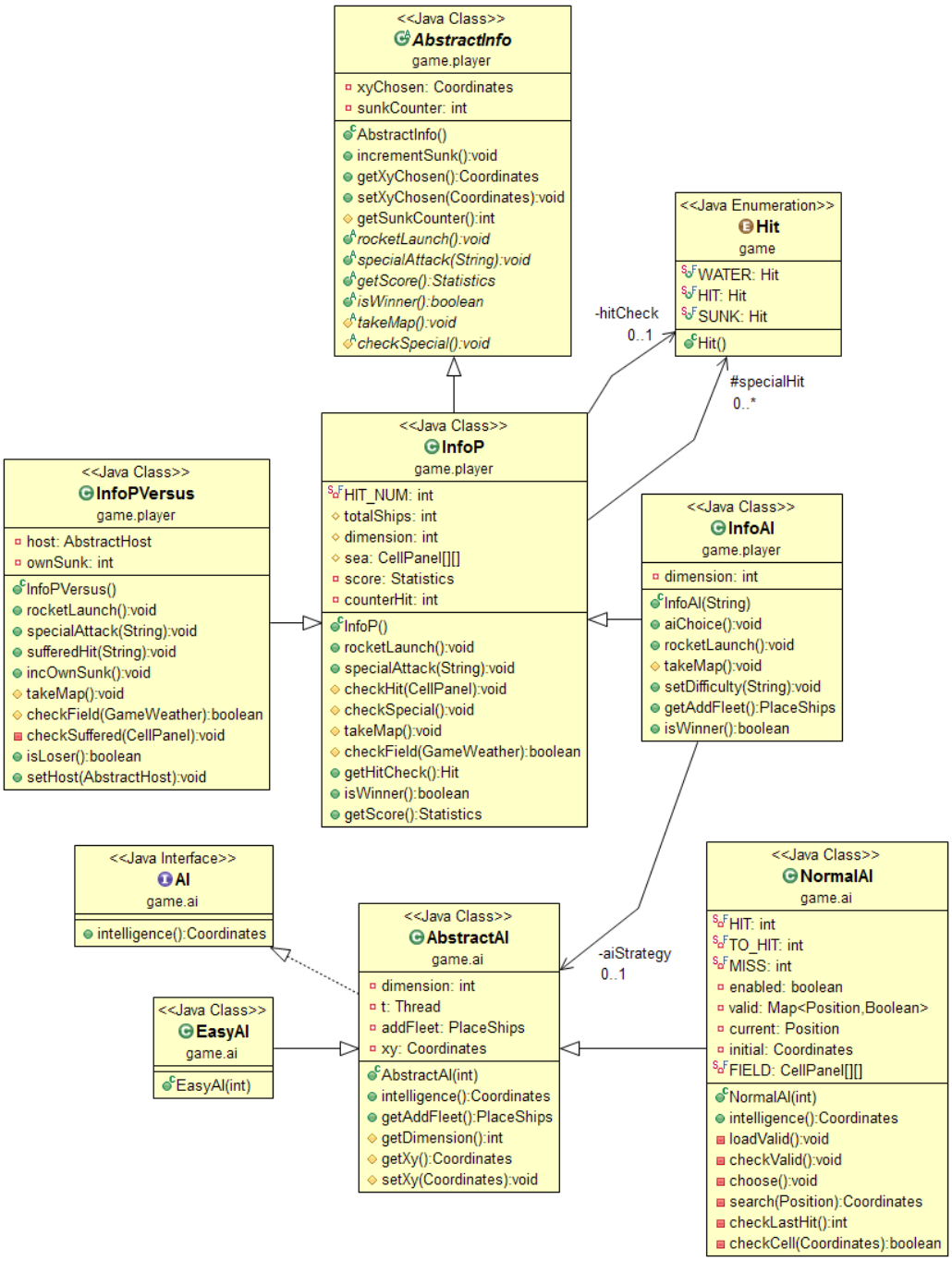
Diagram related to the game view (package: **graphics.play**):



