Project Sprint #2

The SOS game is described in CS449HomeworkOverview.docx. You should read the description very carefully.

Your submission must include the GitHub link to your project and you must ensure that the instructor has the proper access to your project. You will receive no points otherwise.

GitHub link: https://github.com/tchk7/449

Implement the following features of the SOS game: (1) the basic components for the game options (board size and game mode) and initial game, and (2) S/O placement for human players ***without*** checking for the formation of SOS or determining the winner. The following is a sample interface. The implementation of a GUI is required. You should practice object-oriented programming, making your code easy to extend. It is required to separate the user interface code and the game logic code into different classes (refer to the TicTacToe example). xUnit tests are required.

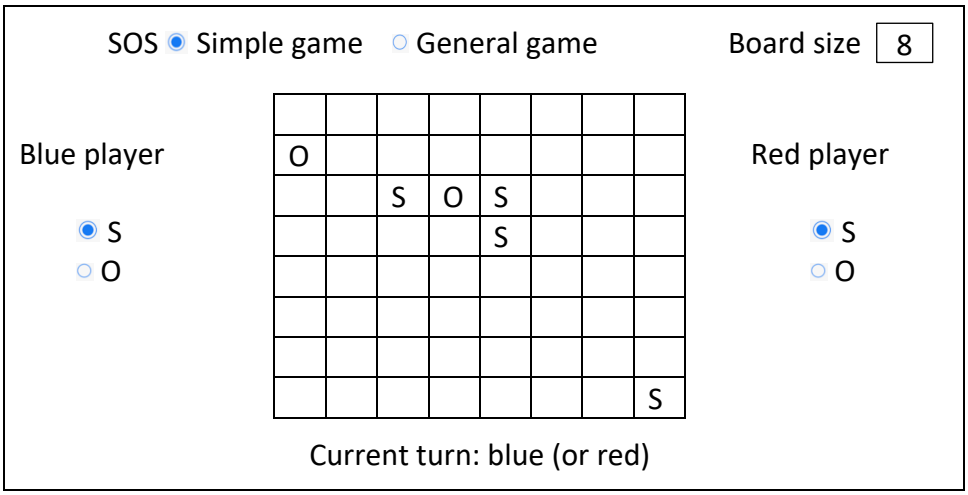

Figure 1. Sample GUI layout of the Sprint 2 program

Deliverables:

1. **Demonstration (8 points)**

   Submit a link to a video of no more than three minutes, clearly demonstrating that you have implemented the required features and written some automated unit tests. In the video, you must explain what is being demonstrated. No points will be given without a video link.
   YouTube/Panopto link:
   https://umsystem.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=db82bbb3-0ecf-4cc0-8b9d-b37e002c91c3

	Feature
1	Choose board size
2	Choose game mode
3	Start a new game of the chosen board size and game mode
4	"S" moves
5	"O" moves
6	Automated unit tests

2. Summary of Source Code (1 points)

Source code file name	Production code or test code?	# lines of code
board.py	Production	52
player.py	Production	10
game.py	Production	83
player_ui.py	Production	39
board_ui.py	Production	36
game_ui.py	Production	91
test_player.py	Test	6
test_board.py	Test	50
test_game.py	Test	85
	Total	452

You must submit all source code to get any credit for this assignment.

3. Production Code vs User stories/Acceptance Criteria (3 points)

Update your user stories and acceptance criteria from the previous assignment and ensure they adequately capture the requirements. Summarize how each of the following user story/acceptance criteria is implemented in your production code (class name and method name etc.)

User Story ID	User Story Name
1	Choose a board size
2	Choose the game mode of a chosen board
3	Start a new game of the chosen board size and game mode
4	Make a move in a simple game
6	Make a move in a general game

User Story ID and Name	AC ID	Class Name(s)	Method Name(s)	Status (complete or not)	Notes (optional)
1. Choose a board size	**1.1** <Starting Game>	Board	__init__	Complete	
	1.2 <Select Board Size>	GameUI, Game	start_new_game	Complete	
2. Choose game mode of a chosen board	**2.1** <Default Game Mode>	GameUI	__init__, simple_radio.setChecked	Complete	
	2.2 <Choosing Game Mode>	GameUI, Game	__init__, update_game_mode	Complete	
3. Start new game mode with chose board	**3.1** <Starting New Game>	Game, Board	start_new_game, reset_board	Complete	
	3.2 <Start New Game Different Size>	GameUI, Game	start_new_game	Complete	
	3.3 <Start New Game	GameUI, Game	start_new_game, update_game_mode	Complete	

