Lazy class & Magic number

Code Smell Issue:
One code refactoring we performed was changing the Corner class from a lazy class to a meaningful class. Because we are using a tile class to include all the board spaces, the spaces at four corners are also required to draw on the board. However, unlike other types of tile, the four corners work differently depending on what corner you are in. Since the jail function is quite complicated, we put functions for handling each of these four corners in the GUI file. However, this left the corner class empty, and it became a lazy class. In addition, we only used the position number to define these four corner spaces in the GUI file and this created some magic numbers in our code.

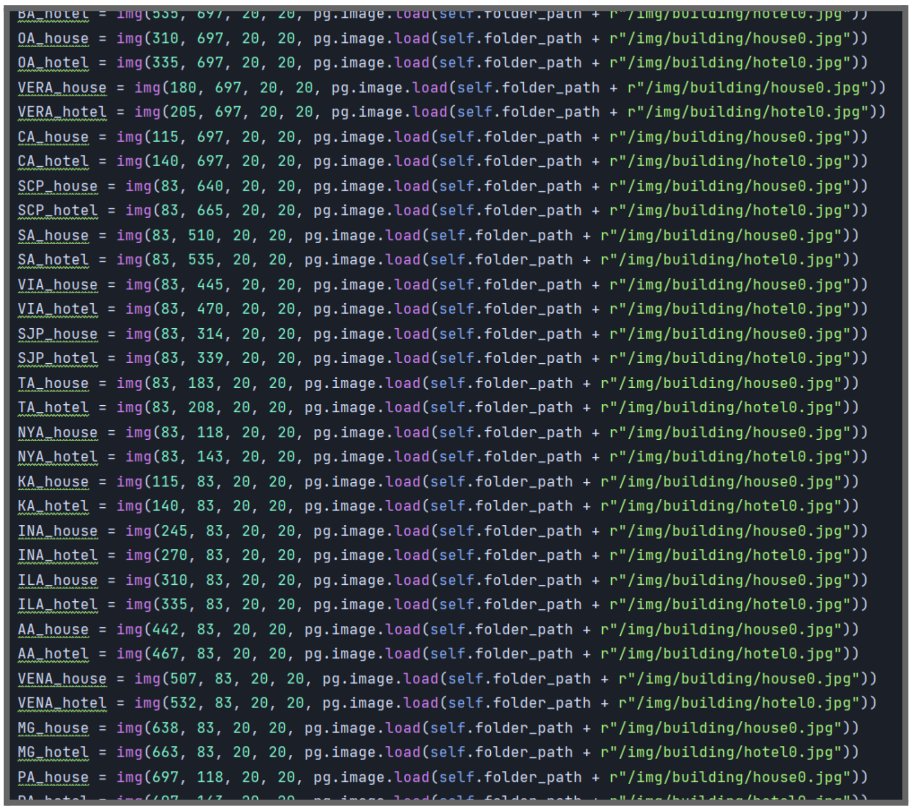
Chosen Technique:
To fix these issues, we used Composing methods that let the Corner class handle the definition of these four corner spaces. This adhered to the single responsibility principle, and the concept of encapsulation. The functions in the GUI file now will call from the Corner class to identify what corner it is, and perform correct actions. The reason we choose this method is that this method can fix the two issues we had at once, and provides more functionality and modularity that is described below.

How it Improved The Code:
By using this method, we not only let the Corner class become meaningful, but also let it handle the right amount of responsibility. Especially since we cannot delete the Corner class because it is part of the board. In addition, by making class variables that show the name of the corner rather than the position number of the corner, this method helps reduce the confusion in our code by reducing the amount of magic numbers. The corner tiles are now able to be used in any context where the player lands on a corner tile. Since the tiles themselves will determine what corner it is (Go, Jail, Free Parking, Go To Jail), the code becomes more modular as we no longer need to explicitly check the players or tiles position in relation to the board. We can simply get the attributes of the tile, and determine what tile type it is, and handle the functionality from there. Furthermore, the code becomes more readable, as the variable names are descriptive. The code has become more extendable, in the sense that if we wanted to expand the board, or add more functionality to the corner tiles, we are able to do so easily. In conclusion, by aligning the class's responsibilities with its purpose and encapsulating corner-specific behavior, the code has become more intuitive, organized, and prepared for future development.

