

Battaglia Navale

Dubkov A.

August 2021

1 Introduction

This document describes functionality of project **Battlefield**(it.- "*Battaglia Navale*") developed by dubkov. First of all project had been implemented using concurrent module of JDK(Java development kit). Also some interesting features will be further described later on in the document. It doesn't contain any external library besides JUnit 5(used for test implementation, for details -Link JUnit). Code had been written in the way to give a possibility to apply some new features for the final Client.

2 Description packages

2.1 Introduction

This section contains information regarding all packages in the project **Battlefield**. Each project's name starts with following signature- *com.battlefield.{Canonical name of package}*. Total number of packages equals to 10.

2.2 com.battlefield.game

This is an entry point of whole project. There are three files: *Main.class*(class represents start point), *Game.class*(singleton represents technical setup), *ThreadStopGame.class*(thread created with task to stop project execution when there is escalated some event).

2.3 com.battlefield.menu

Current package shows 30 implemented menu classes. Each of them implements interface **Menu**(*polymorphism*), we'll speak about them later in detail. Because of Single Responsibility Principle here we can't find **Printer.java** singleton which is located in **com.battlefield.printer**.

All menus are **Threads** because of keeping away from **StackOverflowError**. For example, if program developed just with functions(for menu implementation) and it(program) has three menus with following switches: 1)Option to go back from third to the first, 2)Options to go from the first to the second

one and go forward from the second to the third; the problem is if user after going from the first to the third menu goes to the first one, would execute call for the first function which had been called before, these function's calls fill the stack in Java Virtual Machine. So there are possible solutions for this problem: 1)Implement **Observable Design Pattern**, there are two main disadvantages-difficult to manage, requests more space in memory, 2)Utilize **thread**- is more efficient than with functional programming.

2.4 com.battlefield.player

Package is represented by classes-models on the highest level of game. There are three main classes:**Human.class** (designed for player creation);**Computer.class** (for computer creation); **PlayerRepository.class** (Repository design pattern to save players inside).

2.5 com.battlefield.printer

Package contains just one singleton- Printer. Mentioned class has different methods(or functions) which help to print strings regarding context.

2.6 com.battlefield.settings

Package contains initial setup for a project and all strings, needed for output in console, if these strings will be translated game would work anyway.

2.7 com.battlefield.ships

Bundle of classes made for manipulation all possible ships. Also, as **com.battlefield.player**, contains repository design pattern-**ShipRepository.class**. **ShipRepository.class** is created for ship Manipulation

2.8 com.battlefield.userinputs

Two classes are included in this project, created for input manipulation, contains *UserInputCheck.class, UserInputHandler.class*. *The first one is created for input validation, the second one for handling*

2.9 com.battlefield.security

*Package contains just one class ExtendedSecurityManager which needed to implement access by reflection just for **Junit package**. So it's impossible to have access to private methods, variables*

2.10 com.battlefield.tests

Unique package which utilizes features of external library(JUnit). All classes designed in this package gives a possibility to test application. User friendly interface(in this case means console) shows all test(passed and not) and highlights with corresponding color: red(for not passed tests) and green(for passed), there are many advantages to write tests in this way about them you can read in the next section.

3 Description classes

3.1 com.battlefield.game

3.1.1 Main.java

Class contains just main thread where application is initialized.

3.1.2 Game.java

Enum, where can be found *ExecutorService*(used to manage threads), *PlayerRepository*(design pattern created for manage players), *TurnGame*(for turn management of players)

3.1.3 ThreadStopGame.java

Thread for single game execution. Must be stopped when all active users equal to 1(remains as winner).

3.2 com.battlefield.menu

Has 30 different classes presented as menu(each menu implements interface *Menu*), achieve concept of polymorphism(switch between menus), each menu has three methods overridden thanks to implemented interface *Menu*: **label()**- returns name of current Menu, **launch()**- launches Menu, **run()**- from *Runnable* interface(Interface segregation principle- dividing interface in small parts- *Menu* interface extends *Runnable*), **show()**- prints in console menu's components. Bundle of menus is a kind of *LinkedList*(one directional), but with possibility in some Menus to go back(and go to ancestors). Some menus have parent- gives possibility to go back, or execute operation and go back. Each thread launches corresponding menu.

3.2.1 AddBattleshipMenu.java, AddCarrierMenu.java, AddDestroyerMenu.java, AddPatrolBoatMenu.java, AddSubmarineMenu.java

AddBattleshipMenu.java is identical as *AddCarrierMenu.java*, *AddDestroyerMenu.java*, *AddPatrolBoatMenu.java*, *AddSubmarineMenu.java*. The main difference is in addition of ship's type(Carrier, Battleship, Submarine,

*Destroyer, PatrolBoat). After addition it goes to object called "parentMenu"- where objects of mentioned classes had been created(here it might be **AddShipByTypeMenu.java**).*

3.2.2 AddShipMenu.java

*. This class helps switch to following menus: **AddShipByTypeMenu.java**, **AddShipByCellsMenu.java**. Also there are switch back(goes to "parentMenu") with menu which turns off game.*

3.2.3 AddShipByTypeMenu.java

*As it mentioned before this is predecessor menu of **AddBattleshipMenu.java**, **AddCarrierMenu.java**, **AddDestroyerMenu.java**, **AddPatrolBoatMenu.java**, **AddSubmarineMenu.java**.*

3.2.4 AddShipByCellMenu.java

This menu assists for addition different ships dynamically. Player can create ship just with two cells: initial and final.

3.2.5 CreateComputerMenu.java

Current class is created for computers' creation, there are controls(there is no option to add more than one computer inside of PlayerRepository).

3.2.6 CreateHumanMenu.java

Creates Human with name choosen by user. It is impossible to add users with the same username, because of identification's problem.

3.2.7 DowngradeShipMenu.java

Downgrades ship from upper level(with more cells) to lower one(less cells)- difference equals to one cell.

3.2.8 MainMenu.java

Entry menu of program. Offers 3 choices: 1) Play game(if there is sufficient quantity of players in PlayerRepository); 2) Manage player to Repository; 3) Quit game(this one can be found in all menus).

3.2.9 ManageUsersMenu.java

This one is used for basic operations with user: create, delete human/computer; go back and turn off.

