Max Ho: Object-oriented abusers

Old code:

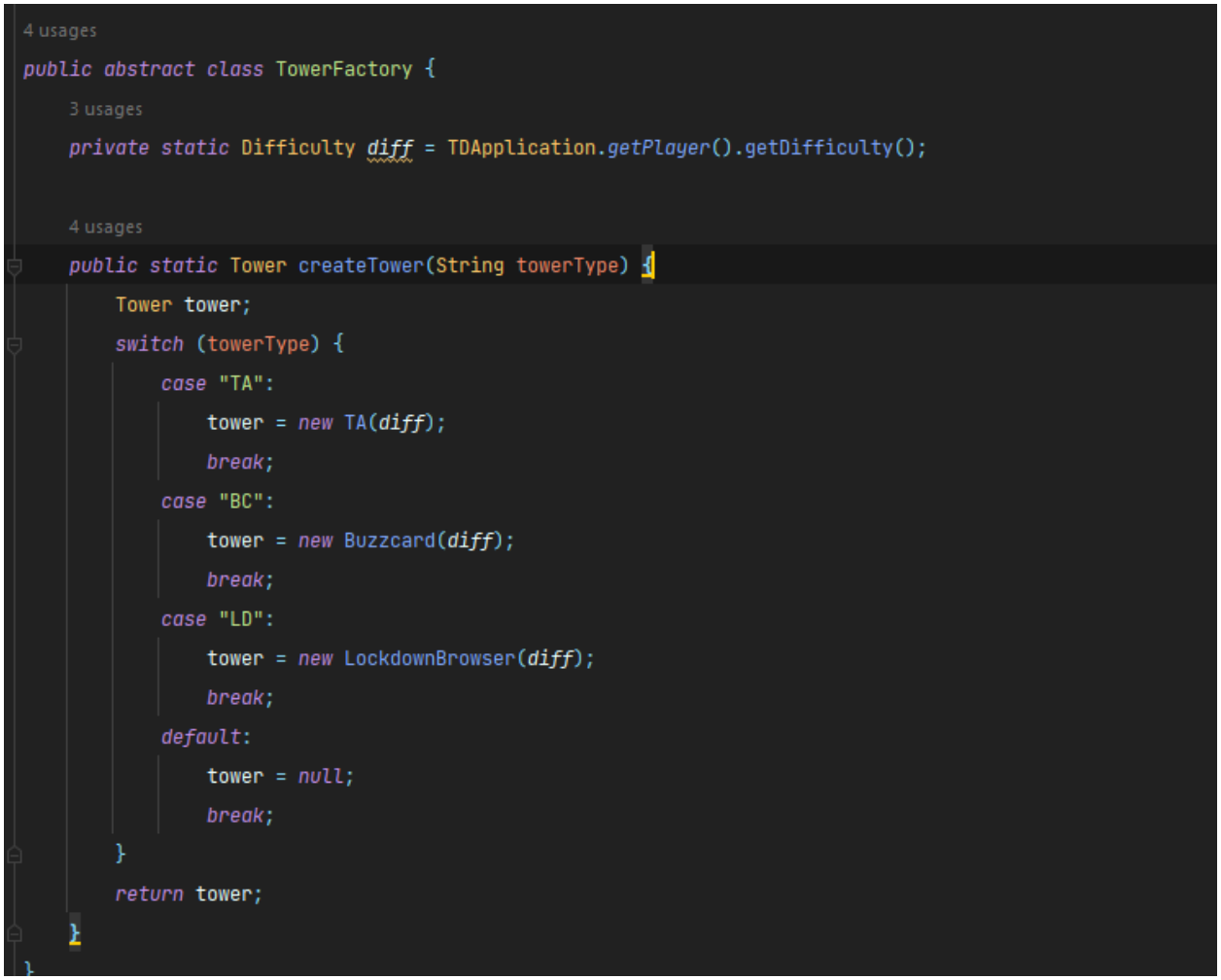
```
22      public void selectTower(ActionEvent event) {
23          String towerObject = event.getSource().toString().substring(10, 12);
24          tower = null;
25          switch (towerObject) {
26          case "TA":
27              tower = new TA(TDApplication.getPlayer().getDifficulty());
28              break;
29          case "BC":
30              tower = new Buzzcard(TDApplication.getPlayer().getDifficulty());
31              break;
32          case "LD":
33              tower = new LockdownBrowser(TDApplication.getPlayer().getDifficulty());
34              break;
35          default:
36              tower = new LockdownBrowser(Difficulty.CHEAT_SHEET);
37          }
38          description.setText(tower.getDescription());
39          description.getText();
40      }
```

Figure 1. selectTower() using an accursed switch statement to make a tower.