The **GameView** class builds the game window and is associated with the **GameController**. Before the game starts, the game options view is shown, allowing players to choose the map size, its color and the difficulty (only in single-player). This data is passed to the GameController which will make the GameView build the proper field of game, that is a **CellPanel** matrix where, ships, which are instantiated using the **Ship** class, are placed.
The Ship class has its name, health (corresponding to its length) and a *Set* containing the coordinates of the ship itself, as fields. It has a private constructor: it is not possible to create a generic ship. To make one, there are some static methods which return the predefined types of ship. There are also some methods to hit a ship and know if it is sunk. Once, sunk, the *sunkDialog* which highlights the ship with red, is called and the player informed that it has been sunk. The CellPanel class extends from *JPanel* and it creates every cell on the game map. Its dimension depends on the size of the grid. Every cell is shows with a sea background and a black *Lineborder*. Every panel has the xy coordinates. It also has a field to memorize if a ship has been placed on and which ship it is. There are some methods to update the state of the panel when hit too. Ships are selected by clicking on the respective images which is actually a button, **JRadioShip**, extending from *JRadioButton*. They're rotated using the keyboard because of the **ChangePositionListener** implementing a *KeyListener* and placed using the **PlaceListener** implementing *MouseListener*. The **HitListener** is used to click cells on the map. When the game is finished the summary page pops with all the statistics of the player.

4.2 Division of work: Nicola di Berardino

Diagram related to the player (packages: **game.player**, **game.ai**, **game**):



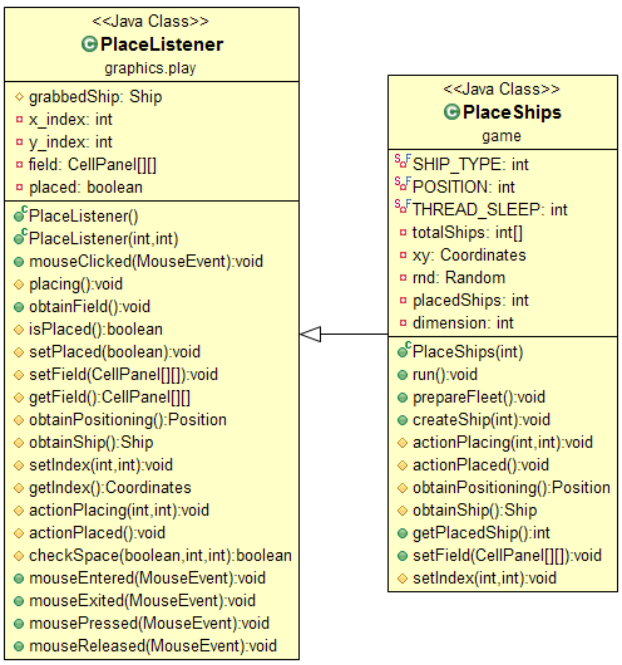
- **AbstractInfo** is an abstract class which contains all the skeleton methods that are used by its all sub-classes.
- **InfoP** checks all the actions users can perform in single-player such as hitting or using special actions. Additionally, checks if the user wins and updates his/her score.

- **InfoPVersus** checks the actions players can perform in multi-player games.
- **InfoAI** checks the actions that the AI can perform. Its difficulty is set using the *aiStrategy* object with the proper class.
- **AI** is an interface.
- **AbstractAI** is an abstract class which holds all the common methods for the AI difficulties in single-player and has already everything needed to use the Easy difficulty. The *Strategy pattern* has been used.
- **EasyAI** is the easy difficulty. It hits randomly. The equation 1 shows what ideally would be the probability for this AI to win the game without failing.
- **NormalAI** is the normal difficulty. When it hits a ship, it tries to determine its direction by hitting without a precise order the squares next to that shot. When found, it hits all the squares in that row until finds water. Then it does the same on the opposite direction, whether or not it has sunk the ship. When done, is starts hitting randomly like **EasyAI**.
- **Hit** is an enumerated type used to classify the attack result: *WATER* if missed, *HIT* if hit and *SUNK* if hit and sunk.

