DOCUMENTATION

Battleship game

This is the documentation file of the Battleship game developed by by Dinh Dang Khoa, Doan Y Nhi and Nguyen The Anh.

1. Tools:
   - Main language: Python
   - Environment: Python 3.7 - Platform: Window (10) - Library for Python: ○ Pygame 1.9
     - ○ Numpy 1.16
     - ○ Pyinstaller 3.4 / 3.5 dev
   - Binaries for app-local deployment from Window Software Development Kit for Windows (10).

2. Structure and design:
   - The project is divided into 6 different modules, which all serve different purposes. These include:

     a. main.py : The main running code for the project.

     b. backend.py : serve as the background for the main.py by implementing the DSA_battleship.py.

     c. screen.py : contains design patterns used for the creation of the interface.

     d. screen_info.py : getting the optimal resolution for the game.

     e. config_game.py : setting the optimal resolution for the game.

     f. DSA_battleship.py : contain a fully functional text-based BattleshipGame class which can be utilized later.
   - The structure can help us to quickly identify the problem, and therefore make it easier to debug/ implement new feature.
   - The design of the class if of follow:

- DSA_battleship.py:
    - BattleshipGame class: contains most of the methods such as checking, validating position, etc.

```
def __init__(self, ships):
# Private @field
__userBoard
# Each user ship point to -> the length remained of its self
# Private @field __computerBoard
# Each user ship point to -> the length remained of its self
# Computer's targeting phase initially is False.
# Tracks computer hits and is used to determine whether or not to continue the
targeting phase.
# Target stack for computer in targeting mode, containing all possible and valid moves
following a hit by the computer.
# Dictionary used to determine the lowest hit points of remained ships to adjust
parity in computer hunting phase.
# Each item is a list of coordinate [int x, int y]
# Each item is a list of coordinate [int x, int y]
# List of user's ship names that been destroyed.
# List of computer's ship names that been destroyed.
# Dictionary of user's ship and its position {'Ship_Name': [int x, int y, str:
orientation], ...}
# Dictionary of user's ship and its position {'Ship_Name' : [int x, int y, str:
orientation], ...}
```

```
def drawBoard(self, hide):
    # Draw boards
    # Aircraft is A, Destroyer is D, Submarine is S, Patrol boat is P, Battleship is B
    # Rows are denoted by characters from A to J
    # Columns are denoted by numbers from 1 to 10
```

```
def makeA_Move(self, computer, x, y):
    # Return {' '} if it's a miss and return {'A', 'D', 'S', 'P', 'B'} if it's a hit
```

```
def validatePlacement(self, computer, ship, size, x, y, orientation):
    # Use the computer's or user's board depending on whether or not True has been
passed into the method.
    # With the appropriate board, check if the caller has provided x and y coordinates
of an empty cell.
    # If the orientation is vertical, check whether the x coordinate + the size of the
ship exceeds the bounds of
    # the game board, if so, return False.
    # Otherwise, check if each x and y coordinate of the ship would fill an empty
    # cell, if so, return True, otherwise return False.
    # This process is identical for checking valid horizontal placement except that
# the size of the ship is added to the y coordinate instead of the x coordinate to
check if the ship is     # within the bounds of the game board.
```

```
def getEnemyFleet(self, computer):
    # Create empty lists for the entire fleet of all ships sunk
    # If the computer is calling this method, iterate through the users's board,
otherwise iterate through the computers's board
    # For each ship in the ship dictionary defined in the init, if the hit points of
the ship are greater than zero,
    # call the what ship private method to get the full name of the ship, and then
append it to the ships to sink list.
    # Otherwise, if the hit points of the ship is 0, get the full name
    # of the ship as previously using the what ship method, and append it to the ships
sunk list.     # Finally, append both lists to the fleet list and return it.
```

```
def checkWinning(self, computer):
    # Pass true or false depending on whether the user or computer is calling the
method.
    # To determine if the computer or user has won, check the length of the ships to
sink list
    # generated by the get enemy fleet method. If there are still ships on the board
that have