3.2.10 MenuDuringBattle.java

Class had been created for game management, if there is no stop Class will execute all turns sequentially after attack's execution. Each attack can cause ships' cancellation or player deletion from game(but not from Repository!).

3.2.11 MenuHelper.java

*Contains methods for menu display(with back option, without back option, takes in input two cells of ships etc). **MenuHelper.java** is developed with scope to help all menus.*

3.2.12 PostBattleMenu.java

*When one user remains with boolean variable **loose** equal to false, from **MenuDuringBattle.java** game passes to **PostBattleMenu.java**. From current one it is possible to go into **MainMenu.java** or turn off game.*

3.2.13 PrepareForBattlePlayerMenu.java

*Class created for Ship Repository manipulation, there are available following operations: addition, upgrade,downgrade, deletion, print all available ship, and change turn. When all users are ready(had been allocated all available ships), game will be launched with two threads(one for the next Menu, another one-for **ThreadStopGame.java**)*

3.2.14 QuantityUsersMenu.java

Shows quantity of players and their names,after goes back into parent menu, passed as parameter in constructor.

3.2.15 RemoveAllShipsFromDefenceBoardMenu.java

Removes all ships from Defence Board of human player. Practically, player can utilize this feature for ships' reallocation.

3.2.16 RemoveComputerMenu.java

*This one was built for computer deletion from players' repository. Decreases static counter inside of **Computer.java**. After execution goes back to parent Menu.*

3.2.17 RemoveShipByCellMenu.java

Removes Ship which contains selected cell from ships' repository(belongs to player), as from defence board.

3.2.18 RemoveShipMenu.java

*Intermediate step between **PrepareForBattleMenu.java** and **RemoveShip-ByCellMenu.java**, **RemoveAllShipsFromDefenceBoardMenu.java***

3.2.19 RemoveUserMenu.java

Removes user from repository, after input of username(identificator).

3.2.20 ShowAvailableShipsMenu.java

Prints all available ships- their quantity, space occupied by them.

3.2.21 ShowBoardForAttackMenu.java

*Shows Board for Attack when Player assaults in **MenuDuringBattle.java***

3.2.22 ShowDefenceBoardMenu.java

Shows Defence Board with all enemies' shots.

3.2.23 UpgradeShipMenu.java

Menu had been created for ship Upgrade, hence it is possible to maximize ship.

3.3 com.battlefield.player

3.3.1 Player.java

Abstract class which helps to implement polymorphism. This class must be extended to create two different types of players: computer, human.

Computer.java has methods for generation of defence board

3.3.2 Human.java

*Extends **Player.java** and permits to manage human player, particularly, operations: create player with certain name, upgrade ship corresponding to player, downgrade ship corresponding to player, remove ship corresponding to player.*

3.3.3 Computer.java

*Extends **Player.java**, has methods to generate ships on defence board in random way.*

3.3.4 PlayerRepository.java

Repository design pattern(link to design pattern- analog in Spring) developed for user manipulation. Gives possibility to add, remove users(humans and computers also), core is a Singleton because of taking away the prospect to create two different repositories.

3.3.5 PlayerStorage.java

Singleton which is created for users' addition and deletion, generally manipulations with list.

3.3.6 Turn.java

*Class developed for manage Turns committed by players, if one person made his turn and after there is a computer **Turn.java** manages also that one(all computers' turns, if there are more than one in PlayerRepository)*

3.4 com.battlefield.boards

3.4.1 Cell.java

This is the smallest unit of board, in this class there are four overridden methods of Object class: `compareTo`(needed for `TreeSet`), `equals`(for `HashMap`), `hashCode`(for `HashMap`), `toString`(for better representation).

3.4.2 Board.java

*An abstract class which initializes objects related to concreted player, has method `showBoard`- shows representation of ships on map. Core of class is `HashMap`, permits to return values associated with cell in constant time($O(1)$). To return values associated with cell there had been overridden `hashCode` method of **Cell.java**.*

3.4.3 BoardForAttack.java

*Class contains just one method **attack(parameters....)**, assaults whole `PlayerRepository` besides attacking player. Method **toString()** represents an object.*

3.4.4 BoardForDefence.java

*Extends **Board.java**. Incorporates methods for board manipulation, especially it is possible to generate ship with two cells(initial cell and last cell), also there are methods dedicated to upgrade ship, downgrade ship and remove ship. These operations are possible thanks to assumptions:1) Cell on the left is less than cell on the right of board(**compareTo** method);2) Cell on the top is less than cell on the bottom of board(**compareTo** method).*

3.4.5 BoardHelper.java

This class contains auxiliary methods for generation ships from initial cell and last cell(horizontal or vertical ship). Identifies if cells are on the same line, if there are touches with another ships already generated on board.

3.4.6 BoardValidator.java

Current class contains methods for different checks on the board, for generation ships if there are touches between generated ship and other ships on a board.

3.5 com.batterfield.printer

3.5.1 Printer.java

***Printer.java** had been created to show output in console, methods are divided by parameter passed.*

3.6 com.batterfield.security

3.6.1 ExtendedSecurityManager.java

*This class must to be included to avoid reflection from any class(gives possibility to access any private method/class). Just one package has permission to use reflection-**Junit**(utilized for test).*

3.7 com.battlefield.settings

3.7.1 Status.java

*Aim of this class is to create statuses for sending errors and successful execution by programmer. In some classes there are exceptions, but for better performance there were utilized statuses, all messages of **Status.java** had been defined in **Words.java**. Usually status will define behaviour, controls if methods had been executed with success. Works as protocol.*

3.7.2 Words.java

***Words.java** represents all messages for output in console written in English, as well as board's letters and numbers. If all messages will be translated in other language game would work anyway.*

3.8 com.battlefield.ships

3.8.1 Ship.java

Abstract class represents ship- core of each ship. Two main components of ship are type and occupied cells.

3.8.2 Carrier.java

***Battleship.java**(Portaaereo-it.) is a type of ship in Battlefield. Represented by five cells. Relative quantity and size can be found in **Words.java**.*

3.8.3 Battleship.java

Battleship.java(Corazzata-it.) is a type of ship in Battlefield. Represented by four cells. Relative quantity and size can be found in **Words.java**.

