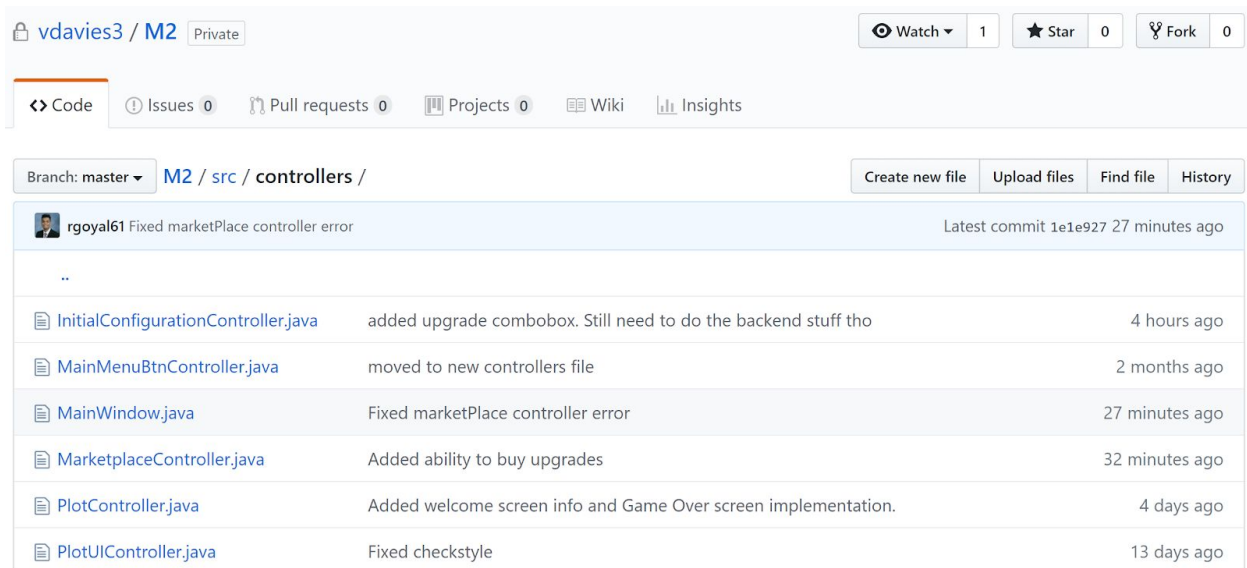


## GRASP & SOLID Principles in Harvest Hustler / Extra Credit

### SOLID

#### 1. Common Reuse Principle

Controllers are used together because of the way certain use cases work together, such as inventory being controlled and filled in the MarketPlaceController and then items from the inventory being applied to the farm on the PlotController. Furthermore, most classes within each controller change information regarding the player which is held in the InitialConfigurationController. This is an example of the CRP because we choose to package these together because they are used together.



vdavies3 / M2 Private

Watch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights

Branch: master M2 / src / controllers /

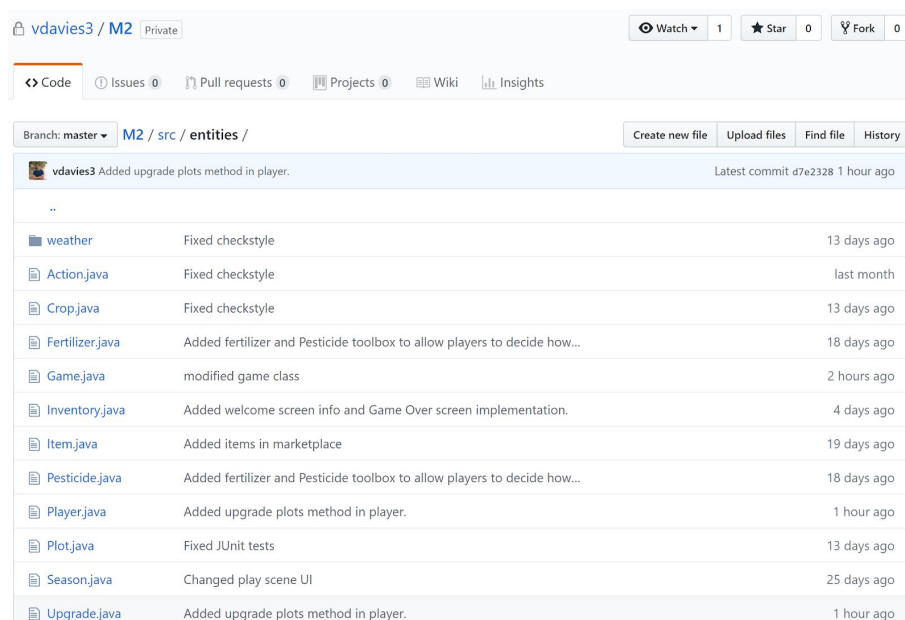
Create new file Upload files Find file History

