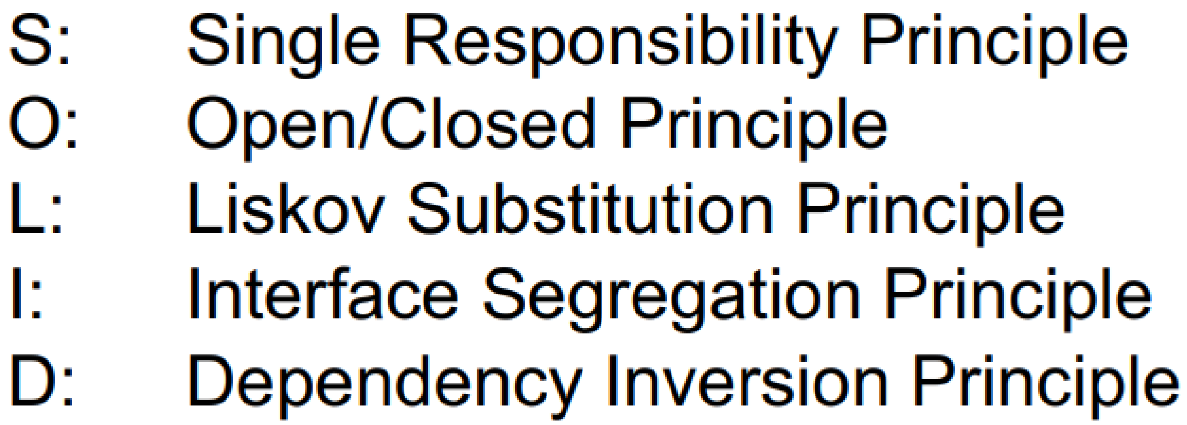
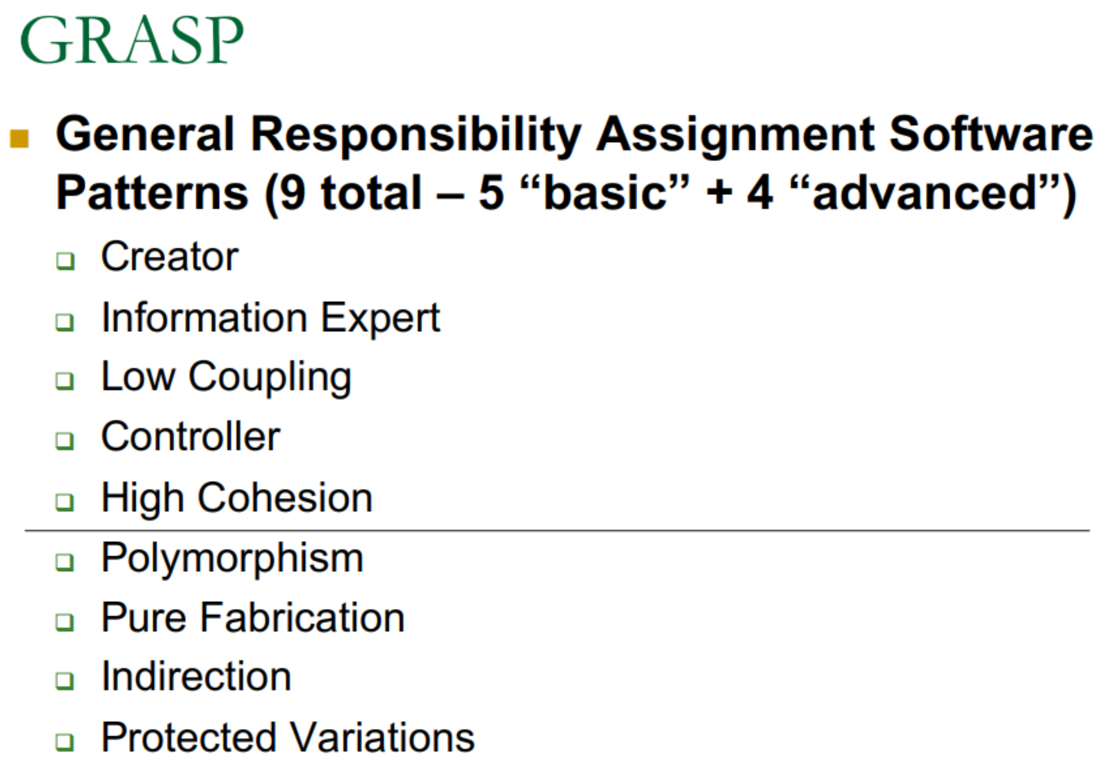
SOLID and GRASP Principles