3.8.4 Destroyer.java

Destroyer.java(Crociera-it.) is a type of ship in Battlefield. Represented by three cells. Relative quantity and size can be found in **Words.java**.

3.8.5 Submarine.java

Submarine.java(Sottomarino-it.) is a type of ship in Battlefield. Represented by three cells. Relative quantity and size can be found in **Words.java**.

3.8.6 PatrolBoat.java

PatrolBoat.java(Nave Assalto-it.) is the smallest(occupies just 2 cells) ship in Battlefield. There is no chance to create ship with different quantity of space. Represented by two cells. Relative quantity and size can be found in **Words.java**.

3.8.7 RepositoryShips.java

Interface- needs to be implemented in **ShipRepository.java**.

3.8.8 ShipStorage.java

ShipStorage.java- class created for ships manipulation inside of HashMap(where keys are types of ships and values - ships).

3.8.9 ShipRepository.java

ShipRepository.java represents Repository Design Pattern mentioned above(see **PlayerRepository.java**). Major differences between this class and **PlayerRepository.java** include: core of **ShipRepository.java** is not a Singleton(it isn't an obstacle to create different ship's repositories for different players).

3.8.10 ShipCheck.java

Class created for ship's identification depending on cells' quantity.

3.9 com.battlefield.tests

3.9.1 Description of advantages

Tests, implemented in this manner, help to analyze behaviour of all classes in simpler way. It is possible to generate different inputs and pass it into test methods, also is simpler to define errors inside of test methods or methods,

we are testing on. Gives possibility to test if certain error will be caught(not depending on text in console).

3.9.2 Description features with examples

In **BattleshipTest.java** there are two methods which test if **IllegalArgumentException** will be thrown, **@DisplayName** changes the name displayed in console.

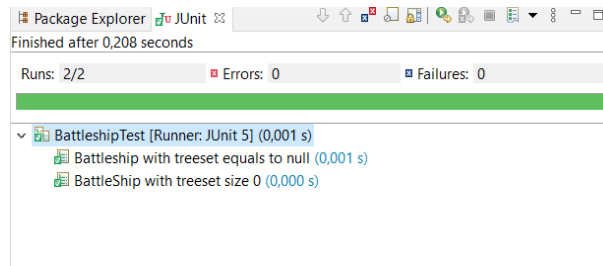


Figure 1: Junit Output for BattleshipTest.java

Secondly, **JUnit classes** can utilize multiprogramming concept, precisely, in class **UserInputHandler.java** there are four inner classes created with scope to use multiprogramming. Output you can see below.

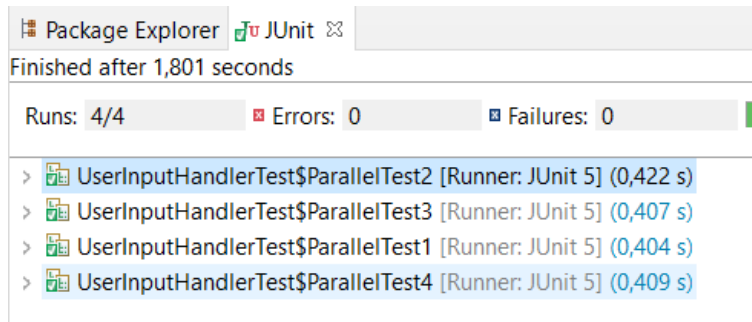


Figure 2: Junit Output for UserInputHandlerTest.java

Technically this is a pseudoparallelism, makes impression that all classes are executed in parallel way, but they aren't. Immediately after execution of one class **JUnit** starts execution of another. Also these classes test values taken in input thanks to **ByteArrayInputStream.java** and library **System.java**.

In this framework there are a lifecycle hooks, and in some classes there is an initial setup made with annotation **@BeforeEach**, **@BeforeAll**.

As it said before there is a possibility to verify different inputs dynamically, it can be reached with **@MethodSource**, as it had been done in **BoardValidatorTest.java**.

3.10 com.battlefield.userInput

3.10.1 UserInputHandler.java

Current class contains methods for taking input from user and thanks to this class user can send input from keyboard(utilizes **java.util.Scanner**).

3.10.2 UserInputCheck.java

Used for control inputs in **UserInputHandler.java**, if inserted number is in range of quantity of submenus(it might be menu with back option, without back option)

3.11 com.battlefield.security

3.11.1 ExtendedSecurityManager.java

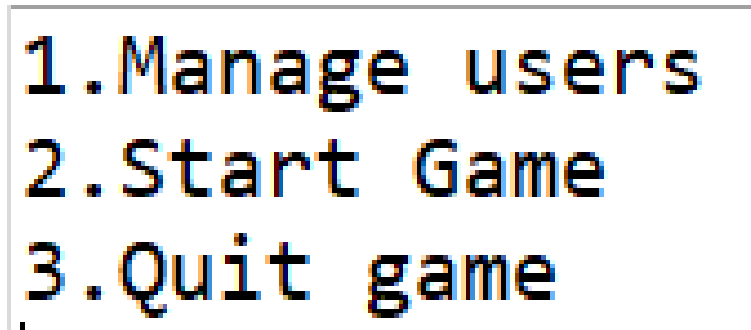
Includes two methods **isJUnitTest()**- controls if it was a call from package which is Junit or not, **checkPermission()**-throws Exception if there is an access from package which is not Junit.

ExtendedSecurityManager.java was made for avoiding Reflection written by other programmers on one of this classes.

