CMPT 276 Project - Final Phase

Group : EFT
Members: Joao, Justin, Anton, Van, Hanbo

Dependencies needed to run the game:
PyQt5

Code should be run in the phase 3 folder and around 1920x1080 Resolution
Updates compare with phase 2:

- New Game Interface: Ideal looking game interface with more functionalities
- Tutorial: Additional tutorial page for game rules
- Win Condition & Game Scoring: Correct game scores for players with win condition
- Achievements: More achievements were added
- Save and Load: Game could be saved & loaded
- Increased Speed of Game Interaction Feedback (Pressing button, AI, etc..)

Phase 3

Refactoring:

This is a critical process in software development that focuses on improving the structure, readability, and maintainability of code while preserving its functionality. It is essential for ensuring that your codebase remains adaptable, scalable, and easy to work with as your project evolves. When addressing specific code smells, such as duplicated code, magic numbers, and redundant patterns, employing appropriate refactoring methods is crucial to achieving these goals.
The selected refactoring methods, namely "Extract Method," "Replace Magic Number with Symbolic Constant," and "Introduce a Factory Class," offer distinct advantages over alternative approaches in addressing their respective code smells. These methods not only alleviate the immediate issues but also contribute to long-term code quality and efficiency. Below, we explore how each method is better suited for its corresponding case of code smells.

Smell 1: Dispensables — Repeated Creation of PyQt Buttons
Description: We observed repetitive blocks of code responsible for creating PyQt buttons with similar properties and styling, but mostly the same. This repetition can lead to code redundancy and create difficulties for the modularity and maintainability of our application. Further, the repetitive, clunky code is hard to read, which impedes testing or efficient change if necessary.

Impact: Redundant code for creating GUI elements, such as buttons, can result in inefficiencies and complications when making widespread changes to appearance or behavior. (shotgun surgery)

Refactoring Method: Extract Method and Introduce a Factory Class. In response to this issue, we opted to centralize the button creation process to enhance modularity and maintainability. By creating a dedicated class and function, we made it simpler, the creation of PyQt buttons with consistent styles and behaviors. This approach simplifies the process of modifying or updating button properties across the application. Our solution entailed developing the `button.createButton` function, which facilitates the creation of standardized buttons with specified styles.

Before: buttons were being created all around

```
# CREATE EXIT
exit_button = QPushButton('Main Menu', self)
exit_button.clicked.connect(self.on_exit_clicked)
exit_button.setStyleSheet(
    "QPushButton { border-radius: 25px; padding: 20px; font-size: 20px; border: 2px solid black; background-color: rgb(224, 166, 181);}")

# CREATE PASS
pass_button = QPushButton('Pass', self)
pass_button.clicked.connect(lambda: next_player_clicked(self.playerList, self.turn, self.boardLayout))
pass_button.setStyleSheet(
    "QPushButton { border-radius: 25px; padding: 20px; font-size: 20px; border: 2px solid black; background-color: rgb(224, 166, 181);}")

# CREATE CONFIRM
confirm_button = QPushButton('Confirm', self)
confirm_button.clicked.connect(lambda: confirm_placement(self.boardLayout, self.playerList, self.turn))
confirm_button.setStyleSheet(
    "QPushButton { border-radius: 25px; padding: 20px; font-size: 20px; border: 2px solid black; background-color: rgb(224, 166, 181);}")

# CREATE ROTATE
rotate_button = QPushButton('Rotate', self)
rotate_button.clicked.connect(self.rotate_piece)
rotate_button.setStyleSheet(
    "QPushButton { border-radius: 25px; padding: 20px; font-size: 20px; border: 2px solid black; background-color: rgb(224, 166, 181);}")
```

After: factory method for simple creation and styling of buttons

```
# CREATE EXIT
exit_button = button.createButton(("rgb(224, 166, 181)","rgb(244, 195, 209)", "rgb(202, 123, 139)"), (300, 60), "Exit", "25",parent=self)
exit_button.clicked.connect(self.on_exit_clicked)

# CREATE PASS
pass_button = button.createButton(("rgb(224, 166, 181)","rgb(244, 195, 209)", "rgb(202, 123, 139)"), (300, 60), "Pass", "25",parent=self)
pass_button.clicked.connect(lambda: next_player_clicked(self.playerList, self.turn, self.boardLayout, True, self))

# CREATE CONFIRM
confirm_button = button.createButton(("rgb(224, 166, 181)","rgb(244, 195, 209)", "rgb(202, 123, 139)"), (300, 60), "Confirm Placement","25", parent=self)
confirm_button.clicked.connect(lambda: confirm_placement(self.boardLayout, self.playerList, self.turn, self))

# CREATE ROTATE
rotate_button = button.createButton(("rgb(224, 166, 181)","rgb(244, 195, 209)", "rgb(202, 123, 139)"), (300, 60), "Rotate","25", parent=self)
rotate_button.clicked.connect(self.rotate_piece)
```

```python
def createButton(colours, size, text, borderRadius, parent=None):
    colour = colours[0]
    hoverColour = colours[1]
    pressColour = colours[2]
    button = QPushButton(parent)
    button.setText(text)
    button.setFixedSize(size[0],size[1])
    button.setStyleSheet(
        f"QPushButton {{"
        f"    font-size: 24px; padding: 10px;"
        f"    color: black; background-color: {colour};"
        f"    border: 3px solid black; border-radius: {borderRadius}px;"
        f"}}"
        f"QPushButton:hover {{"
        f"    background-color: {hoverColour};"
        f"}}"
        f"QPushButton:pressed {{"
        f"    background-color: {pressColour};"
        f"}}")

    return button
```

Smell 2: Dispensables — Duplicated Code for Going Back to Main Menu

Description: In our code, we observed the presence of redundant logic for returning to the main menu in multiple places. This duplication results in unnecessary code repetition and poses challenges for maintenance and updates.