S:	Single Responsibility Principle
O:	Open/Closed Principle
L:	Liskov Substitution Principle
I:	Interface Segregation Principle
D:	Dependency Inversion Principle

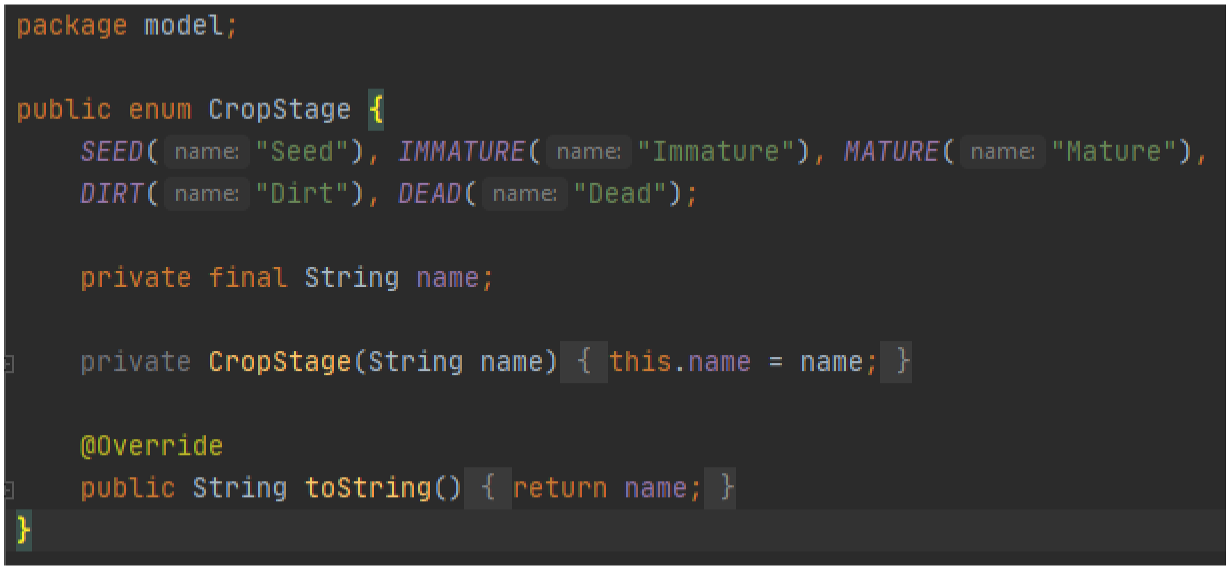


GRASP

- **General Responsibility Assignment Software Patterns (9 total – 5 “basic” + 4 “advanced”)**
  - Creator
  - Information Expert
  - Low Coupling
  - Controller
  - High Cohesion
  - Polymorphism
  - Pure Fabrication
  - Indirection
  - Protected Variations

SOLID Principles (3):

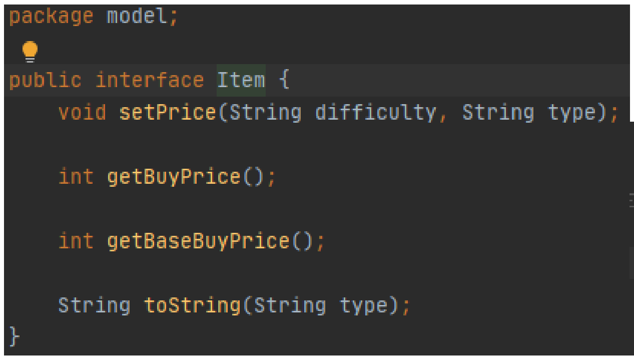
Single Responsibility Principle (SRP):