	Different Mode>				
4. Make move in simple game	**4.1** <Invalid Move>	Game, Board	handle_click, is_empty	Complete	
	4.2 <Valid Move>	Game, PlayerUI, Board, BoardUI	handle_click, get_selected_letter, put_letter, setText	Complete	
6. Make move in a general game	**6.1** <Invalid Move>	Game, Board	handle_click, is_empty	Complete	
	6.2 <Valid Move>	Game, PlayerUI, Board, BoardUI	handle_click, get_selected_letter, put_letter, setText	Complete	

4. Tests vs User stories/Acceptance Criteria (3 points)

Summarize how each of the user story/acceptance criteria is tested by your test code (class name and method name) or manually performed tests.

User Story ID	User Story Name
1	Choose a board size
2	Choose the game mode of a chosen board
3	Start a new game of the chosen board size and game mode
4	Make a move in a simple game
6	Make a move in a general game

4.1 Automated tests directly corresponding to the acceptance criteria of the above user stories
You are required to use ChatGPT to create at least 2 unit tests. You also need to ensure that the generated user stories are correct, and refine them if not. At the end of the submission, provide the screenshots of your ChatGPT prompts and answers, along with errors ChatGPT made and you fixed. You may also use another LLM, including hosted locally. Points will be deducted if no screenshots are provided.

User Story ID and Name	Acceptance Criterion ID	Class Name (s)	Method Name(s)	Description of the Test Case (input & expected output)
1. Choose a board size	1.1	test_board.py	test_default_board	board = Board(), assert board.size == 3
	1.2	test_game.py	test_start_new_game_with_resize	Set text box to "5", click "New Game", assert board.size == 5
	1.3	test_board.py	test_minimum_board_size	board = Board(size=2), assert board.size == 3
2. Choose game mode	2.1	test_game.py	test_game_mode_default	Create Game, assert controller.game_mode == "Simple"
	2.2	test_game.py	test_game_mode_change	Simulate click on "General" radio, assert controller.game_mode == "General"

3. Start new game mode with chose board	3.1	test_game.py	test_start_new_game	Make move, click "New Game" (empty text), assert board.is_empty(1,1)
4. Make move	4.1	test_board.py	test_non_empty_cell	board.put_letter(0, 0, "S"), assert board.put_letter(0, 0, "O") == False
	4.2	test_board.py	test_put_letter	board.put_letter(0, 0, "S"), assert board.get_cell(0, 0) == "S"

4.2 Manual tests directly corresponding to the acceptance criteria of the above user stories

User Story ID and Name	Acceptance Criterion ID	Test Case Input	Test Oracle (Expected Output)	Notes
1. Choose a board size	1.2	Type "12" into "Board Size" box. Click "New Game".	The grid is replaced by a 12x12 grid.	
	1.3	Type "2" into the text box. Click "New Game".	The grid becomes a 3x3 grid (defaults to min).	
	1.4	Try to type "13" into the "Board Size" box.	The text box does not accept the "13" reverts to “3”.	
2. Choose game mode	2.2	Click "General Game" radio button.	"Simple Game" unchecks. Console prints "Game mode set to: General".	
3. Start new game...	3.1	Place an "S" at (0,0). Leave "Board Size" box empty. Click "New Game".	The "S" at (0,0) disappears. The board is still 3x3. Turn resets to Blue.	

4.3 Other automated or manual tests not corresponding to the acceptance criteria of the above user stories

Number	Test Input	Expected Result	Class Name of the Test Code	Method Name of the Test Code
1	Create a 3x3 board and fill every cell.	board.is_full() returns True.	test_board.py	test_full_board
2	Create a default 3x3 board.	board.is_full() returns False.	test_board.py	test_full_board
3	Try to type "abc" into the "Board Size" text box.	The input is ignored due to QIntValidator.	Manual Test	

Gemini

I used Google's Gemini to generate the two tests in test_game. Giving it all my code allowed it to create good test that passed so there were no changes from me. One thing I found is that it went over the top with the tests compared to how I wrote my other test. That would be the only thing I change in future iterations unless it good practice to write them that way.