Change Preventer: Divergent Change

Code Smell Issue:

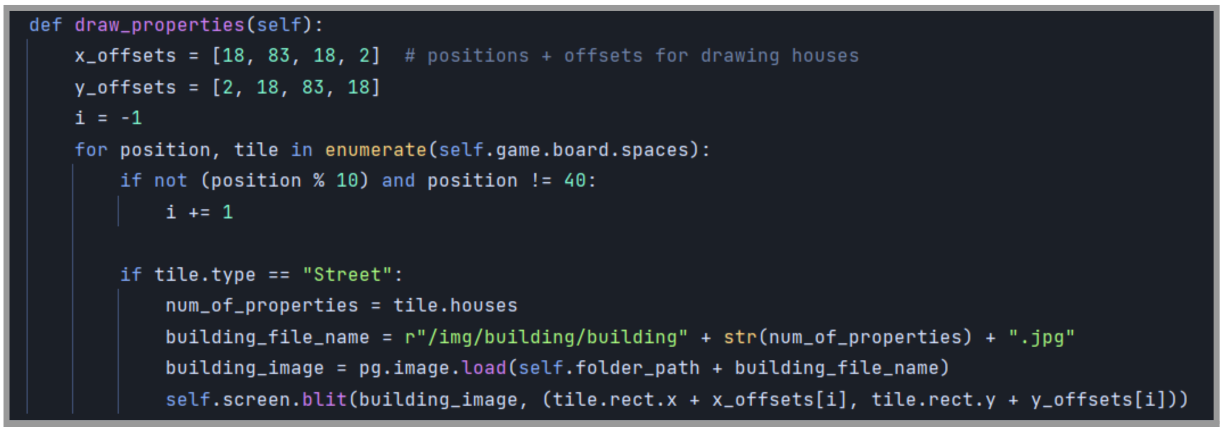
One code refactorization that was performed was changing the way the building images were drawn onto the board. Previously, an object of class img was created for each building. This img class had attributes such as the dimensions and coordinates of the image, pygame image and pygame rect attributes, and various other types of metadata. This class worked well for some cases (such as for the dice, or the title deed images), but was not ideal for drawing the building images. Since each building image was hardcoded, if we wanted to change the position of the image on each tile, we would need to change every line of code that specified the coordinates of the image. Furthermore, the img class was made for general image placement, and did not hold any attributes related to what tile each building image belonged to. Therefore, if a player were to buy or sell buildings on a property, the only way to update the image would be with multiple conditional statements for each tile. Lastly, the repetitive code resulted in increased memory usage, as each image had its own metadata. A screenshot of a portion of the original code is shown below.



```
BA_hotel = img(335, 697, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
OA_house = img(310, 697, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
OA_hotel = img(335, 697, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
VERA_house = img(180, 697, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
VERA_hotel = img(205, 697, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
CA_house = img(115, 697, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
CA_hotel = img(140, 697, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
SCP_house = img(83, 640, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
SCP_hotel = img(83, 665, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
SA_house = img(83, 510, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
SA_hotel = img(83, 535, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
VIA_house = img(83, 445, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
VIA_hotel = img(83, 470, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
SJP_house = img(83, 314, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
SJP_hotel = img(83, 339, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
TA_house = img(83, 183, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
TA_hotel = img(83, 208, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
NYA_house = img(83, 118, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
NYA_hotel = img(83, 143, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
KA_house = img(115, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
KA_hotel = img(140, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
INA_house = img(245, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
INA_hotel = img(270, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
ILA_house = img(310, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
ILA_hotel = img(335, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
AA_house = img(442, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
AA_hotel = img(467, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
VENA_house = img(507, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
VENA_hotel = img(532, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
MG_house = img(638, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
MG_hotel = img(663, 83, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
PA_house = img(697, 118, 20, 20, pg.image.load(self.folder_path + r"/img/building/house0.jpg"))
PA_hotel = img(697, 143, 20, 20, pg.image.load(self.folder_path + r"/img/building/hotel0.jpg"))
```

Chosen Technique:

One method of refactoring mentioned in class for this case would be to export the relevant data into a file, read the relevant data into each instance of the img class, and have each instance deal with drawing based on its own data. We chose to do something similar, but with various optimizations by utilizing the properties of the tiles themselves. The tile class actually had all the necessary information we needed for drawing the correct building images into the correct places. Each tile knew the number of houses it owned, and it had a pygame rect object that we could use to draw the property image onto. Therefore, the only values we needed to know outside of the existing tile class data, was the pixel offset of the location to draw each building image. The refactored code is shown below:



```python
def draw_properties(self):
    x_offsets = [18, 83, 18, 2]  # positions + offsets for drawing houses
    y_offsets = [2, 18, 83, 18]
    i = -1
    for position, tile in enumerate(self.game.board.spaces):
        if not (position % 10) and position != 40:
            i += 1

        if tile.type == "Street":
            num_of_properties = tile.houses
            building_file_name = r"/img/building/building" + str(num_of_properties) + ".jpg"
            building_image = pg.image.load(self.folder_path + building_file_name)
            self.screen.blit(building_image, (tile.rect.x + x_offsets[i], tile.rect.y + y_offsets[i]))
```

By using the properties of the tiles themselves, we're able to draw the correct image to the screen without needing to explicitly update the image when a player buys/sells a house. This technique also has the benefit of using less memory, because entire classes are not made for the images, rather they are read from a file and put on the screen. One thing to note is that the refactoring technique described in class was actually used here, but only for creating the tiles. The relevant data is read from a file into each instance of the tile class on game startup.

<u>How it Improved The Code:</u>
With the new function to draw the buildings onto the screen, the code became much more adaptable to changes without extensive modification. In fact, this adaptability was proven when we decided to modify the UI so that it only showed a single image for houses (rather than separate images for houses and buildings). The only changes we had to make for drawing houses was to the draw_properties function, rather than to the ~45 lines of code we had previously. This refactorization also agrees with the DRY principle. Not only do we no longer need to repetitively create img objects for each building, we also don't need to call the draw function for each image every time we want to draw the houses. Instead, this function is called in multiple places in the GUI class, with no need for repetition. The function also provides a better separation of concerns. The tile class is focused on its properties while the image-drawing logic is encapsulated in a dedicated function. Lastly, this function is scalable and extendable. If we wanted to add more tiles to the board, or change the number of possible houses, this function would require minimal change, and would be able to support more complex interactions and features.

Duplicate Code

Code Smell Issue:
Before the refactor took place there were separate classes for the collections and iterators for each of Chance and Community Chest. They shared a large amount of the same code with few differentiations that made them unique. One of the few distinctions was the jail card number attribute and different functions to get card operations. This code engineering caused the duplicate code, code smell and we had to put in a lot more work to maintain the codebase. It also was the cause of issues in testing as we would often change one class to fix bugs in it, then run the test, only for it to fail when running the other collections/iterator class. It was exceedingly exasperating as the testing was manual and would take a few minutes to correctly verify the functionality.

Chosen Technique:
The code was fixed by merging the two implementations of iterators and collections into one of each. Although the new classes are more general, it could be seen as more clear as the magic numbers in them before are replaced by variables that you have to initialize to create the function. Though we could have pulled the common code into a common parent class, it would have caused the original implementations to become lazy as each class would build on the parent class by two lines for two functions. This way of refactoring also will be easier to read since the code is all in one place rather than split over multiple classes.

How it Improved The Code:
This refactoring change has several improvements to the codebase. The codebase has become easier to maintain as the duplicated code was merged into a single class, allowing additions and modifications to be centralized. This means that the card functionality is now more scalable, and extendible. After the refactorization, the errors that occurred relating to drawing cards were easier to find and fix as there is only one class of each, and the discrepancies between the classes no longer existed. The testability of the code is also easier since now we only have to test 2 classes instead of 4. The overall architecture of the card classes has become clearer, and encapsulated within a singular unit. The readability of the classes is also better than before as the whole cards section before was chaotic since it had to include at least 4 more classes (2 for interface and 2 for duplication). In conclusion, the refactoring effort to merge the Chance and Community Chest implementations has significantly improved the codebase by eliminating code duplication, resolving code smells, enhancing maintainability, improving testing efficiency, and promoting readability.