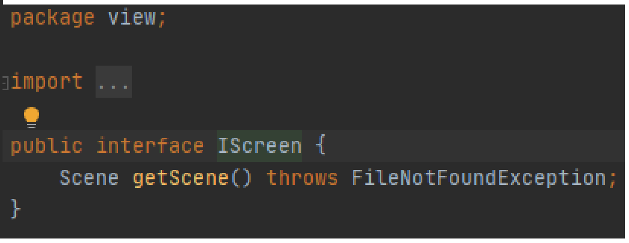
```
package model;

public enum CropStage {
    SEED( name: "Seed"), IMMATURE( name: "Immature"), MATURE( name: "Mature"),
    DIRT( name: "Dirt"), DEAD( name: "Dead");

    private final String name;

    private CropStage(String name) { this.name = name; }

    @Override
    public String toString() { return name; }
}
```

This class enumeration for crop stage follows the SOLID principle of SRP. The class is only responsible for holding the stages of the crop’s life cycle. It is precise and contains short naming conventions that are self-explanatory. It is easy to understand and alter if more crop stages need to be added or removed.

Interface Segregation Principle (ISP):



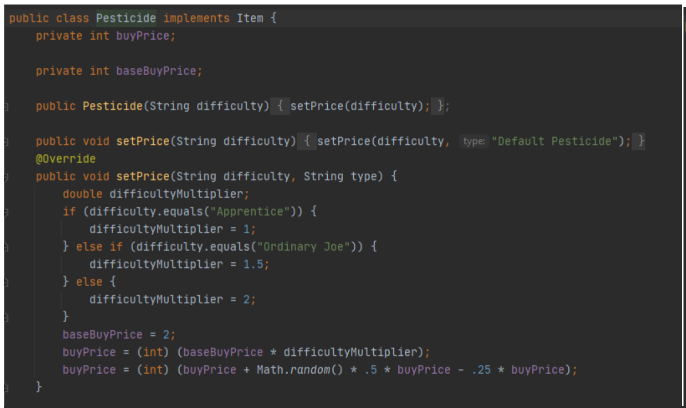
```
package model;

public interface Item {
    void setPrice(String difficulty, String type);

    int getBuyPrice();

    int getBaseBuyPrice();

    String toString(String type);
}
```



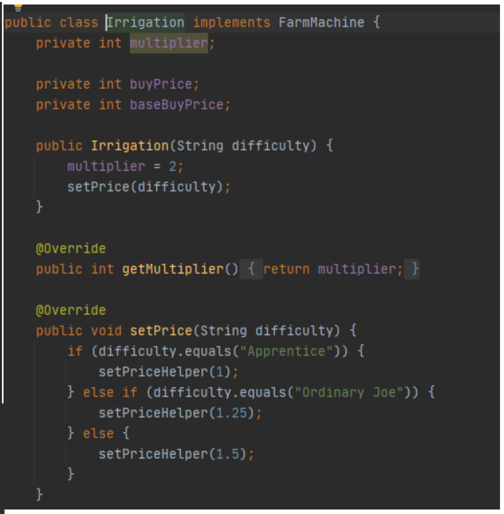
```
package view;

import ...

public interface IScreen {
    Scene getScene() throws FileNotFoundException;
}
```

Both classes are related to the functionality of the Inventory class in our code. Instead of making a larger interface including both, two smaller more focuses interfaces are created. The interfaces are made as small as possible further focusing on the ISP.