rgoyal61 Fixed marketPlace controller error Latest commit 1e1e927 27 minutes ago

|                                     |  |                |
|-------------------------------------|--|----------------|
| InitialConfigurationController.java | added upgrade combobox. Still need to do the backend stuff tho | 4 hours ago    |
| MainMenuBtnController.java          | moved to new controllers file                                  | 2 months ago   |
| MainWindow.java                     | Fixed marketPlace controller error                             | 27 minutes ago |
| MarketplaceController.java          | Added ability to buy upgrades                                  | 32 minutes ago |
| PlotController.java                 | Added welcome screen info and Game Over screen implementation. | 4 days ago     |
| PlotUIController.java               | Fixed checkstyle   | 13 days ago    |

#### 2. Stable Abstractions Principle (SAP)

The entities package is abstract due to the way in which we choose to separate each item from the abstract class Item itself, which is very stable as it has few to no dependencies. Similarly the weather package within the entities package abstracts the differing seasons and weather occurrences available to players. Thus the most stable, yet also abstract within the entity package are the further weather package and also the Item abstraction.



vdavies3 / M2 Private

Watch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights

Branch: master M2 / src / entities /

Create new file Upload files Find file History

vdavies3 Added upgrade plots method in player. Latest commit d7e2328 1 hour ago

|                 |  |             |
|-----------------|--|-------------|
| weather         | Fixed checkstyle   | 13 days ago |
| Action.java     | Fixed checkstyle   | last month  |
| Crop.java       | Fixed checkstyle   | 13 days ago |
| Fertilizer.java | Added fertilizer and Pesticide toolbox to allow players to decide how... | 18 days ago |
| Game.java       | modified game class  | 2 hours ago |
| Inventory.java  | Added welcome screen info and Game Over screen implementation.           | 4 days ago  |
| Item.java       | Added items in marketplace   | 19 days ago |
| Pesticide.java  | Added fertilizer and Pesticide toolbox to allow players to decide how... | 18 days ago |
| Player.java     | Added upgrade plots method in player.                                    | 1 hour ago  |
| Plot.java       | Fixed JUnit tests  | 13 days ago |
| Season.java     | Changed play scene UI  | 25 days ago |
| Upgrade.java    | Added upgrade plots method in player.                                    | 1 hour ago  |

```
41 lines (31 sloc) | 767 Bytes
Raw Blame History

1 package entities;
2
3 public abstract class Item {
4
5     private int basePrice;
6     private String type;
7     private String color;
8
9     public Item(int basePrice, String type) {
10         this.basePrice = basePrice;
11         this.type = type;
12     }
13
14     public String color() {
15         return color;
16     }
17
18     public void setColor(String color) {
19         this.color = color;
20     }
21
22     public void setBasePrice(int basePrice) {
23         this.basePrice = basePrice;
24     }
25
26     public int getBasePrice() {
27         return this.basePrice;
28     }
29
30     @Override
31     public boolean equals(Object obj) {
32         Item o = (Item) obj;
33         return (this.type.equals(o.type));
34     }
35
36     public String getType() {
37         return this.type;
38     }
39
40     public abstract Item clone();
41 }
```