I am building an sos game. I need 2 tests for the following criteria: I need to test that the default game mode is a simple game. I will also need to test the changing of game modes. I currently only have 2 game modes simple and general so once default is tested, the I need to test that I can toggle and change to general. here is the relevant code:

game.py: from functools import partial

```
from sprint_2.model.board import Board


class Game():
    def __init__(self, game_ui):
        self.game_ui = game_ui
        self.board_ui = self.game_ui.get_board_ui()
        self.player_uis = self.game_ui.get_player_uis()
        self.current_player = 0

        self.buttons = []
        self.connect_buttons()

        self.game_mode = "Simple"


self.game_ui.simple_radio.toggled.connect(self.update_
game_mode)

self.game_ui.general_radio.toggled.connect(self.update_
game_mode)

        # self.buttons = self.board_ui.get_buttons()
        #
        # for row_index, row in enumerate(self.buttons):
        #     for col_index, btn in enumerate(row):
        #         btn.clicked.connect(partial(self.handle_click,
row_index, col_index))
```

```
self.game_ui.new_game.clicked.connect(self.start_new_
game)

    def connect_buttons(self):
        self.buttons = self.board_ui.get_buttons()
        for row_index, row in enumerate(self.buttons):
            for col_index, btn in enumerate(row):
                btn.clicked.connect(partial(self.handle_click,
row_index, col_index))

    # def board_change(self, size):
    #     new_board = Board(size)
    #
    #     self.board_ui =
self.game_ui.build_board_ui(new_board)
    #
    #     self.connect_buttons()
    #
    #     self.start_new_game()


    def handle_click(self, row, col):
        board = self.board_ui.get_board()

        if not board.is_empty(row, col):
            return

        player_ui = self.player_uis[self.current_player]

        letter = player_ui.get_selected_letter()

        board.put_letter(row, col, letter)
        self.buttons[row][col].setText(letter)

        self.current_player = 1 - self.current_player

        next_player = self.player_uis[self.current_player]
```

```
        self.game_ui.player_turn_label.setText(f"
{next_player.get_player().name}'s Turn")

    def start_new_game(self):

        text = self.game_ui.board_size_text_box.text()
        if text.isnumeric():
            size = int(text)
        else:
            size = 3

        current_size = self.board_ui.get_board().size

        if size != current_size:

            board = Board(size)

            self.board_ui =
self.game_ui.build_board_ui(board)

            self.connect_buttons()
        else:
            self.board_ui.get_board().reset_board()


        for row in self.buttons:
            for btn in row:
                btn.setText("")

        self.current_player = 0

        next_player = self.player_uis[self.current_player]

        self.game_ui.player_turn_label.setText(f"
{next_player.get_player().name}'s Turn")

    def update_game_mode(self):
        if self.game_ui.simple_radio.isChecked():
            self.game_mode = "Simple"
        elif self.game_ui.general_radio.isChecked():
            self.game_mode = "General"
```

player.py:

```
class Player:
    def __init__(self, name):
        self.name = name + " Player"
        self.letter = ""

    def set_letter(self, letter_choice):
        if letter_choice.upper() in ("S", "O"):
            self.letter = letter_choice
        else:
            raise ValueError("Letter must be an 'S' or 'O'")
```

board.py:

```
class Board:
    def __init__(self, size = 3):
        if size < 3:
            size = 3
        self.size = size
        self.grid = self._create_grid()

    def _create_grid(self):
        """Create an empty grid"""

        return [["" for _ in range(self.size)] for _ in
range(self.size)]

    def is_empty(self, row, col):
        """Check if cell is empty"""
        return self.grid[row][col] == ""

    def is_full(self):
        """Check if board is full"""
        for row in range(self.size):
```

```
        for col in range(self.size):
            if self.is_empty(row, col):
                return False
    return True

def validate_move(self, row, col):
    """Check if a move is valid"""

    return (0 <= row < self.size and
            0 <= col < self.size and
            self.is_empty(row, col)
            )

def put_letter(self, row, col, letter):
    """Place letter on board"""

    if self.validate_move(row, col):
        self.grid[row][col] = letter
        return True
    else:
        return False

def get_cell(self, row, col):
    """Return cell value"""

    return self.grid[row][col]

def reset_board(self):
    """Reset board to empty"""

    self.grid = self._create_grid()
```