Impact: The use of duplicated code can lead to inconsistencies, increased efforts in maintenance, and a higher likelihood of introducing errors when modifications are required. This code smells violates the DRY(Don't Repeat Yourself) principle, which makes the code harder to maintain and fix inconsistencies

Refactoring Method: Extract Method. To address this concern, we opted to create a single function or method that encapsulates returning to the main menu. This approach helps eliminate duplication by allowing us to call this function from various parts of the codebase. Our solution involved the creation of the `go_back_main_menu` function, which simplifies the process of hiding the current window and displaying the main menu window. This function encapsulates the navigation logic and calls whenever it is needed to go to the main menu.

Before: each new window had a creation of the go back process

```
    self.back_button.clicked.connect(self.go_back_main_menu())

    #Buttons formatting
    self.layout = QVBoxLayout(self)
    self.layout.setAlignment(Qt.AlignCenter)
    self.layout.addWidget(self.file1_button, alignment=Qt.AlignCenter)
    self.layout.addWidget(self.file2_button, alignment=Qt.AlignCenter)
    self.layout.addWidget(self.file3_button, alignment=Qt.AlignCenter)
    self.layout.addWidget(self.file4_button, alignment=Qt.AlignCenter)
    self.layout.addWidget(self.file5_button, alignment=Qt.AlignCenter)
    self.layout.addWidget(self.back_button, alignment=Qt.AlignCenter)

    Codeium: Refactor | Explain | Generate Docstring
    def go_back_main_menu(self):
        self.close()
        self.mainMenu = MainWindow(self.soundPlayer)
        self.mainMenu.show()
```

After: process was extracted and set to be a common factor when needed

```
self.back_button.clicked.connect(lambda: go_back_main_menu(self))
```

```
def go_back_main_menu(window):
    window.close()
    window.main_menu = MainWindow(window.soundPlayer)  # create an instance of your main menu
    window.main_menu.show()
```

Smell 3: Bloaters — Use of Magic Numbers

Description: While analyzing our code, we noticed the use of magic numbers, which are numeric constants lacking clear context or explanation, hard-coding numeric values. This practice can reduce code readability and hinder our understanding of the intent behind these numbers.

Impact: Relying on magic numbers can hinder code maintainability and comprehension. Without proper context, these numbers may lead to confusion or errors during future changes. Their meaning is not clearly represented and it hardens the purpose of the code and possible bug walkthroughs.

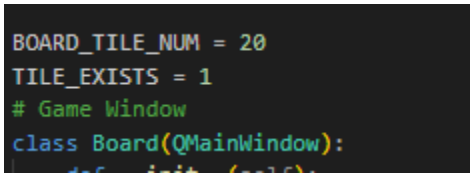
Refactoring Method: Symbolic Constant. To enhance our code's clarity and maintainability, we decided to replace these magic numbers with named constants or variables that provide meaningful context. Our approach involved creating constants to represent these magic numbers, thereby making the code more self-documenting and transparent.

Before: too many occurrences of isolated numbers with no explanation

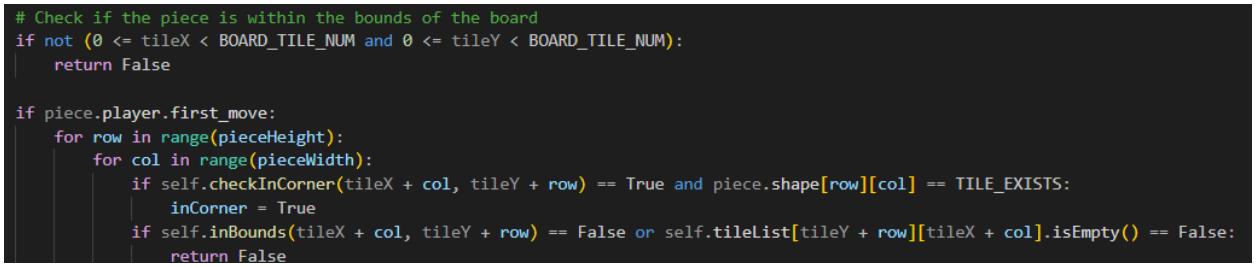
```
# Check if the piece is within the bounds of the board
if not (0 <= tileX < 20 and 0 <= tileY < 20):
    return False

if piece.player.first_move:
    for row in range(pieceHeight):
        for col in range(pieceWidth):
            if self.checkInCorner(tileX + col, tileY + row) == True and piece.shape[row][col] == 1:
                inCorner = True
            if self.inBounds(tileX + col, tileY + row) == False or self.tileList[tileY + row][tileX + col].isEmpty() == False:
                return False
```

After: names are self-explanatory and can be easily assigned



```
BOARD_TILE_NUM = 20
TILE_EXISTS = 1
# Game Window
class Board(QMainWindow):
```



```
# Check if the piece is within the bounds of the board
if not (0 <= tileX < BOARD_TILE_NUM and 0 <= tileY < BOARD_TILE_NUM):
    return False

if piece.player.first_move:
    for row in range(pieceHeight):
        for col in range(pieceWidth):
            if self.checkInCorner(tileX + col, tileY + row) == True and piece.shape[row][col] == TILE_EXISTS:
                inCorner = True
            if self.inBounds(tileX + col, tileY + row) == False or self.tileList[tileY + row][tileX + col].isEmpty() == False:
                return False
```

In conclusion, through these collaborative efforts, we've successfully addressed these code smells, leading to a more maintainable, readable, and modular codebase. These refactoring strategies align with industry best practices, contributing to the robustness and manageability of our code.