$$\prod_{i=1}^{n} \frac{n-i+1}{m-i+1} \tag{1}$$

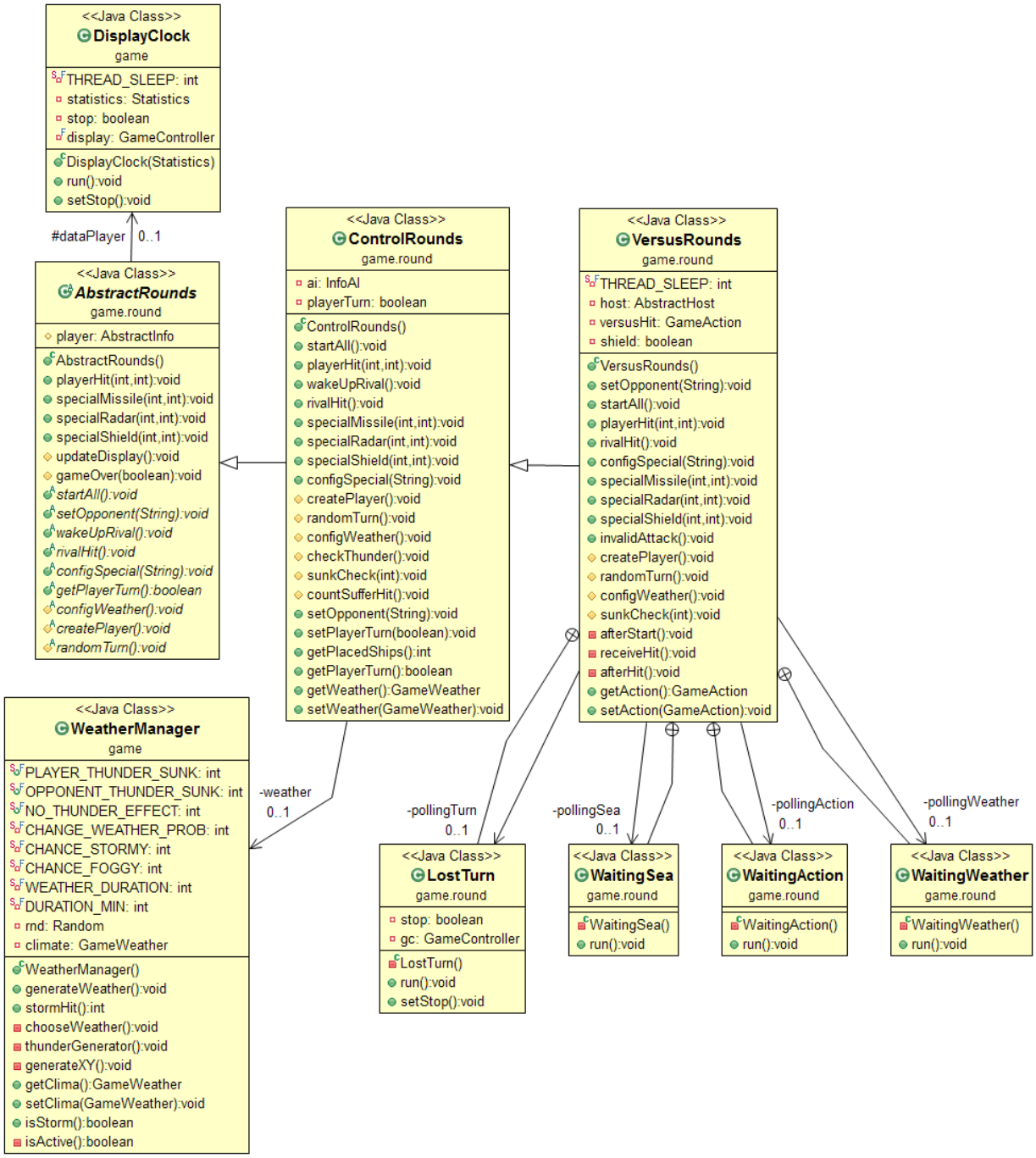
n : number of squares occupied by ships
m : number of total squares

Diagram related to the AI ship placing (package: **game**):



- **PlaceShips** is used to place ships by the AI. It implements *Runnable* and extends form the *PlaceListener* class from **graphics.play** package. It uses the size of the game map to determine the number of ships and the position for each one is randomly generated. Obviously, ships cannot overlap