    # not yet been sunk, return False, otherwise return True since all ships have been
sunk.
```

```
def checkIfSunk(self, computer, ship):
    # Check if all parts of the ship is hit
```

```
def incrementRounds(self):
    # increment the instance variable rounds by one after both the player and computer
have shot
```

```
def getHits(self, computer):
    # Pass true if returning hits for computer or false if checking hits for the
player.
    # Iterate through the game board and check for hits ('#'), if a hit is
encountered, increment     # the variable hits by one. After iterating, return hits.
```

```
def getMisses(self, computer):
    # Pass true if returning misses for computer or false if checking misses for the
player.
    # Iterate through the game board and check for misses ('#'), if a miss is
encountered, increment     # the variable misses by one. After iterating, return
misses.
```

```
def userPlaceShips(self, ship, size):
    # While ship placement is not valid, prompt the user to enter in x and y
coordinates for their shot.
    # After valid input has been accepted, ask the user for the orientation they would
like to place their ship and only accept either v or h.
    # After all input is validated, call the validate placement method to check
# if the ship can be placed on the board, if not, alert the user and prompt them to
enter a new set of coordinates and orientation.
    # If the ship can be placed on the board, alert the player that they have placed a
ship and the type of ship placed.
```

```
def computerPlaceShips(self, ship, size):
    # While a valid placement for a particular ship has not been randomly generated by
the computer,
    # generate random x and y coordinates between 0 and 9 and a random orientation
(either v or h).
    # Continue generating random coordinates until the ship can be placed on the
board.
```

```
def computerMakesMove(self):
    # Pass the battleShip game object, the size of the ships, and whether or not the
computer is in targeting mode.
    # Initialize parity variable as the minimum ship size still in play. If the
computer is in targeting mode,
    # pop a move from the targetStack and play that move. Otherwise, choose a random x
(letter) coordinate between 0 and 9
    # set the range of possible y coordinates (numbers) to the result of the modulo of
the x coordinate and the parity(min ship size).
    # Generate a random y coordinate in the defined range and then play the resulting
move.     # If the computer has fired a shot in the same location previously, generate
new x and y coordinates.
    # Otherwise, print a message informing the user that the computer has missed their
ship,     # or if the computer has hit the user's ship, check if it has sunk by calling
the check if sunk method.
    # If the ship has not been sunk, push to targetStack moves to the left, right,
top, and bottom of the hit and return True(True is targeting mode).
```

```
def userMakesMove(self):
    # Until a valid move is played on the board, ask the user for coordinates for
their shot.
    # If the a shot has already been taken at that location, alert the user and prompt
them to enter another set of coordinates.
    # If the user has missed, print a message and return.
    # If the user has hit a ship, print a message indicating they hit a ship and then
call the check if sunk method.
```

```
def userMakesMoveAtXY(self, x, y):
```

```
def getLatestShot(self, isComputer):
```

```
def getLatestSunkShipPosition(self, isComputer):
```

```
def getLatestSunkShipName(self, isComputer):
```

✦ These 3 functions do what the name suggest.

```
def getComputerPlacedShips(self):  # Get computer's ships name & position
def userInput():
    # Until the user enters in valid input, prompt them to enter a letter and number
for the x and y coordinate respectively.
    # To do this, check if the user's input consists of two elements separated by a
space,
    # if so check if the x coordinate consists of a letter and the y coordinate
consists of a number.
    # Next convert the y coordinate to an integer and check if it is within 1 and 10.
    # Next check if the x coordinate is between the letters a and j.     # Finally
return the x and y coordinates as integers between 0 and 9.
```

- Exception Handling class:

```
class UserShipSunkException(Exception):
    """ Raised when a new ship of user been sunk !"""
pass  # End UserShipSunkException constructor !
  class
ComputerShipSunkException(Exception):
    """ Raised when a new ship of computer been sunk !"""
pass  # End ComputerShipSunkException constructor !
```