### 3. Stable-Dependencies Principle

In Harvest Hustler, a very important class is the `InitialConfigurationController`, which was previously touched on in example 1. This class depends on many variables within the Entities package which are unique to themselves and have no reliance on one another or classes in other packages. For this reason, dependency moves in the direction of increasing stability.

```

8  public class InitialConfigurationController {
9
10     private Crop initialCrop;
11     private Season initialSeason;
12     private String name;
13     private Player initialPlayer;
14     private Game game;
15     private List<Crop> allCrops = new ArrayList<>();
16     private List<Item> allItems = new ArrayList<>();
17     private List<Upgrade> allUpgrades = new ArrayList<>();
18
19     public InitialConfigurationController() {
20         this.name = "Player";
21         this.initialSeason = Season.FALL;
22         initialPlayer = new Player(0, name);
23         this.game = new Game(Season.FALL, initialPlayer, 1);
24
25         allCrops.add(new Crop("Corn", 100, 9, "Seed", "yellow"));
26         allCrops.add(new Crop("Wheat", 75, 6, "Seed", "goldenrod"));
27         allCrops.add(new Crop("Soybean", 50, 5, "Seed", "brown"));
28         allCrops.add(new Crop("Potatoes", 120, 10, "Seed", "peru"));
29         allCrops.add(new Crop("Carrots", 150, 12, "Seed", "orange"));
30         allCrops.add(new Crop("Green Beans", 90, 7, "Seed", "green"));
31
32         allItems.add(new Fertilizer());
33         allItems.add(new Pesticide());
34
35         allUpgrades.add(new Upgrade(500, "Tractor"));
36         allUpgrades.add(new Upgrade(400, "Irrigation System"));
37         allUpgrades.add(new Upgrade(1000, "+8 plots"));
38     }
39
40     public void nameHandler(String text) {
41         name = text;
42     }
43
44     public void difficultyHandler1() {
45         game.setDifficulty(1);
46     }
47
48     public void difficultyHandler2() {
49         game.setDifficulty(2);
50     }
51
52     public void difficultyHandler3() {
53         game.setDifficulty(3);
54     }
55
56     public void cropHandler(Crop crop) {
57         this.initialCrop = crop;
58     }
59
60     public void seasonHandler(Season season) {
61         this.initialSeason = season;
62     }

```

vdavies3 / M2
Private
Watch 1
Star 0
Fork 0

Code
Issues 0
Pull requests 0
Projects 0
Wiki
Insights

Branch: master
M2 / src / entities /
Create new file
Upload files
Find file
History

vdavies3 Added upgrade plots method in player.
Latest commit d7e2328 2 hours ago

|                 |  |             |
|-----------------|--|-------------|
| ..              |  |             |
| weather         | Fixed checkstyle   | 13 days ago |
| Action.java     | Fixed checkstyle   | last month  |
| Crop.java       | Fixed checkstyle   | 13 days ago |
| Fertilizer.java | Added fertilizer and Pesticide toolbox to allow players to decide how... | 18 days ago |
| Game.java       | modified game class  | 2 hours ago |
| Inventory.java  | Added welcome screen info and Game Over screen implementation.           | 4 days ago  |
| Item.java       | Added items in marketplace   | 19 days ago |
| Pesticide.java  | Added fertilizer and Pesticide toolbox to allow players to decide how... | 18 days ago |
| Player.java     | Added upgrade plots method in player.                                    | 2 hours ago |
| Plot.java       | Fixed JUnit tests  | 13 days ago |
| Season.java     | Changed play scene UI  | 25 days ago |
| Upgrade.java    | Added upgrade plots method in player.                                    | 2 hours ago |

## GRASP

### 1. Low Coupling

Low coupling allowed our project to be more clear and straightforward. This can specifically be said for weather, some weather events such as drought and rain could have potentially been coupled together in one class because of their similarity and effect on plant water levels. However, their application varied too much such that creating two separate classes for them was justified through low coupling principles.