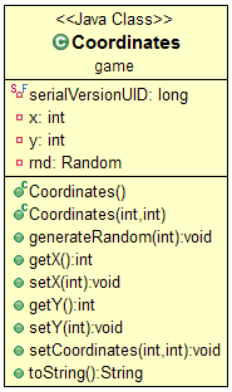
Diagram related to the turns management (packages: **game.round**, **game**):



- **AbstractRounds** contains all the common aspects for the two sub-classes such as calls to hit a square or special attacks. There are also some methods to update the game screen with statistics and to end the game.
- **ControlRounds** handles the order in which the various actions are executed in a game against the AI. Players attacks are based on the **InfoP** class, while IA ones on **InfoAI**. It also randomly generates who's starting first and calls the creation of the weather every turn.

- **VersusRounds** manages the turn order in multi-player games which actions are based on the **InfoPVersus** class, but asynchronously, when someone attacks the other receives the attack and vice versa. It also handles polling through some nested classes that extend *Thread*, used to wait for user actions or any other information exchange between the two hosts. The server class, though, is the only one that generates weather at the end of its turn and sends it to the Client as the weather of the next turn.
- **WeatherManager** generates the weather conditions used in game. Sun (odds 8/10) has no effects, Fog (odds 1/10) temporarily hides the map, ships which have been already sunk are still visible though, and Storm (odds 1/10) generates one lightning per turn in a random field. If it hits water, then nothing happens, but if it hits a ship the square is discovered and marked as hit. Special weather conditions (Fog and Storm) have a random duration, from 2 to 5 turns.
- **DisplayClock** is a *Thread* used to update the clock time during games.
- **WaitingSea** is the first polling, used to wait the map of the opponent.
- **WaitingAction** used to wait the players actions.
- **WaitingWeather** is the polling used by the Client to wait weather. When he receives it, his/her turn can start.
- **LostTurn** uses the timer from **multiplayer.time** package to determine if the player loses his/her turn, in which case a fake attack message with invalid coordinates is send to the other player to signal the turn has ended.

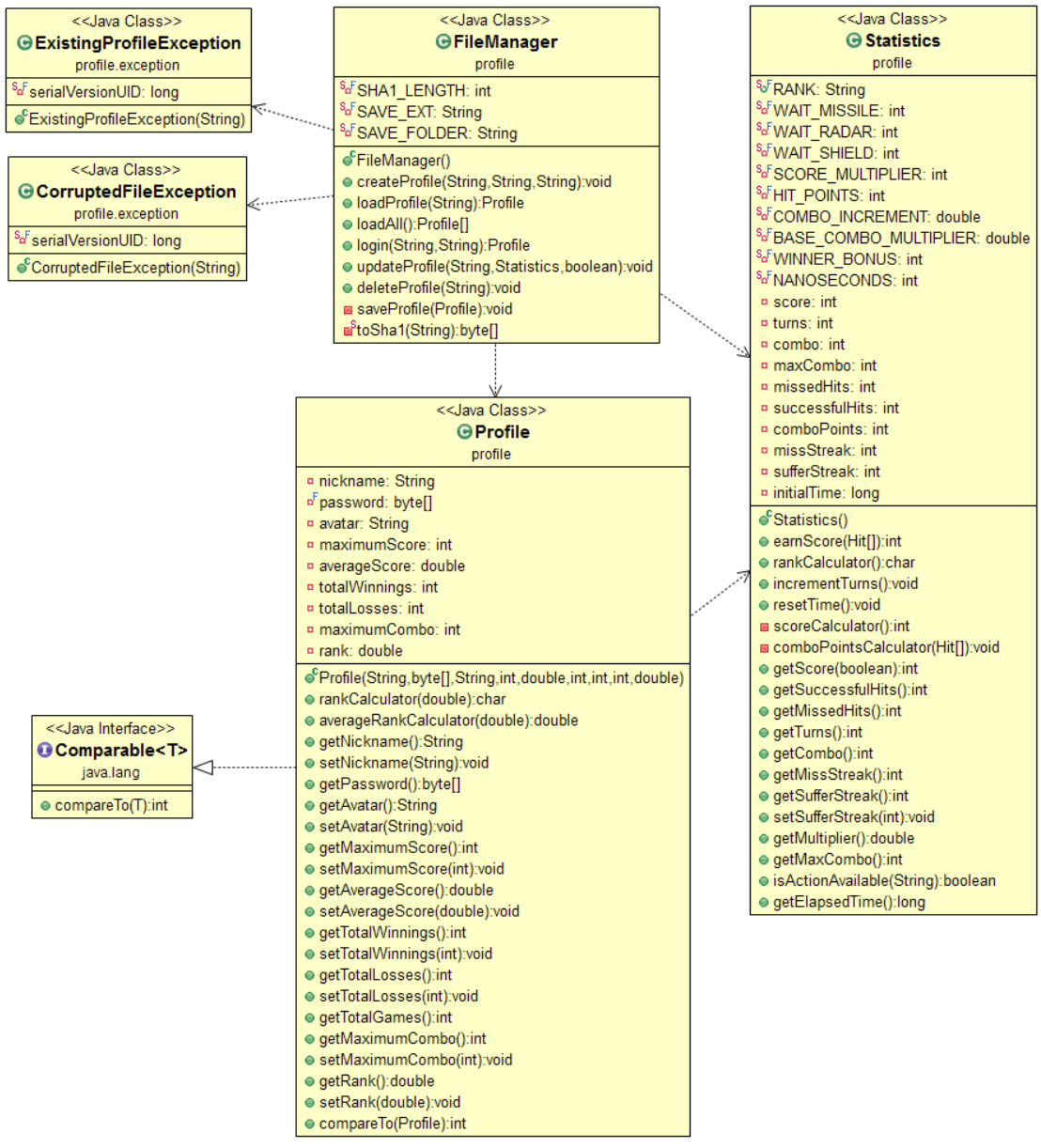
Diagram related to the coordinates management (package: **game**):