Switch statements are VERY bad. In our current code, our every input creates a tower in a series of switch statements depending on the tower class given. This code smell exists because in previous milestones to achieve functionality, we used switch statements to change between objects instead of finding a way to make this modular. This should be resolved using a factory design pattern, as now there are lots of repetitions in our design.

```
22  @       public void selectTower(ActionEvent event) {
23              String towerObject = event.getSource().toString().substring(10, 12);
24              tower = TowerFactory.createTower( towerType: towerObject);
25              description.setText(tower.getDescription());
26              description.getText();
27          }
```

Figure 2. New selectTower() implementing TowerFactory.



```
4 usages
public abstract class TowerFactory {
    3 usages
    private static Difficulty diff = TDApplication.getPlayer().getDifficulty();

    4 usages
    public static Tower createTower(String towerType) {
        Tower tower;
        switch (towerType) {
            case "TA":
                tower = new TA(diff);
                break;
            case "BC":
                tower = new Buzzcard(diff);
                break;
            case "LD":
                tower = new LockdownBrowser(diff);
                break;
            default:
                tower = null;
                break;
        }
        return tower;
    }
}
```

Figure 3. TowerFactory class with static createTower() method.

Using a factory design pattern, we have fixed overuse of switch statements, which uses the difficulty too much to construct our objects. With TowerFactory, now creating different towers is much more streamlined and will be able to even be expanded on in the future without causing shotgun surgery.

Ethan Tong: Long Parameter Lists

All the individual enemy subclasses (SpiralNotebook, Textbook, GroupMe, KhanAcademy, StackOverflow, YouTube, Chegg) have constructors that take in four parameters, which are considered long parameter lists under the bloater code smell category. The width and height parameters do not need to exist because they are intended to remain constant throughout the whole application. Similarly, the difficulty parameter does not need to exist because difficulty should be consistent throughout gameplay.

```
public SpiralNotebook(int height, int width, Difficulty diff, EnemyPath enemyPath) {

    this.setImage(new Image(getClass()
            .getClassLoader().getResourceAsStream( name: "spiralNotebookEnemy.png")));
    this.setFitHeight(height);
    this.setFitWidth(width);

    if (diff == Difficulty.CLOSED_NOTES) {
        monumentDamage = 0.1;
        health = 10;
    } else if (diff == Difficulty.CHEAT_SHEET) {
        monumentDamage = 0.12;
        health = 15;
    } else {
        monumentDamage = 0.14;
        health = 20;
    } // if

    this.updatePosition();
    this.distance = 0;

    this.enemyPath = enemyPath;
    this.tracker = new EnemyPathIterator(this.enemyPath);
    this.target = null;
    this.moveVector = null;
    this.moveSpeed = 6.0;

} // SpiralNotebook
```

Example: SpiralNotebook enemy

The changes made to the parameter list were relatively simple. Width, height, and difficulty were moved outside the constructor to become instance variables. Difficulty is instantiated inside the constructor and width and height are instantiated outside the constructor as constant values.

```
Difficulty diff;
final int width = 50;
final int height = 50;

public SpiralNotebook(Vector2[] enemyPath) {

    diff = TDApplication.getPlayer().getDifficulty();
    this.setImage(new Image(getClass()
            .getClassLoader().getResourceAsStream( name: "spiralNotebookEnemy.png")));
    this.setFitHeight(height);
    this.setFitWidth(width);

    if (diff == Difficulty.CLOSED_NOTES) {
        monumentDamage = 0.1;
        health = 10;
    } else if (diff == Difficulty.CHEAT_SHEET) {
        monumentDamage = 0.12;
        health = 15;
    } else {
        monumentDamage = 0.14;
        health = 20;
    } // if

    this.updatePosition();
    this.distance = 0;

    this.enemyPath = enemyPath;
    this.tracker = new EnemyPathIterator(this.enemyPath);
    this.target = null;
    this.moveVector = null;
    this.moveSpeed = 6.0;

} // SpiralNotebook
```