Branch: master
M2 / src / entities / weather /
Create new file
Upload files
Find file
History

vdavies3 Fixed checkstyle
Latest commit b285761 13 days ago

|                   |                  |             |
|-------------------|------------------|-------------|
| ..                |                  |             |
| Clouds.java       | Fixed checkstyle | 13 days ago |
| Drought.java      | Fixed checkstyle | 13 days ago |
| Rain.java         | Fixed checkstyle | 13 days ago |
| SnowStorm.java    | Fixed checkstyle | 13 days ago |
| WeatherEvent.java | Fixed checkstyle | 13 days ago |

## 2. Protected Variations

The entities package holds many different similar classes in order to allow for more changes and lower coupling. An example of protected variations in the entity package is the separation of seasons from the weather package. While our group considered not separating the two due to various similarities, however the actions of the classes and versions of weather and weather events are too different from the implications of seasons in which trying to combine the two would only complicate things. (left)

```
1 package entities;
2
3 import entities.weather.*;
4
5 public enum Season {
6     FALL("Fall", new Clouds(), "Orange"),
7     WINTER("Winter", new SnowStorm(), "White"),
8     SPRING("Spring", new Rain(), "LightBlue"),
9     SUMMER("Summer", new Drought(), "yellow");
10
11     private final String name;
12     private final WeatherEvent weather;
13     private final String color;
14
15     Season(String name, WeatherEvent weather, String color) {
16         this.name = name;
17         this.weather = weather;
18         this.color = color;
19     }
20
21     public String getName() {
22         return this.name;
23     }
24
25     public String getColor() {
26         return this.color;
27     }
28
29     public WeatherEvent getWeather() {
30         return weather;
31     }
32 }
```

## 3. Polymorphism

Polymorphism plays multiple very important roles in Harvest Hustler because it allows us to increase the applicability to code as the requirements for each milestone continuously grow and change. An example of this is through the abstract class Item which multiple different classes extend in order to be placed in the marketplace at a price determined by algorithms within Item depending on difficulty and importance/multitude. This allowed us to add new items to the marketplace later on such as pesticides without having to completely rewrite code for the entirety of the marketplace. (below)

```
1 package entities;
2
3 public class Fertilizer extends Item {
4     public Fertilizer() {
5         super(50, "Fertilizer");
6         setColor("Brown");
7     }
8
9     @Override
10    public Item clone() {
11        return new Fertilizer();
12    }
13 }
```

## 4. Creator

As aforementioned, the InitialConfigurationController is of high importance because of the information it contains and holds from other classes. It initializes the game creating a name, name handler, crop handler, difficulty handler, etc. as well as creating a basis of

choices should the player choose not to decide such as Fall for the season. The game would not be able to be formed with the creator class of InitialConfigurationController.

```

8  public class InitialConfigurationController {
9
10     private Crop initialCrop;
11     private Season initialSeason;
12     private String name;
13     private Player initialPlayer;
14     private Game game;
15     private List<Crop> allCrops = new ArrayList<>();
16     private List<Item> allItems = new ArrayList<>();
17     private List<Upgrade> allUpgrades = new ArrayList<>();
18
19     public InitialConfigurationController() {
20         this.name = "Player";
21         this.initialSeason = Season.FALL;
22         initialPlayer = new Player(0, name);
23         this.game = new Game(Season.FALL, initialPlayer, 1);
24
25         allCrops.add(new Crop("Corn", 100, 9, "Seed", "yellow"));
26         allCrops.add(new Crop("Wheat", 75, 6, "Seed", "goldenrod"));
27         allCrops.add(new Crop("Soybean", 50, 5, "Seed", "brown"));
28         allCrops.add(new Crop("Potatoes", 120, 10, "Seed", "peru"));
29         allCrops.add(new Crop("Carrots", 150, 12, "Seed", "orange"));
30         allCrops.add(new Crop("Green Beans", 90, 7, "Seed", "green"));
31
32         allItems.add(new Fertilizer());
33         allItems.add(new Pesticide());
34
35         allUpgrades.add(new Upgrade(500, "Tractor"));
36         allUpgrades.add(new Upgrade(400, "Irrigation System"));
37         allUpgrades.add(new Upgrade(1000, "+8 plots"));
38     }
39
40     public void nameHandler(String text) {
41         name = text;
42     }
43
44     public void difficultyHandler1() {
45         game.setDifficulty(1);
46     }
47
48     public void difficultyHandler2() {
49         game.setDifficulty(2);
50     }
51
52     public void difficultyHandler3() {
53         game.setDifficulty(3);
54     }
55
56     public void cropHandler(Crop crop) {
57         this.initialCrop = crop;
58     }
59
60     public void seasonHandler(Season season) {
61         this.initialSeason = season;
62     }