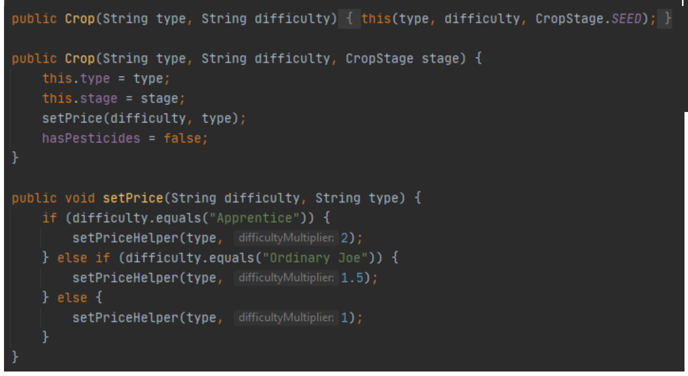
Open / Closed Principle (OCP):



```
public class Pesticide implements Item {
    private int buyPrice;

    private int baseBuyPrice;

    public Pesticide(String difficulty) { setPrice(difficulty); };

    public void setPrice(String difficulty) { setPrice(difficulty, type: "Default Pesticide"); }
    @Override
    public void setPrice(String difficulty, String type) {
        double difficultyMultiplier;
        if (difficulty.equals("Apprentice")) {
            difficultyMultiplier = 1;
        } else if (difficulty.equals("Ordinary Joe")) {
            difficultyMultiplier = 1.5;
        } else {
            difficultyMultiplier = 2;
        }
        baseBuyPrice = 2;
        buyPrice = (int) (baseBuyPrice * difficultyMultiplier);
        buyPrice = (int) (buyPrice + Math.random() * .5 * buyPrice - .25 * buyPrice);
    }
```



```
public class Irrigation implements FarmMachine {
    private int multiplier;

    private int buyPrice;
    private int baseBuyPrice;

    public Irrigation(String difficulty) {
        multiplier = 2;
        setPrice(difficulty);
    }

    @Override
    public int getMultiplier() { return multiplier; }

    @Override
    public void setPrice(String difficulty) {
        if (difficulty.equals("Apprentice")) {
            setPriceHelper(1);
        } else if (difficulty.equals("Ordinary Joe")) {
            setPriceHelper(1.25);
        } else {
            setPriceHelper(1.5);
        }
    }
```

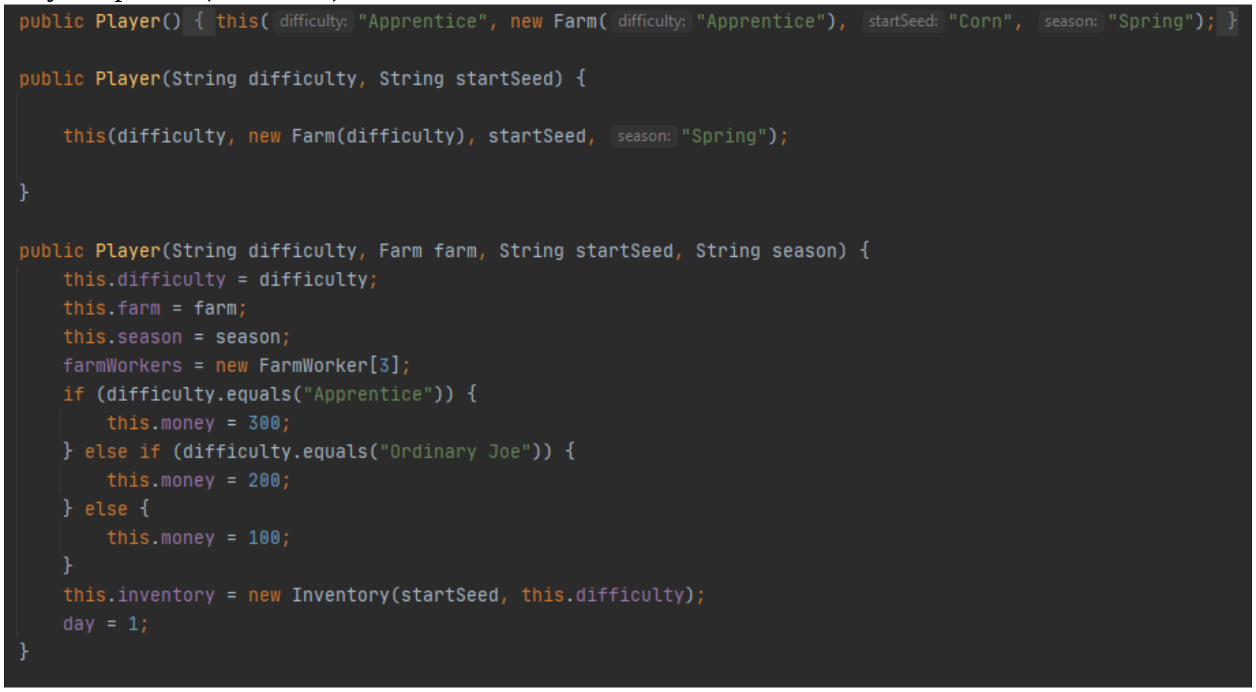


```
public Crop(String type, String difficulty) { this(type, difficulty, CropStage.SEED); }

public Crop(String type, String difficulty, CropStage stage) {
    this.type = type;
    this.stage = stage;
    setPrice(difficulty, type);
    hasPesticides = false;
}

public void setPrice(String difficulty, String type) {
    if (difficulty.equals("Apprentice")) {
        setPriceHelper(type, difficultyMultiplier: 2);
    } else if (difficulty.equals("Ordinary Joe")) {
        setPriceHelper(type, difficultyMultiplier: 1.5);
    } else {
        setPriceHelper(type, difficultyMultiplier: 1);
    }
}
```