4 Description of functionality

4.1 PlayerRepository operations

Program starts from **MainMenu.java**



```
1.Manage users
2.Start Game
3.Quit game
```

Figure 3: Console output MainMenu.java

After selection of the first submenu, game shows following Menu:

"Create user"- means creation of human player, hence **"Create computer"**- creation of computer player, output of these operations you can see below:

Human player had been created because of absence player with the same username.

```
1.Create user
2.Remove user
3.Create computer
4.Remove computer
5.Show all participants
6.Go back
7.Quit game
```

Figure 4: Console output ManageUsersMenu.java

```
1
Insert username
dubkov
dubkov had been created
```

Figure 5: Console output CreateHumanMenu.java

```
3
|COMPUTER1 had been created
```

Figure 6: Console output CreateComputerMenu.java

When computers' addition happens- static variable in **Computer.java** will be incremented by 1 unit- this give possibility to differ computers.

```
4
If there is any computer it will be cancelled
COMPUTER1 had been removed
```

Figure 7: Console output RemoveComputerMenu.java

Above you can observe operations of computer's deletion.
Also user's deletion, if username is presented in *PlayerRepository*:

```
2
Insert username
dubkov
dubkov had been removed
```

Figure 8: Console output RemoveHumanMenu.java

QuantityUsersMenu.java displays all users in *PlayerRepository*:

```
5
Total quantity: :2
dubkov:Human
COMPUTER1:Computer
```

Figure 9: Console output QuantityUsersMenu.java

When all players had been added, we can start a game. Sequence of users' turns corresponds to sequence of user's addition in *PlayerRepository*.

4.2 ShipRepository operations

On figure below you can see all options, available to human player:

```
dubkov 's turn
1.Add Ship
2.Remove Ship
3.Upgrade Ship
4.Downgrade Ship
5.Show available ships
6.Show defence board
7.Next turn/Start Battle
8.Quit game
```

Figure 10: Console output PreparationForBattleMenu.java

*AddShipMenu.java is intermediate menu between ships' addition and **PreparationForBattleMenu.java**:*

```
1
1.Add ship selecting type
2.Add ship inserting two cells
3.Go back
4.Quit game
```

Figure 11: Console output AddShipMenu.java

This menu shows two options for ship addition: 1) selecting type; 2) inserting two cells.

Technically, the second way generates ship depending on cell quantity, adds to player's ShipRepository.

Output of Patrol Boat addition is below:

```
2
Input the first cell in following format:'Letter Number'
a 1
Input the second cell in following format:'Letter Number'
a 2
Patrol boat was added to defence board
```

Figure 12: Console output AddShipByCellMenu.java

Addition ship by selection type before is almost identical but there is a control if ship contains necessary quantity of cells.

```
2
Input the first cell in following format:'Letter Number'
a 1
Input the second cell in following format:'Letter Number'
a 2
Patrol boat was added to defence board
```

Figure 13: Console output AddShipByTypeMenu.java

RemoveShipMenu.java shows 2 possible options, besides quit and go back:

```
2
1.Remove ship choosing cell
2.Remove all ships from defence board
3.Go back
4.Quit game
```

Figure 14: Console output RemoveShipMenu.java

If player knows cell which is part of ship, this ship will be removed from Ship Repository of Player. The second one is for all ships' deletion from Defence Board.

Below you can observe deletion of Patrol Boat.

```

1
Input cell in following format:'Letter Number'
a 1
Operation had been executed with success

```

Figure 15: Console output RemoveShipByCellMenu.java

Following operation removes all ships from Defence Board:

```

Operation had been executed with success
1.Remove ship choosing cell
2.Remove all ships from defence board
3.Go back
4.Quit game

2
Operation had been executed with success

```

Figure 16: Console output RemoveAllShipsFromDefenceMenu.java

The most difficult and interesting operations on Defence Board are upgrade and downgrade. It gives possibility to change quantity of cells which belong to ship with type modification. Below there is shown ship's upgrade from Patrol Boat to Submarine

```

3
Input cell in following format:'Letter Number'
a 3
Ship before Patrol boat
Ship after Submarine
Operation had been executed with success

```

Figure 17: Console output UpgradeShipMenu.java

Inverse option of upgrade is downgrade, result:

```
4
Input cell in following format:'Letter Number'

a 1
Ship before Submarine
Ship after Patrol boat
Operation had been executed with success
```

Figure 18: Console output DowngradeShipMenu.java

*The fifth option of **PreparationForBattleMenu.java** is **Show available ships**. This one prints out all possible types of ships(their names), quantity on the left and size on the right side.*

```
5
|
AVAILABLE SHIPS ARE
2 Patrol boat SIZE:2
2 Destroyer SIZE:3
3 Submarine SIZE:3
1 Carrier SIZE:5
1 Battleship SIZE:4
----
```

Figure 19: Console output ShowAvailableShipsMenu.java

The sixth Menu of **PreparationForBattleMenu.java** is **Show defence board**. Below there is a representation of defence board after Patrol Boat's addition.

```

6
|   | A | B | C | D | E | F | G | H | I | J |
-----
1 |   |   |   |   |   |   |   |   |   |   |
-----
2 | + |   |   |   |   |   |   |   |   |   |
-----
3 | + |   |   |   |   |   |   |   |   |   |
-----
4 |   |   |   |   |   |   |   |   |   |   |
-----
5 |   |   |   |   |   |   |   |   |   |   |
-----
6 |   |   |   |   |   |   |   |   |   |   |
-----
7 |   |   |   |   |   |   |   |   |   |   |
-----
8 |   |   |   |   |   |   |   |   |   |   |
-----
9 |   |   |   |   |   |   |   |   |   |   |
-----
10 |   |   |   |   |   |   |   |   |   |   |
-----

```

Figure 20: Console output ShowDefenceBoardMenu.java

After all ship addition game must be launched. But if Ship Repository is not full, program would return following message:

```

7
Add all available ships to a board

```

Figure 21: Console output ShowDefenceBoardMenu.java

4.3 Game

DuringBattleMenu.java is menu which shows menu during battle, precisely, shows two options- attack, quit.

6											
7											
8											
9											
10											

		A		B		C		D		E		F		G		H		I		J	
1		+				+				+				+				+			
2		+				+				+				+				+			
3		+				+				+				+				+			
4		+				+															
5		+																			
6																					
7		+				+				+											
8		+				+				+											
9		+				+															
10										+		+						+		+	


```

1.Choose cell to attack
2.Quit game

```

Figure 22: Console output DuringBattleMenu.java

Two boards before menu are Board for Attack, Board for Defence. Game will stop, when in PlayerRepository will remain just one user with boolean lose equal false(Winner).

After game finishes Menu switches to **PostBattleMenu.java**

```

h 1
|COMPUTER1 lost
|dubkov won
|Post Battle Menu
|1.Main Menu
|2.Quit game

```

Figure 23: Console output PostBattleMenu.java

There are two options- go back to Main Menu, or quit game. When it will go back to Main Menu program removes all users from PlayerRepository.