- Backend.py: contain the implementation of DSA_battleship.py.

```
import DSA_battleship as bs
class Backend:      # Call
DSA_battleship class.
```

```
@staticmethod
def
start_game():
    # The dictionary @{ships}'s key is the 1st letter of ship's name and the
associated values are ship's length
```

```
@staticmethod
def set_computer_ship(): """ Placing computer' ships into Backend.battleShip """
```

```
@staticmethod def
set_user_ship(ships: dict):
    """ The @parameter ships is a dictionary. Each key point to a list contains 3 main
ships' properties.
```

```
@staticmethod
def check_win():
    """ Return Winner in format [bool,
bool]     false, true : Computer Win
true, false : User Win     false, false :
Neither Win
    """
```

```
@staticmethod def
user_hit_at(x: int, y: int):
    """ Return User's shot, True if hit, False if miss or already shoot at [x, y] !"""
```

```
@staticmethod def
computer_hit_at():
    """ Return computer's shot position at [x, y] !"""
```

```
@staticmethod def
end_game():
```

✦ Contain conditions for ending the game.

```
@staticmethod def
get_score():
```

    - Getting the current score.

```
@staticmethod def
check():
```

    - Check for flag of sunken ships.

```
@staticmethod def
get_health(name:str):
```

    - Get health of current available ships.

  - Screen: setting up the interface
    - Draw class

```
"""Interface to draw, an abstract method to be overridden""" class Draw():
@abstractmethod
        def draw(self):
        pass
```

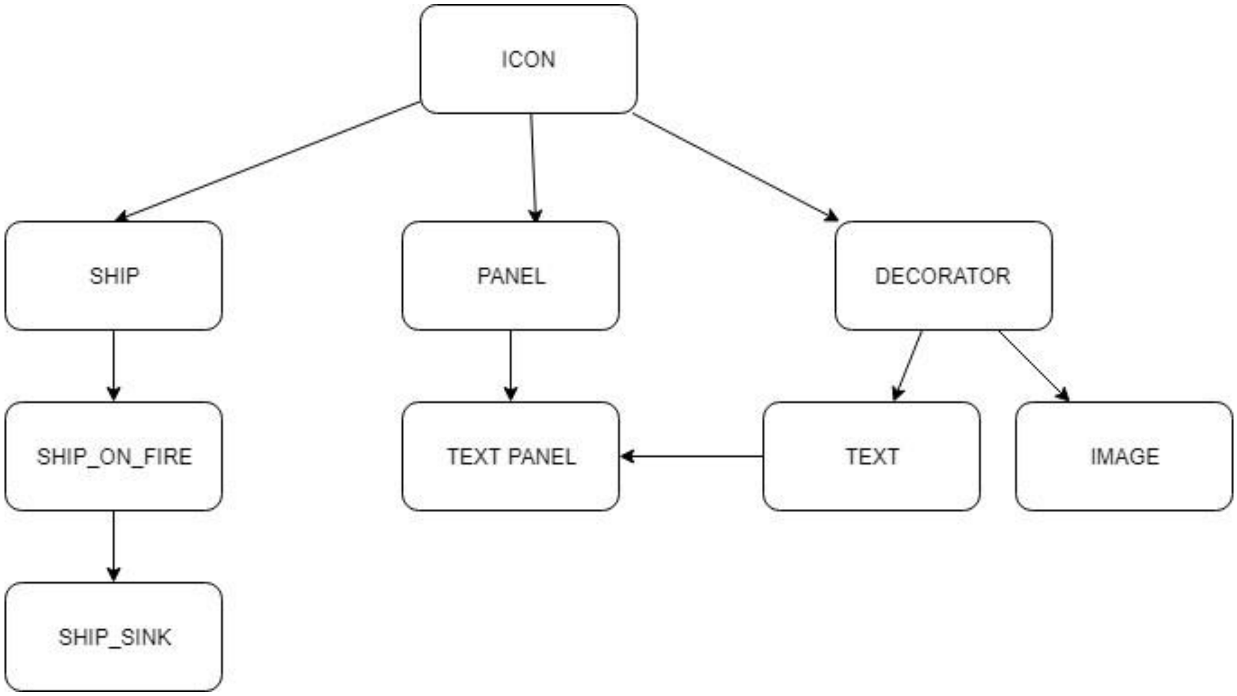
    - JustDraw class

```
"""Class to draw a Pygame surface"""
class JustDraw(Draw):
      def __init__(self,surface:pygame.Surface,display:pygame.display,x:int,y:int):
      def draw(self):
```