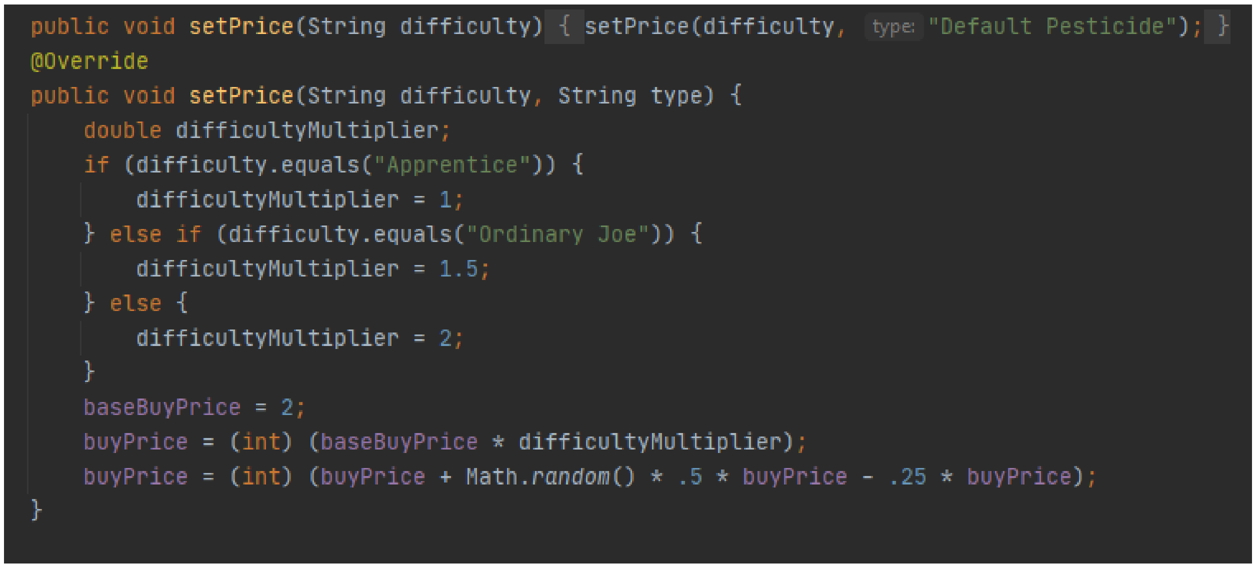
This code above follows the Open / Closed Principle by having a price set in each of the child classes. By placing the variable in each of the classes if follows the principles rule of being open for extension but closed for modification. The functionality is extended by adding code instead of changing the existing code. This makes the code less likely to break the core of the system.