5 SOLID

5.1 Single Responsibility

*Each package has own scope: **com.battlefield.printer** contains enum **Printer.java** which can be used for output in console. **com.battlefield.ships** differs types of ships, works with Ship Repository. Other packages are responsible for manipulations with Player Repository, Board creation etc.*

*Each class has unique scope, such as with methods of **ExtendedSecurityManager.java** there is impossible to manipulate instance of **Human.java**.*

5.2 Open-Closed Principle

Generally it is not a good idea to edit classes, because of support's problem.

*Class **Computer.java** can be extended with another classes to implement different Strategies- open for extension, and modification must be avoided.*

5.3 Liskov substitution principle

*Three superclasses where it is possible to find Liskov substitution principle - **SecurityManager.java**, **Player.java**, **Ship.java**- methods of these classes can be implemented in subclasses and doesn't violate this principle.*

5.4 Dependency inversion

Menu.java helps to utilize this principle. All Classes, which implement *Menu.java*, depend on abstractions not on concrete classes, it gives possibility to switch between menus, and not depending on class.

6 UML Diagrams

6.1 com.battlefield.boards

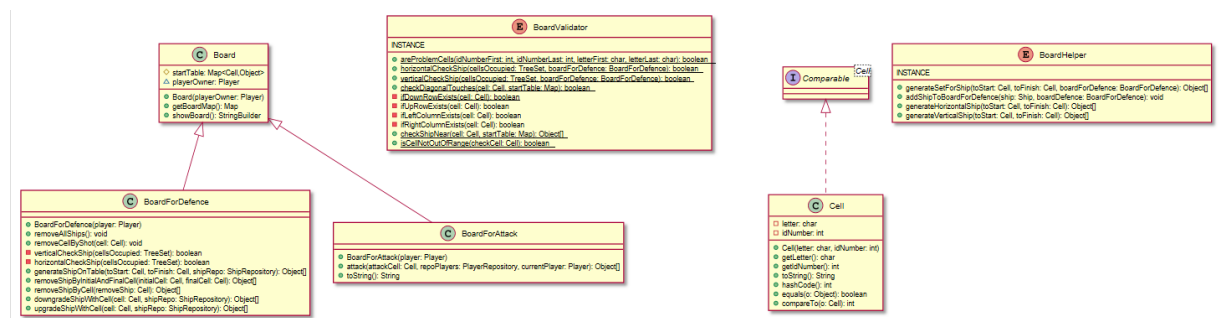


Figure 24: UML-diagram com.batterfield.boards

6.2 com.battlefield.game

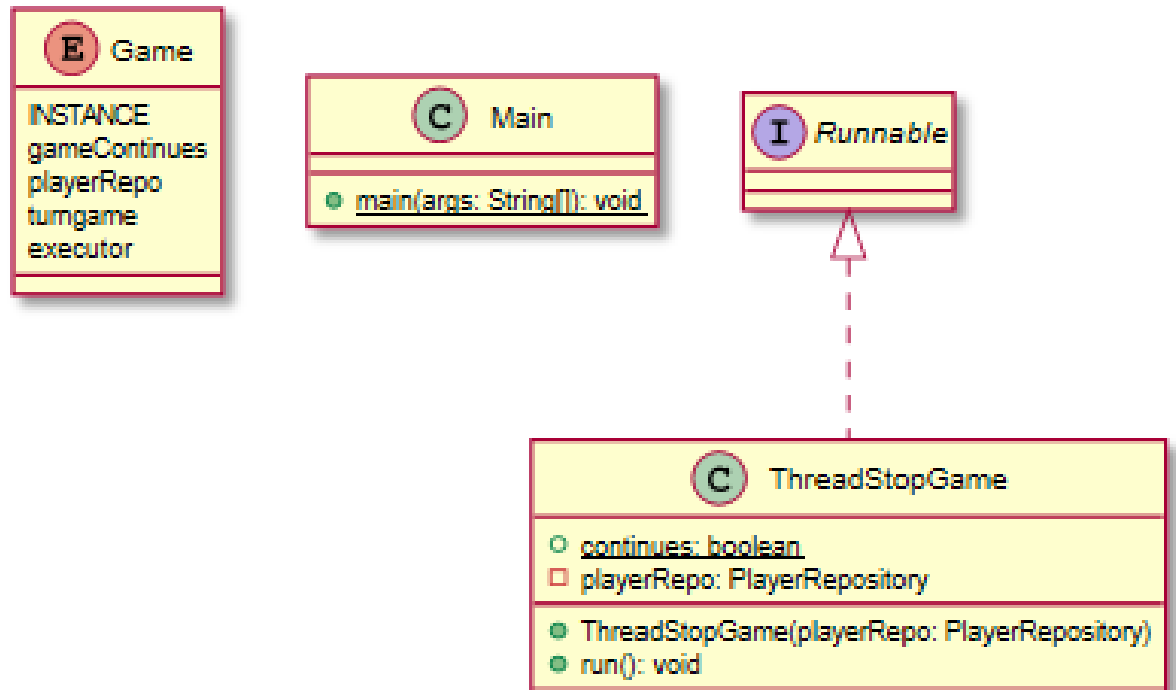


Figure 25: UML-diagram com.battlefield.game

6.3 com.battlefield.menu

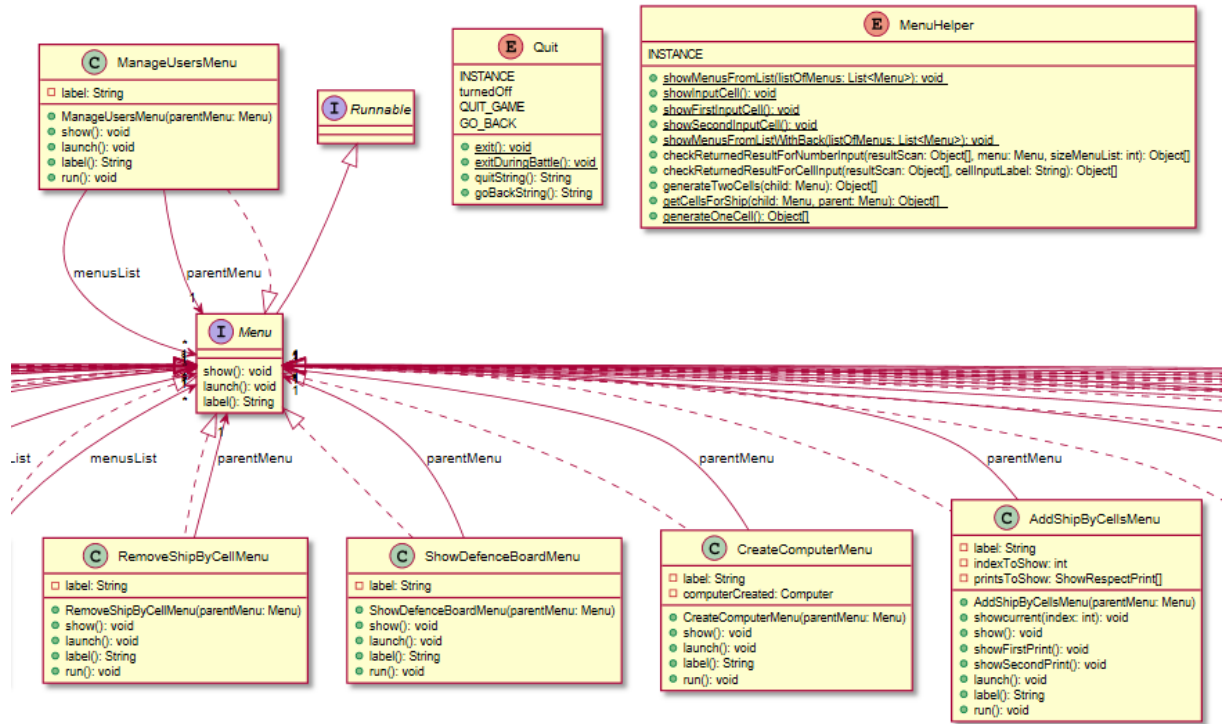


Figure 26: UML-diagram com.battlefield.menu