board ui.py: from PySide6.QtWidgets import QWidget, QVBoxLayout, QGridLayout, QApplication, QPushButton
import sys

from sprint_2.model.board import Board

```
from sprint_2.model.board import Board


class BoardUI(QWidget):
    def __init__(self, board):
        super().__init__()

        self.setWindowTitle("SOS Game")
        self.board = board

        self.layout = QVBoxLayout()
        self.setLayout(self.layout)

        self.grid_layout = QGridLayout()
        self.layout.addLayout(self.grid_layout)

        self.buttons = []
        for row in range(self.board.size):
            button_row = []
            for col in range(self.board.size):
                btn = QPushButton("")
                btn.setFixedSize(60, 60)
                self.grid_layout.addWidget(btn, row, col)
                button_row.append(btn)
            self.buttons.append(button_row)



    def get_buttons(self):
        return self.buttons

    def get_board(self):
        return self.board



    # for rowIndex in range(self.board.size):
    #     for colIndex in range(self.board.size):
    #         cell = QLabel(f"[{rowIndex + 1}, {colIndex + 1}]")
```

```
    # for rowIndex in range(self.board.size):
    #     for colIndex in range(self.board.size):
    #         cell = QLabel(f"[{rowIndex + 1}, {colIndex + 1}]")
    #         cell.setAlignment(Qt.AlignCenter)
    #         self.grid_layout.addWidget(cell, rowIndex,
colIndex)

    # def paintEvent(self, event):
    #     painter = QPainter(self)
    #
    #     width = self.width()
    #     height = self.height()
    #
    #     for col in range(1, self.cols):
    #         x = colIndex * width / self.cols
    #         painter.drawLine(int(xPosition), 0, int(xPosition),
height)
    #
    #     for row in range(1, self.rows):
    #         yPosition = rowIndex * height / self.rows
    #         painter.drawLine(0, int(yPosition), width,
int(yPosition))

if __name__ == "__main__":

    board = Board()

    app = QApplication(sys.argv)
    window = BoardUI(board)
    window.show()
    sys.exit(app.exec())

player ui.py: from PySide6.QtCore import Qt
from PySide6.QtWidgets import QWidget, QVBoxLayout,
QLabel, QApplication, QRadioButton
import sys

from sprint_2.model.player import Player
```

```
from sprint_2.model.player import Player

class PlayerUI(QWidget):
    def __init__(self, player):
        super().__init__()

        self.player = player

        player_name = QLabel(player.name)
        player_name.setAlignment(Qt.AlignCenter)

        self.letter_s = QRadioButton("S")
        self.letter_s.setChecked(True)
        self.letter_o = QRadioButton("O")

        player_layout = QVBoxLayout()

        player_layout.addWidget(player_name)
        # player_layout.addSpacing(5)
        player_layout.addWidget(self.letter_s,
alignment=Qt.AlignCenter)
        player_layout.addWidget(self.letter_o,
alignment=Qt.AlignCenter)

        self.setLayout(player_layout)

    def get_selected_letter(self):
        """Get letter player wants to play"""

        if self.letter_s.isChecked():
            return "S"
        elif self.letter_o.isChecked():
            return "O"
        return None

    def get_player(self):
        return self.player
```

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    player = Player("Blue")
    window = PlayerUI(player)
    window.show()
    sys.exit(app.exec())

game ui.py: import sys

from PySide6.QtCore import Qt
from PySide6.QtGui import QIntValidator
from PySide6.QtWidgets import QWidget, QHBoxLayout,
QCheckBox, QLabel, QLineEdit, QGridLayout,
QApplication, \
    QRadioButton, QPushButton

from sprint_2.controller.game import Game
from sprint_2.model.board import Board
from sprint_2.model.player import Player
from sprint_2.view.board_ui import BoardUI
from sprint_2.view.player_ui import PlayerUI


class GameUI(QWidget):

    def __init__(self):
        super().__init__()
        self.setWindowTitle("SOS Game")

        self.game_board = Board()
        self.board_ui = BoardUI(self.game_board)
        self.blue_player = Player("Blue")
        self.blue_player_ui = PlayerUI(self.blue_player)
        self.red_player = Player("Red")
        self.red_player_ui = PlayerUI(self.red_player)

        self.new_game = QPushButton("New Game")

        self.game_type_label = QLabel("Game Mode:")
```

```
        self.game_type_label = QLabel("Game Mode:")
        self.simple_radio = QRadioButton("Simple Game")
        self.simple_radio.setChecked(True)
        self.general_radio = QRadioButton("General Game")

        radio_layout = QHBoxLayout()
        # checkbox_layout.addWidget(game_type_label)
        radio_layout.addWidget(self.game_type_label)
        radio_layout.addWidget(self.simple_radio)
        radio_layout.addWidget(self.general_radio)