    - The base classes of objects in the game:

```
class Icon(Draw):        """create an icon"""
class Decorator(Icon): """create an decorator for the icon""" class
Panel(Icon):  """create a panel inherit from the icon""" class
Text(Decorator): """create a text on top of the panel""" class
Image(Decorator): """create an image on top of the panel"""
class Grid(Icon):   """base grid class""" class
Ship(Icon):   """create ship based on icon""" class
Computer_ship(Ship):"""create ship for computer"""
class Ship_on_fire(Ship): """decorator for ship"""
class Sunk(Ship):   """decorator for ship""" class
BuildShip():
```

The following diagram illustrates the relative relationship between the object base classes:

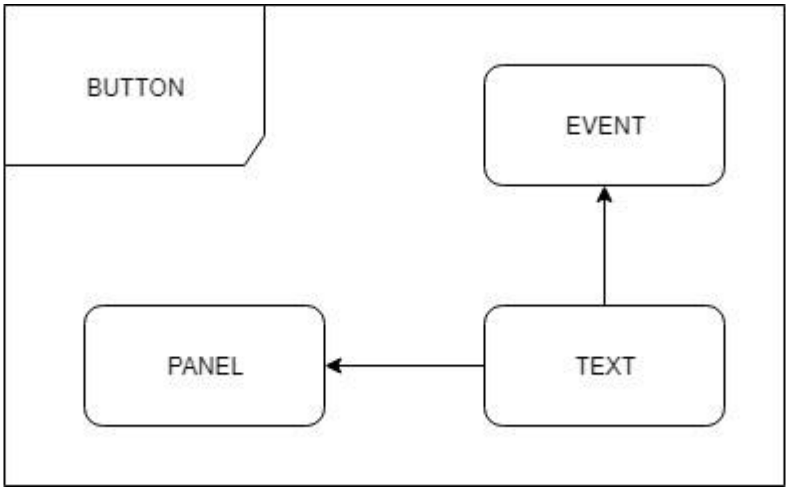


- ✦ The event handling classes:

```
class Event(Draw):   class
Interactive_Ship(Event): class
Interactive_Board(Event):
class Player_Board(Draw):
class Computer_Board(Event):
class Button(Event): class
Info(Draw):
class Count_Down(Draw):
class Winner(Draw):
class Loser(Draw): class
Shell(Icon): class
Missile(Icon): class
Effect_Board(Draw):
class Score_Board(Icon):
class Event_Subject():
class Listener():
```

These classes will respond when the condition is right, sending back the signal to the object classes or trigger a state in the main class. The event class acts as an observer and will react to a certain condition/ action.

The following diagram illustrates the relative relationship between classes in a button:



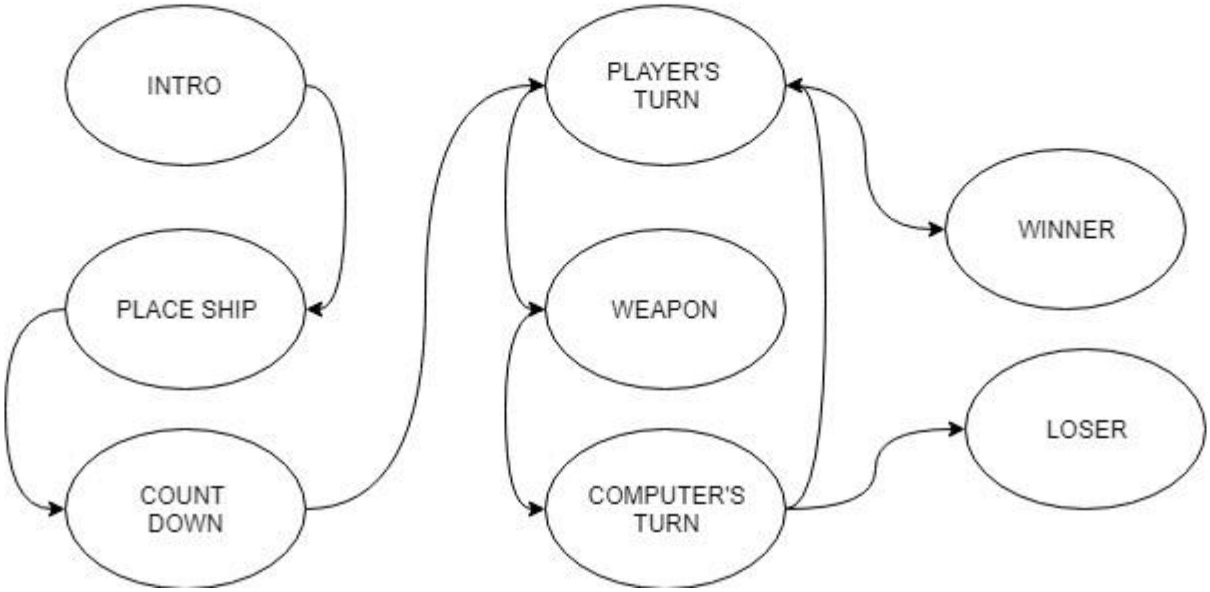
- main.py: contains the classes that are required to build the states of the game.
  - ✦ Hu class contains the attributes as well as the 2 static methods to get the game running.

```
class Hu():
         @staticmethod
         def run_game():
       @staticmethod            def load():
```

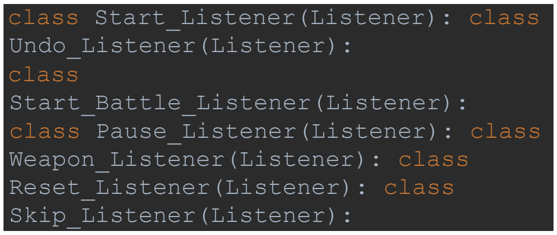
- ✦ The following classes are the states which the game will eventually go through. They all derived from an abstract state at the beginning.

```
class Abstract_State(Event_Subject):
         @abstractmethod
        def _build_data(self): class
Intro_State(Abstract_State): class
Place_Ship_State(Abstract_State): class
Count_Down_State(Abstract_State): class
Player_State(Abstract_State): class
Weapon_State(Abstract_State): class
Computer_State(Abstract_State): class
Winner_State(Abstract_State): class
Loser_State(Abstract_State):
```

The following diagram illustrates the states of the games:

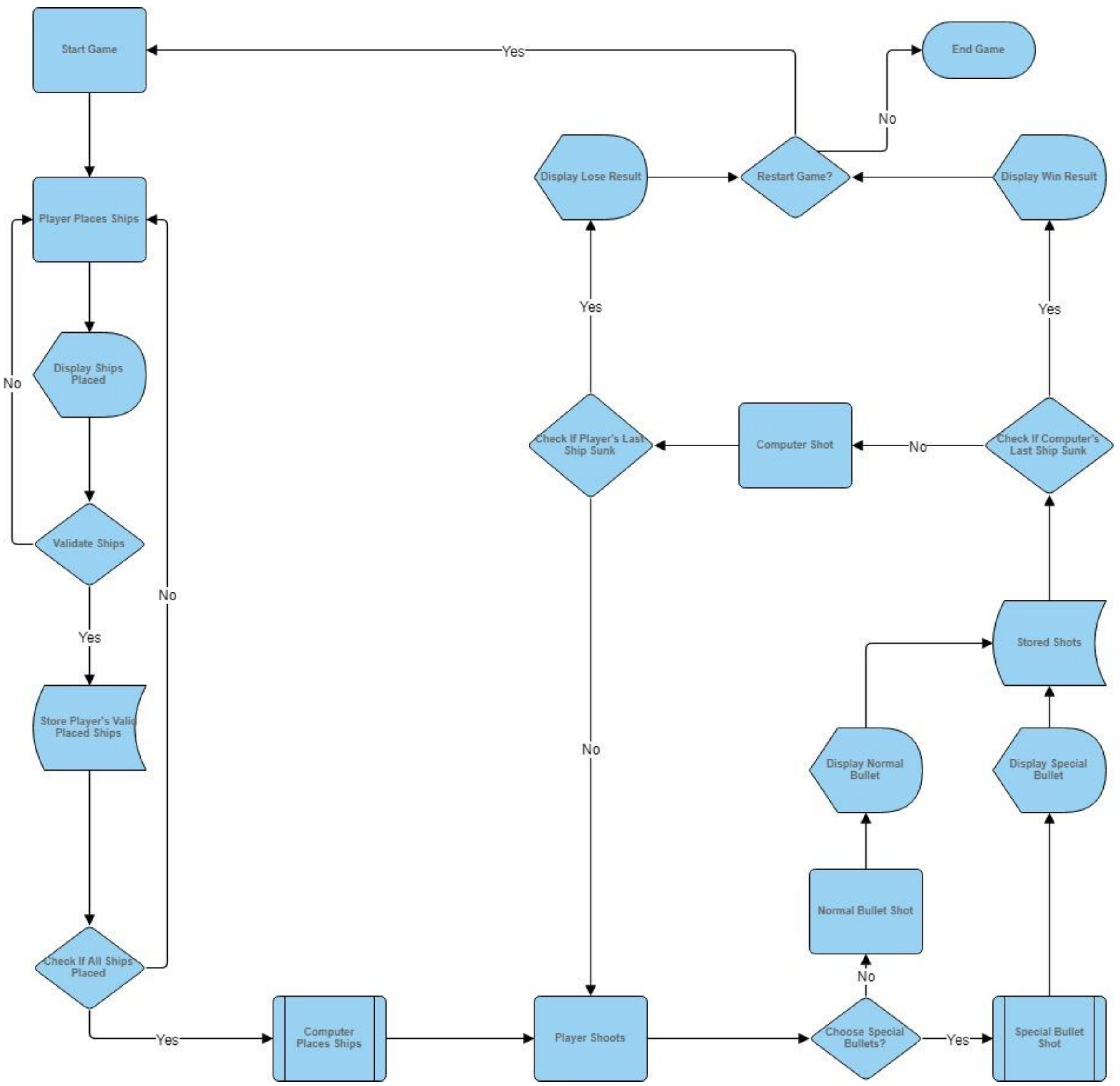


- ✦ The following classes receive the listener from screen.py above and react to it accordingly.



```
class Start_Listener(Listener): class
Undo_Listener(Listener):
class
Start_Battle_Listener(Listener):
class Pause_Listener(Listener): class
Weapon_Listener(Listener): class
Reset_Listener(Listener): class
Skip_Listener(Listener):
```

Design Flowchart


Start Game
Yes
End Game
No
Player Places Ships
Display Lose Result
Restart Game?
Display Win Result
Display Ships Placed
No
Yes
Yes
Check If Player's Last Ship Sunk
Computer Shot
No
Check If Computer's Last Ship Sunk
Validate Ships
No
Yes
Stored Shots
Store Player's Valid Placed Ships
No
Display Normal Bullet
Display Special Bullet
Normal Bullet Shot
No
Check If All Ships Placed
Yes
Computer Places Ships
Player Shoots
Choose Special Bullets?
Yes
Special Bullet Shot