6.4 com.battlefield.player

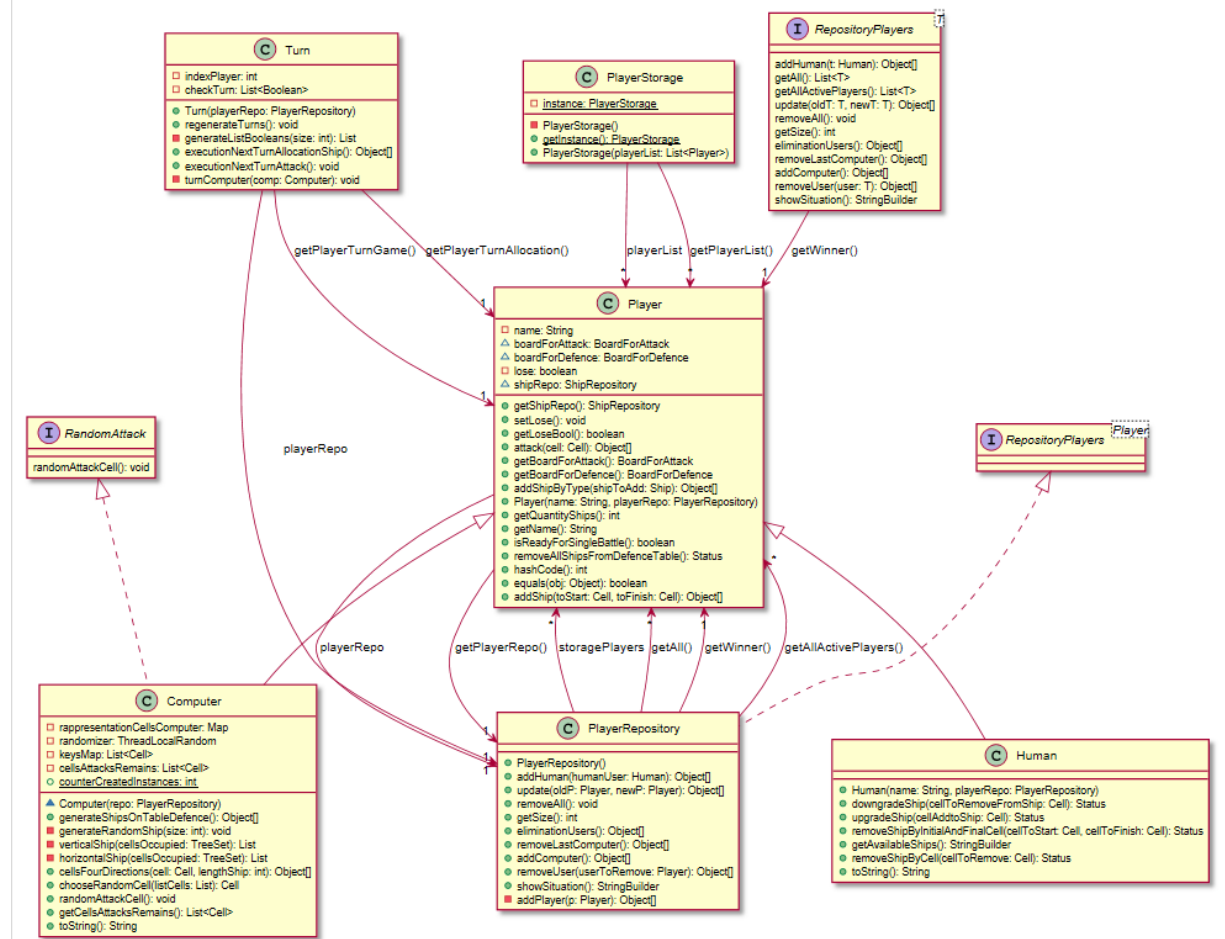


Figure 27: UML-diagram com.battlefield.player

6.5 com.battlefield.printer

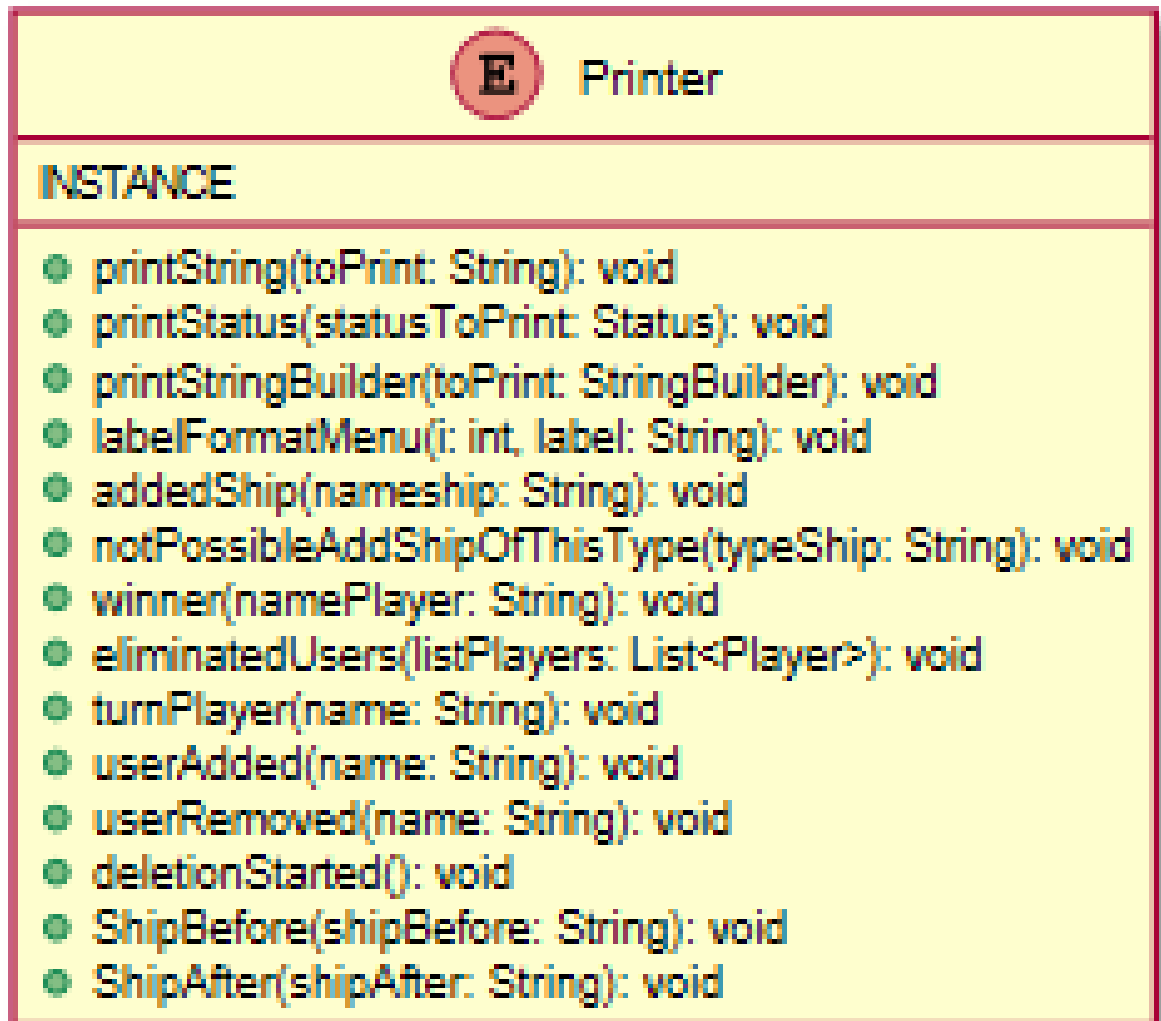


Figure 28: UML-diagram com.battlefield.printer

6.6 com.battlefield.security

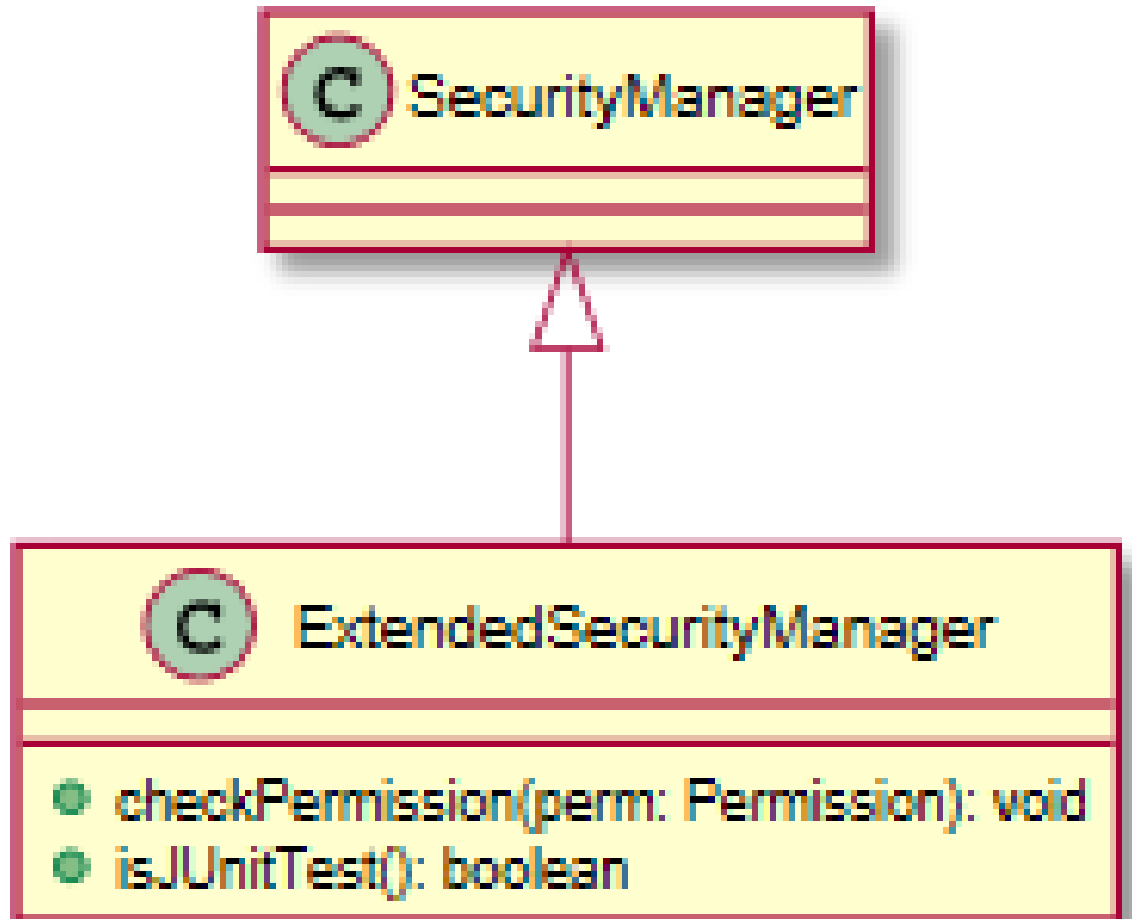


Figure 29: UML-diagram com.battlefield.security

6.7 com.battlefield.settings



Figure 30: UML-diagram com.battlefield.settings

6.8 com.battlefield.ships

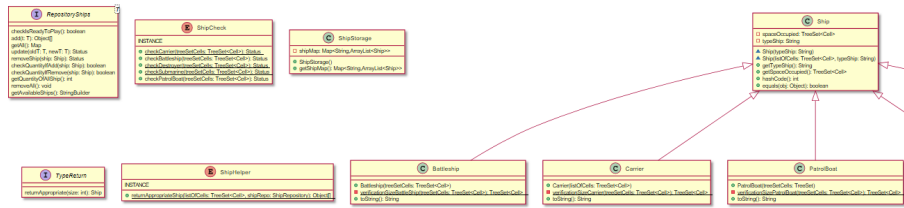


Figure 31: UML-diagram(1 part) com.battlefield.ships

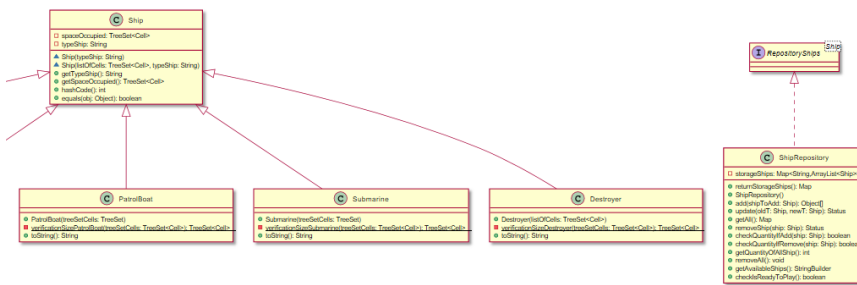


Figure 32: UML-diagram(2 part) com.battlefield.ships

6.9 com.battlefield.tests

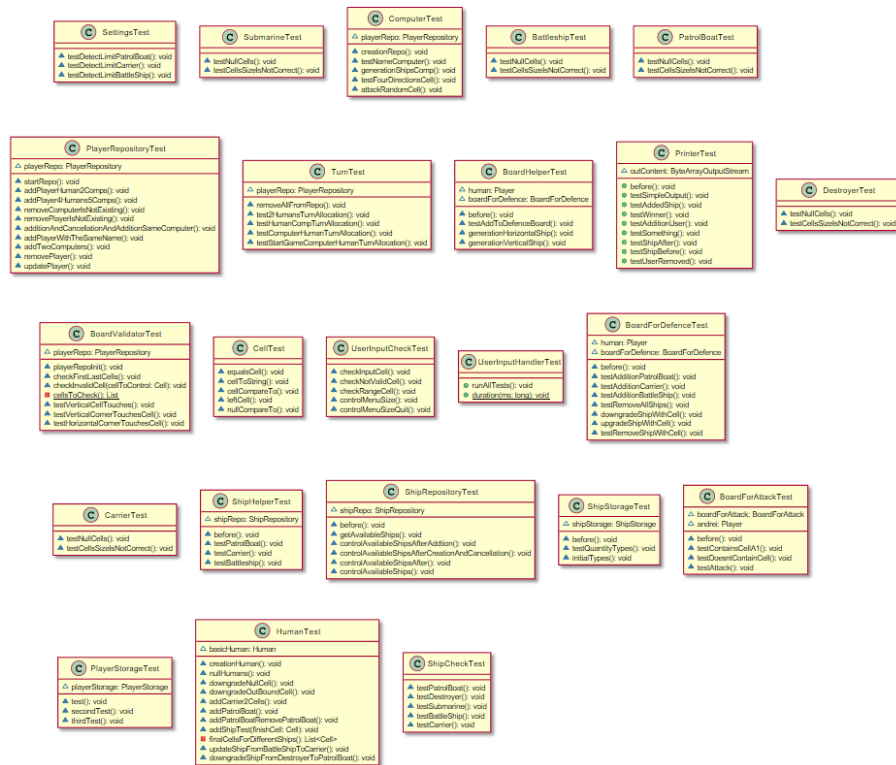


Figure 33: UML-diagram com.battlefield.tests

6.10 com.battlefield.userinput

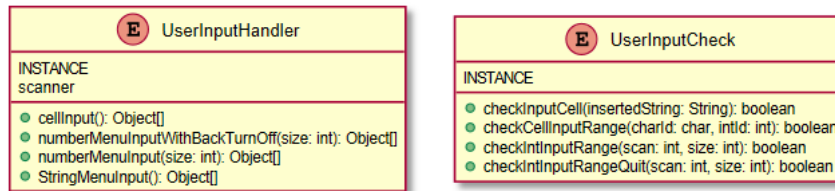


Figure 34: UML-diagram com.battlefield.userinput

7 Polymorphism

7.1 com.battlefield.ships

Ship.java is an abstract class(superclass). With mentioned class it is possible to add different ships to ShipRepository, get methods such as **getCellsOccupied()**, **getTypeShip()**.

7.2 com.battlefield.player

Ship.java is an abstract class(superclass) in package **com.battlefield.player**. Concept of polymorphism is analog as in **com.battlefield.ships**: permits to add players in PlayerRepository, returns names of users and their Ship Repositories.

7.3 com.battlefield.menu

Here polymorphism helps to switch between menus without dependence on certain classes, also it can be possible to change parent dynamically.