        # mode_size_layout = QVBoxLayout()
        # mode_size_layout.addWidget(game_type_label,
alignment=Qt.AlignCenter)
        # mode_size_layout.addLayout(checkbox_layout)

        board_size_label = QLabel("Board Size")
        self.board_size_text_box = QLineEdit()

self.board_size_text_box.setValidator(QIntValidator(3,
12))  # Only allows 3–15

self.board_size_text_box.setPlaceholderText("Choose 3
to 12")
        #
self.board_size_text_box.returnPressed.connect(self.set
_board_size)

        board_size_layout = QHBoxLayout()
        board_size_layout.addWidget(board_size_label)

board_size_layout.addWidget(self.board_size_text_box)



        self.player_turn_label = QLabel(f"
{self.blue_player.name}'s Turn",
alignment=Qt.AlignCenter)
```

```
        self.grid = QGridLayout()
        self.setLayout(self.grid)

        self.grid.addLayout(radio_layout, 0, 0, 1, 2)
        # self.grid.addWidget(self.game_type_label, 0, 0, 1,
1)
        # self.grid.addWidget(self.simple_radio, 0, 1, 1, 1)
        # self.grid.addWidget(self.general_radio, 0, 2, 1, 1)
        # self.grid.addLayout(mode_size_layout, 0, 1, 1, 2)
        self.grid.addLayout(board_size_layout, 0, 3, 1, 1)
        self.grid.addWidget(QWidget(), 0, 2, 1, 1)
        self.grid.addWidget(self.blue_player_ui, 2, 0, 1, 1)
        self.grid.addWidget(self.board_ui, 1, 1, 3, 3)
        self.grid.addWidget(self.red_player_ui, 2, 4, 1, 1)
        self.grid.addWidget(self.player_turn_label, 4, 0, 1, 5)
        self.grid.addWidget(self.new_game, 0, 4, 1, 1)

        self.resize(500, 500)

        self.controller = Game(self)

    def get_player_uis(self):
        return [self.blue_player_ui, self.red_player_ui]

    def get_board_ui(self):
        return self.board_ui

    # def set_board_size(self):
    #     text = self.board_size_text_box.text()
    #     if not text.isnumeric():
    #         return
    #
    #     size = int(text)
    #
    #     self.controller.board_change(size)

    def build_board_ui(self, board):

        self.grid.removeWidget(self.board_ui)
```

```
        self.board_ui.deleteLater()

        self.game_board = board
        self.board_ui = BoardUI(self.game_board)
        self.grid.addWidget(self.board_ui, 1, 1, 3, 3)

        return self.board_ui

    # def get_new_game_button(self):
    #     return self.new_game


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = GameUI()
    window.show()
    sys.exit(app.exec())
```

the tests should be similar structure to these however I understand you may need something to test app interaction.

test board.py: from sprint_2.model.board import Board

```
def test_default_board():
board = Board()
assert board.size == 3

def test_initial_board():
board = Board(size= 3)
assert board.size == 3
for row in range(board.size):
for col in range(board.size):
assert board.get_cell(row, col) == ""

def test_minimum_board_size():
board = Board(size=2)
assert board.size == 3

def test_put_letter():
board = Board(3)
letter_played = board.put_letter(0, 0, "S")
assert letter_played == True
assert board.get_cell(0, 0) == "S"

def test_non_empty_cell():
board = Board(3)
assert board.is_empty(0, 0)
board.put_letter(0, 0, "S")
assert not board.is_empty(0, 0)
letter_played = board.put_letter(0, 0, "O")
assert letter_played == False

def test_full_board():
board = Board(3)
assert board.is_full() == False

for row in range(board.size):
for col in range(board.size):
board.put_letter(row, col, "S")
assert board.is_full() == True

def test_reset_board():
board = Board(3)
for row in range(board.size):
for col in range(board.size):
board.put_letter(row, col, "S")
assert not board.is_empty(0, 0)
assert board.is_full() == True
board.reset_board()
assert board.is_full() == False
```

test player.py:

```
from sprint_2.model.player import Player
from sprint_2.model.board import Board

def test_player_letter_choice():
blue = Player("Blue")
```

```
def test_player_letter_choice():
blue = Player("Blue")
blue.set_letter("S")
assert blue.letter == "S"

def test_player_move():
board = Board()
blue = Player("Blue")
blue.set_letter("S")

assert board.is_empty(0, 0) == True

blue.make_move(board, 0, 0)

assert board.is_empty(0, 0) == False
assert board.get_cell(0, 0) == "S"

let me know if you need any further clarification
```

Show thinking

Of course. Based on the code you've provided, I've created the necessary tests. To test the game logic without needing to run the full graphical user interface, I've used mock objects to simulate the UI components.

Here is the test file you requested.

SOS Game Tests
Oct 21 at 7:14 PM

Open

Try again without Canvas

I've created a single file, `test_game.py`, containing two tests that verify your required criteria. I used the `pytest` framework and its fixture system, which is consistent with the test examples you provided.

Let me know if you need tests for any other part of your SOS game!

SOS Game Tests

```python
import sys
from unittest.mock import MagicMock

import pytest
from PySide6.QtWidgets import QApplication

# This assumes your project structure allows this import.
# If you run into ModuleNotFoundError, you may need to adjust your Python path.
from sprint_2.controller.game import Game


@pytest.fixture(scope="session")
def qapp():
    """
    This fixture creates a QApplication instance for the entire test session.
    A QApplication is necessary to test any code that interacts with Qt widgets,
    even if the widgets are not physically shown.
    """
    # Check if a QApplication instance already exists
    app = QApplication.instance()
    if app is None:
        # If not, create one
        app = QApplication(sys.argv)
    return app


@pytest.fixture
def game_with_mock_ui(qapp):
    """
    This fixture sets up a Game instance with a mocked UI.
    Mocking the UI allows us to test the game's logic in isolation,
    without needing to interact with a live GUI.
    """
    # 1. Create a mock object that pretends to be the GameUI
    mock_game_ui = MagicMock()

    # 2. The Game's __init__ method calls get_board_ui() and get_player_uis().
    #    We need to ensure these methods exist on our mock and return other mocks.
    mock_game_ui.get_board_ui.return_value = MagicMock()
    mock_game_ui.get_player_uis.return_value = [MagicMock(), MagicMock()]

    # 3. Instantiate the Game class, passing in our mocked UI
    game = Game(mock_game_ui)

    # 4. Return both the game instance and the mock UI so our tests can use them
    return game, mock_game_ui

```

```python


def test_default_game_mode_is_simple(game_with_mock_ui):
    """
    Given a new game instance,
    When the game is initialized,
    Then the default game mode should be "Simple".
    """
    # The fixture 'game_with_mock_ui' creates the game instance
    game, _ = game_with_mock_ui

    # Assert that the game_mode attribute is set to "Simple" by default
    assert game.game_mode == "Simple"


def test_changing_game_mode_to_general(game_with_mock_ui):
    """
    Given a game instance in the default "Simple" mode,
    When the general game mode radio button is selected,
    Then the game mode should be updated to "General".
    """
    game, mock_game_ui = game_with_mock_ui

    # 1. (Optional) Confirm the initial state is "Simple"
    assert game.game_mode == "Simple"

    # 2. Simulate the UI change: the user clicks the "General" radio button.
    #    In the UI, this would make simple_radio unchecked and general_radio checked.
    #    We simulate this by telling our mocks what to return when isChecked() is called.
    mock_game_ui.simple_radio.isChecked.return_value = False
    mock_game_ui.general_radio.isChecked.return_value = True

    # 3. Manually call the update_game_mode method.
    #    In the actual application, the 'toggled' signal from the radio button
    #    would trigger this method. In our test, we call it directly.
    game.update_game_mode()

    # 4. Assert that the game mode has been successfully changed
    assert game.game_mode == "General"

```