```

## 5. Pure Fabrication

While upgrading objects could have been done in each items specific class, fabricating a specific upgrades class was the better choice. This is because it allowed for high cohesion, low coupling, and especially reuse as we added more and more items that had

the potential to be upgraded as well. Controlling this change in one class as opposed to various different classes simplified the code.

```
1  package entities;
2
3  public enum Upgrades {
4      TRACTOR("Tractor", 500),
5      IRRIGATION("Irrigation", 400),
6      PLOTS("+8 plots", 1000);
7
8      private final int basePrice;
9      private final String name;
10
11     Upgrades(String name, int basePrice) {
12         this.basePrice = basePrice;
13         this.name = name;
14     }
15
16     public String getName() {
17         return this.name;
18     }
19
20     public int getBasePrice() { return this.basePrice; }
21
22     @Override
23     public String toString() {
24         return name + " " + "$" + basePrice;
25     }
26
27 }
```

#### Extra Credit Project Idea:

Our idea for a new project will be based on an airport. The project will highlight the main processes that take place in an airport, starting from when someone enters the terminal to when they take off in a plane. This project will also pay attention to specifics as well such as charging



the passenger for check in bags and passengers going through security check. Overall, this project will have clearly defined milestones for each step of the process that a passenger goes through in an airport. The visuals for this project would include coding a simple building as the airport, a runway for plane sprites, and people sprites going into the airport. There would also be a table maintaining which passengers are good to go and how much money they spent on tickets and such.

Milestone 1: This milestone will focus on having the passengers coming into the airport, including what airline they will fly on as well as how many bags they have. This milestone should create the functionality of being able to create an introduction screen where a player chooses one of at least three airlines they are flying, how many bags they plan to check, difficulty level, and their destination (at least three options as well). After making these choices the player is taken into the airport where their bags are counted and they are given the letter/number combo of their gate (which they later must have memorized).

Milestone 2: The second milestone furthers baggage check by creating code to determine price per bag based on difficulty and then coming up with a total. This milestone also makes some type of “wallet” or amount of money which can be either randomized at the beginning or determined by difficulty. Should the player not have enough money to pay for all carry ons they have the choice to play a mini “stocks” game where they can invest money in different companies to make more or potentially lose more (the game itself will be made in M3)

Milestone 3: Actually create a stock game with at least 3 companies to invest in, one to lose money, one to either gain or lose depending on difficulty, and one to gain. Have the game add and take directly from the wallet as should the bag check station when enough money is made

Milestone 4: Have the person go through security, randomize whether or not they get through based on difficulty. If they don’t pass within three tries they lose the game and have to restart. If they do pass, they are taken to an area with different hallways leading to every gate, they must choose the gate they were shown in the beginning for their plane. Number of tries to guess depends on difficulty. If they guess correctly they are taken to the gate, if not they are taken back to the beginning.

Milestone 5: For this milestone the player is at their flight gate and given the opportunity to level up to first class, from standard, if they have enough money. Should this occur they can board their flight immediately otherwise the player has to walk around a bit to wait for their boarding time. While walking should they choose to walk into a souvenir store or bathroom they get a message that this made them miss their flight and lose the game. They can also choose to go to a cafe to purchase a coffee in which if they don’t have enough money they play stocks again.



Milestone 6: Milestone 6 takes you onto the plane, where if you have any carry on you must put it in overhang, if it doesn't fit however you must pay an extra fee to put it in the back, if you don't have enough money for this play the stocks game again. Should the player finally be seated in the plane the game is won.