CMPT276 Phase 3: Refactoring

Warthog Group

Code Smell #1: Magic Numbers

Before refactoring, our code contained a lot of “magic numbers” whose meaning was not very clear. For example, here is some of our old code:

```
SCREEN = pygame.display.set_mode((1300, 800))
```

```
Opponent_board = Board(8, 8, (150, 150), 550, False)
```

Also, we drew objects using hexadecimal colours. For example,

```
pygame.draw.rect(screen, "#59A2E1", cell, 2)
```

```
elif self.is_hit and self.ship == None:
    pygame.draw.rect(screen, "#DAE159", cell)
```

It is not very obvious what (1300, 800) represents when setting up the screen, or what the parameters represent when constructing a Board object. To fix this, we replaced these “magic numbers” with declared constants:

```
SCREEN_SIZE = (1300, 800)
SCREEN = pygame.display.set_mode(SCREEN_SIZE)
```

```
self.board = Board(
            size=game_size,
            num_ships=ship_count,
            coords=coords,
            width=width,
            display=display,
            foreign=foreign
)
```

This had the bonus effect of allowing us to customize our game by allowing for games with different board sizes and number of ships, since these parameters were now variable rather than hard coded.

For the colours, we enumerated each hexadecimal colour using a Colours class:

```
class Colours(Enum):
    NAVY_BLUE = "#042574"
    GOLD = "#b68f40"
    WHITE = "#ffffff"
    RED = "#ff0000"
    DARK_RED = "#8B0000"
    YELLOW = "#ffff00"
```

This allowed us to draw things using the name of the colour, rather than the hexadecimal literal. For example,

```
elif not self.is_guessed:
    pygame.draw.rect(screen, Colours.GOLD.value, cell, 2)
```

Code Smell #2: Shotgun Surgery

In earlier revisions of our project, we drew our UI screens in many different places. The file game.py included the logic for the main menu screen, ship placement screen, and gameplay screen, while the GameManager class was responsible for the end game screen. We then added more and more screens, such as a settings screen, an opponent selector screen, etc. These screens were drawn all over the place; some were in game.py, and some were in GameManager. This became really confusing and difficult to locate where the code for a specific screen was. Furthermore, if we wanted to add a screen, we had to edit the code for the many different screens that routed to and from that screen, and implement the routing logic there. This is an example of Shotgun Surgery, where one change requires a change to be made in many separate places.

To fix this, we implemented each screen as a separate class, and all screens inherited from a Screen abstract base class. This allowed us to easily locate where the code for each screen was located.

We then made a Router class, which contained a stack of screens the user had visited. This made routing between screens to be much easier, as we could simply push the next screen onto the routing stack and go back to the previous screen by popping from the stack. We no longer need to change many different screens to add a single screen.

Code Smell #3: Bloater Method

In previous versions of our code, this is what the constructor looked like for the Board class:

```
def __init__(self, size, num_ships, coords, width, display, foreign=False):
    self.__nships = num_ships
    self.__size = size
    self.__coordinates = coords
    self.__width = width
    self.__display = display
    self.__foreign = foreign

    # Create the ships
    self.__ships = []
    for i in range(self.__nships):
        if i < 3:
            # append a 1x1 ship
            ship = NormalShip(1)
            self.__ships.append(ship)
        elif i >= 3 and i < 5:
            ship = NormalShip(2)
            self.__ships.append(ship)
        elif i >= 5 and i < 7:
            ship = NormalShip(3)
            self.__ships.append(ship)
        elif i >= 7 and i < 9:
            ship = NormalShip(4)
            self.__ships.append(ship)
        else:
            print("attempting to add an invalid ship")
    # Create the cells
    cell_size = self.__width / self.__size
    x_0 = self.__coords[0]
    y_0 = self.__coords[1]
    self.__cells = []
    for x in range(self.__size):
        row = []
        for y in range(self.__size):
            location_x = x_0 + x * cell_size
            location_y = y_0 + y * cell_size
            row.append(
                Cell(
                    coords=(x, y),
                    width=cell_size,
                    location=(location_x, location_y),
                    foreign=foreign,
                )
            )
        self.__cells.append(row)
```

This constructor is humungous and very hard to understand. To fix this, we used the abstract factory design pattern and created a BoardFactory class, which was solely responsible for creating the Ships and the Cells.

Now, the Board constructor looks like this:

```
def __init__(self, size, num_ships, coords, width, display,
foreign=False):
    self.__nships = num_ships
    self.__size = size

    self.__coordinates = coords
    self.__width = width
    self.__display = display
    self.__foreign = foreign

    self.__board_factory = BoardFactory(
        self.__size, self.__nships, self.__coordinates, self.__width
    )
```

In addition, we made a method that allows the Board to build its ships and cells:

```
def build_board(self):
    self.__cells = self.__board_factory.create_cells(self.__foreign)
    self.__ships = self.__board_factory.create_ships()
```

This improves upon the Single Responsibility Principle; each method now is responsible for one thing only and we don’t have one gigantic constructor.