- **Coordinates** is used to hold coordinates which can be also randomly generated.

4.3 Division of work: Michel Paoloni

Diagram related to the profile management (packages: **profile**, **profile.exception**):



- **Statistics** holds all the data relative to game-play. Some of this information is shown in the game window, some in the summary page at the end of the game. The main score is based on the ratio between *hits* and *failures* but, combos, that are the number of hits in a row, also affect score: the more the hit-streak goes, the more the score will grow. The ranking system, however, is solely based on the ratio between the two. Equation 2 shows the correct formula for the score calculation.

- **FileManager** is used to save, loads and update profiles after a new game has been played. Every profile is saved in one single file with extension `.srsav` allowing users

to easily export their own profiles. The password is salted with the player's nickname and hashed using the well-known SHA-1 algorithm with one iteration. Although passwords are salted and hashed per user, this mechanism does not provide any additional security. The safety of the account is demanded to the user who must set a strong password.

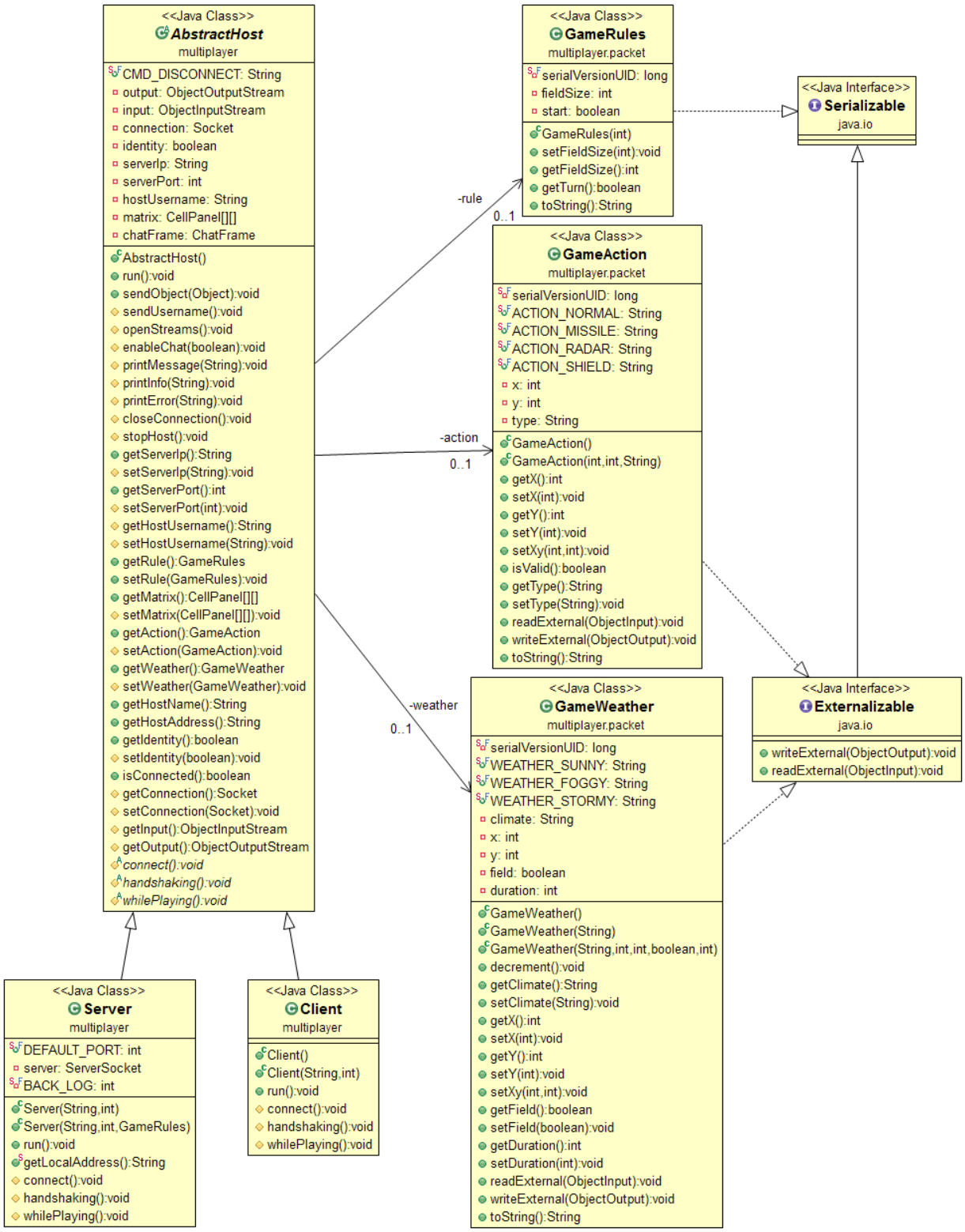
- **Profile** is used mainly as a temporary holder to store all the information relatively to a player's profile. It implements the *Comparable* Java interface because of the main window ranking feature.

- **ExistingProfileException** and **CorruptedFileException** are a couple of new Exceptions which have been defined. The former is used when someone tries to create a profile with the same nickname of another player, the latter when the name of the saving file does not match the player nickname in the file itself (i.e. if someone renames it).

$$\text{Score} = \frac{\text{hits}}{\text{misses}} k + h \sum_{j=1}^{n} \sum_{i=1}^{r} ((1 - m) + m \cdot i) \tag{2}$$

k : constant
h : hit points
n : number of hit-streaks
r : hits in a row
m : multiplier increment

Diagram related to the profile management (packages: **multiplayer**, **multiplayer.packet**):



- **AbstractHost** extends from the *Thread* Java class and holds all the common aspects of the two sub-classes.
- **Server** is used to host games. It is similar to the **Client** class but there are a few differences: the player chooses the rules for the game and the server is also responsible to generates the weather conditions. The method *getLocalAddress*[1] is used to get a proper local address where to start the server.
- **Client** is used to connect to a server.
- **GameRules** is used to send rules. It implements the *Serializable* Java interface.
- **GameAction** is used to game actions. It implements the *Externalizable* Java interface.
- **GameWeather** is used to send weather conditions. It implements the *Externalizable* Java interface.