Example: SpiralNotebook enemy

Luke Walter: Data Class

The EnemyPath class was designed to contain an array of coordinates that enemies can travel towards and iterate through as they move towards the monument. However, the class itself only includes the array itself, a method to calculate size, and a getter for the array. This information is already included in arrays themselves. Originally, more functionality was meant to be implemented in this class, but it was delegated to EnemyPathIterator. So, there is no reason for this class to exist when a Vector2 array has the same functionality.

```
1   package com.closednotes.proctr.objects;
2
3   public class EnemyPath {
4
        3 usages
5       Vector2[] pathNodes;
6
7       public EnemyPath(Vector2[] pathNodes) {
8           this.pathNodes = pathNodes;
9
10      } // Constructor
11
12      public int size() {
13          return pathNodes.length;
14
15      } // size
16
        1 usage
17      public Vector2[] toArray() {
18          return pathNodes;
19
20      } // toArray
21
22  } // EnemyPath
```

EnemyPath (deleted class)

```
17        */
18       public static EnemyWrapper createEnemy(char variant, EnemyPath enemyPath) {
19
```

Code example before refactor

```
17        */
18       public static EnemyWrapper createEnemy(char variant, Vector2[] enemyPath) {
19
```

Code example after refactor (no functional change to code outside of data type)

Kewal Kalsi: Long Method Code Smell

```
public void attackEnemy(EnemyWrapper enemy) {
    InputStream animation = null;
    switch(this.getTower().getClass().getSimpleName()) {
        case("TA"):
            animation = TDApplication.class.getClassLoader()
                    .getResourceAsStream( name: "TA_Projectile.png");
            break;
        case("Buzzcard"):
            animation = TDApplication.class.getClassLoader()
                    .getResourceAsStream( name: "Buzzcard_Swipe.png");
            break;
        case("LockdownBrowser"):
            animation = TDApplication.class.getClassLoader()
                    .getResourceAsStream( name: "Lockdown_Projectile.png");
            break;
        default:
            break;
    }
    final ImageView image = new ImageView(new Image(animation, v: 10, v1: 20,  b: true,  b1: true));
    TranslateTransition t = new TranslateTransition(Duration.seconds(0.5), image);
    t.setFromX(this.getX() + this.getFitWidth()/2);
    t.setFromY(this.getY() + this.getFitHeight()/2);
    t.setToX(enemy.getGameObject().getX() + enemy.getGameObject().getFitWidth()/2);
    t.setToY(enemy.getGameObject().getY() + enemy.getGameObject().getFitHeight()/2);
    enemy.getAnchor().getChildren().add(image);
    t.setOnFinished((finish) -> enemy.getAnchor().getChildren().remove(image));
    t.playFromStart();
    enemy.getGameObject().loseHealth(this.getTower().getDamage());
    if (enemy.getGameObject().getHealth() < 0) {
        enemy.deactivate();
        TDApplication.getPlayer().earnMoney(5);
        GameController.updateMoney();
    }
```

Long methods can make code convoluted and difficult to understand. The attack enemy method above has too many things happening in it. First, it determines the animation image to run, then creates the transition translation animation to run. Finally, it plays the animation, attacks the enemy, and forces the game to check if the enemy has died or not. This is too much happening in one method. In order to fix this long method, I can create private helper methods that do these intermediate steps and have this method simply call those methods. After creating helper methods to lower the size of the method, the new method looks like:

```
public void attackEnemy(EnemyWrapper enemy) {
    InputStream animation = determineAnimation(enemy);
    createTransition(animation, enemy);
    enemy.getGameObject().loseHealth(this.getTower().getDamage());
    if (enemy.getGameObject().getHealth() < 0) {
        enemy.deactivate();
        TDApplication.getPlayer().earnMoney(5);
        GameController.updateMoney();
    }
}
```

This code is much easier to read. You can tell that the method is creating the animation, then the transition, then attacking the enemy, and then finally checking it. This code removes the code smell and is much better for the code overall.