AbstractHost is an abstract class from which **Server** and **Client** inherit. Since they are both used for playing, they have a lot of aspects in common.
There are three main common stages for which it has been used the *Template method*: *connect*, *handshaking*, *whilePlaying*. The first one is the connection phase: when the server has been started it waits until someone connects. On the other hand when the Client starts, it attempts to connect to the Server (which has to be already created). When both hosts are connected the handshaking part can take place. During this stage, information needed to start the game is exchanged between the two that is, usernames, rules for the game and respective player's maps. After the rules have been sent from the server to the client, the chat is enabled and players can start placing ships on their grid. When a client is ready, he can press the button to start and his map will be sent to the opponent. When a player receive the opponent's map, his handshaking part ends and the whilePlaying stage can take place. Despite this, the game will start only when the two player have received their opponent's maps.

In order to send messages from an host to the other, the Java object serialization has been used, so some classes such as **GameRules**, **CellPanel** and **Ship** (the last two for the maps exchange), implement the *Serializable* interface. There are some performance problems though: it depends on reflection, it has a verbose data format. It is, in general, both slow and band-width intensive. To overcome these problems, for messages which are frequently exchanged the *Externalizable* interface has been used instead.

[1]It was found on the Internet and then slightly modified.

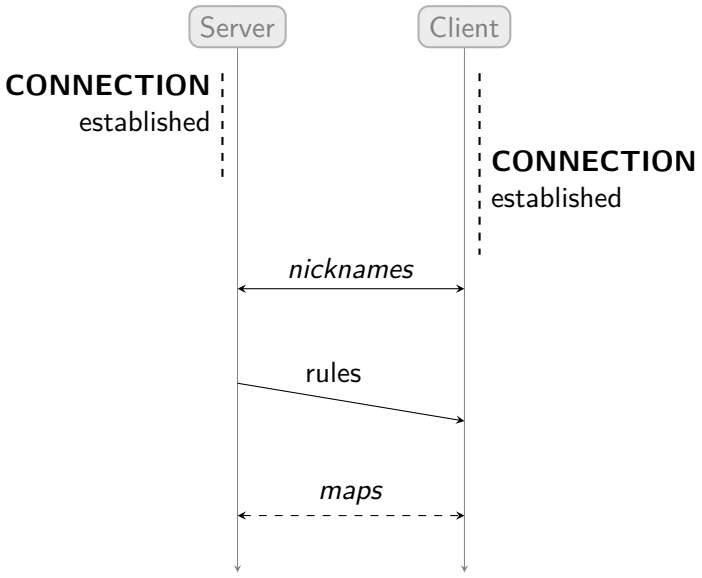
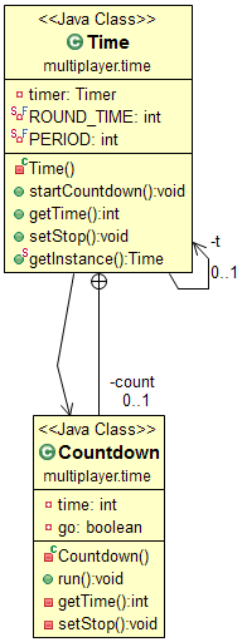


Figure 1: *Handshaking* phase.

Diagram related to the turn time management (package: **multiplayer.time**):



- **Time** is used to start a countdown timer in multi-player mode. Every user has a certain amount of time to make a decision: when it runs out the player's turn ends and the control change to the other player. In order to do so and keep both games synchronized, when there is no more time available the game sends to the other host a fake attack message with invalid coordinates.
  This class uses the *Singleton pattern*.

- **Countdown** extends from the *TimerTask* Java class and every time the method to

start the countdown is called, a new task is scheduled, that is, the turn timer.

5 Testing

All the testing was done manually, trying to go through all the possible cases and scenarios. Particular attention was paid at both single and multi-player modes.

6 Notes

The **Server** and **Client** classes which were independently coded were re-factorized as sub-classes of **AbstractHost**. At first, before using object serialization to send information between players, only string messages were planned. Then, regular expressions would have been used to evaluate every message.

The multi-player protocol designed is very simple and it might fails on very overcrowded and slow networks.

At first the coding process was done independently, but from at certain point on, interactions have been pretty frequently. One member joined the team late, so the others had to wait some things were completed.

Anyway, the whole team is pretty happy with the final result which fulfilled the great majority of their expectations.