GRASP Principles (5):

Polymorphism (advanced):



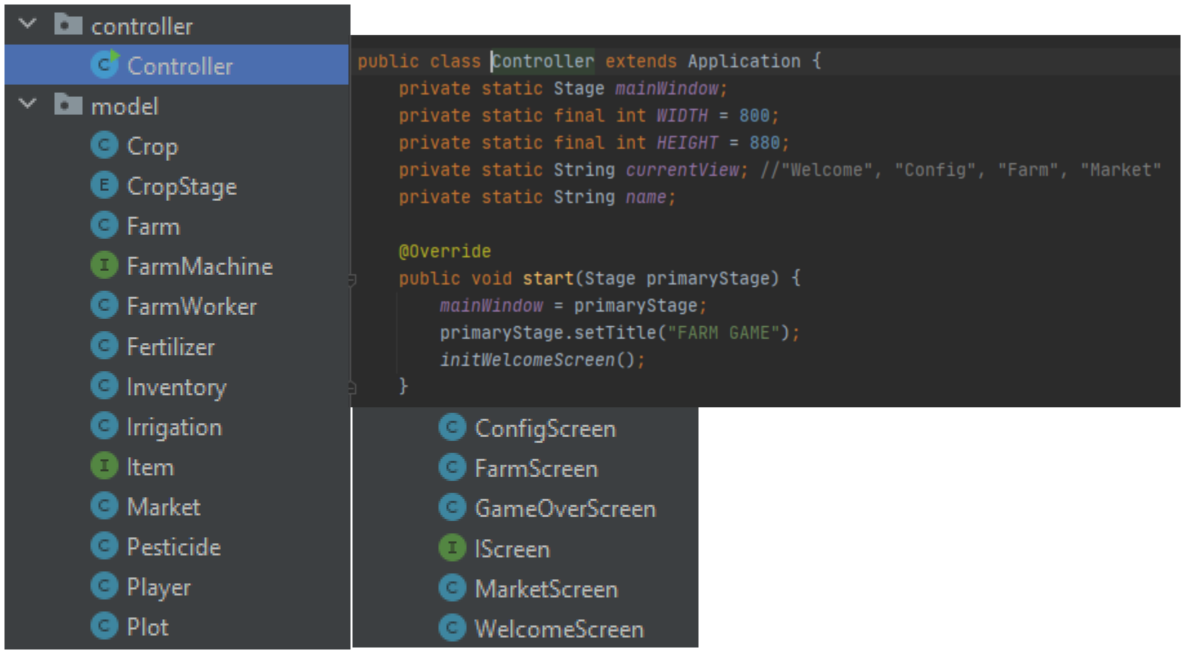
```
public Player() { this( difficulty: "Apprentice", new Farm( difficulty: "Apprentice"),  startSeed: "Corn",  season: "Spring"); }

public Player(String difficulty, String startSeed) {

    this(difficulty, new Farm(difficulty), startSeed,  season: "Spring");

}

public Player(String difficulty, Farm farm, String startSeed, String season) {
    this.difficulty = difficulty;
    this.farm = farm;
    this.season = season;
    farmWorkers = new FarmWorker[3];
    if (difficulty.equals("Apprentice")) {
        this.money = 300;
    } else if (difficulty.equals("Ordinary Joe")) {
        this.money = 200;
    } else {
        this.money = 100;
    }
    this.inventory = new Inventory(startSeed, this.difficulty);
    day = 1;
}
```



```
public void setPrice(String difficulty) { setPrice(difficulty,  type: "Default Pesticide"); }
@Override
public void setPrice(String difficulty, String type) {
    double difficultyMultiplier;
    if (difficulty.equals("Apprentice")) {
        difficultyMultiplier = 1;
    } else if (difficulty.equals("Ordinary Joe")) {
        difficultyMultiplier = 1.5;
    } else {
        difficultyMultiplier = 2;
    }
    baseBuyPrice = 2;
    buyPrice = (int) (baseBuyPrice * difficultyMultiplier);
    buyPrice = (int) (buyPrice + Math.random() * .5 * buyPrice - .25 * buyPrice);
}
```

The setPrice method is used twice in this section of code and the player method is used three times. The parameters of them are different to account for a default case and a case where the type is specified. The name of the method is simple and reusable which allows for polymorphism between changing and adding parameters. This allows for added parameters to make a new method with the same name that will add specificity, if needed, to alter the pricing.

Controller (basic):

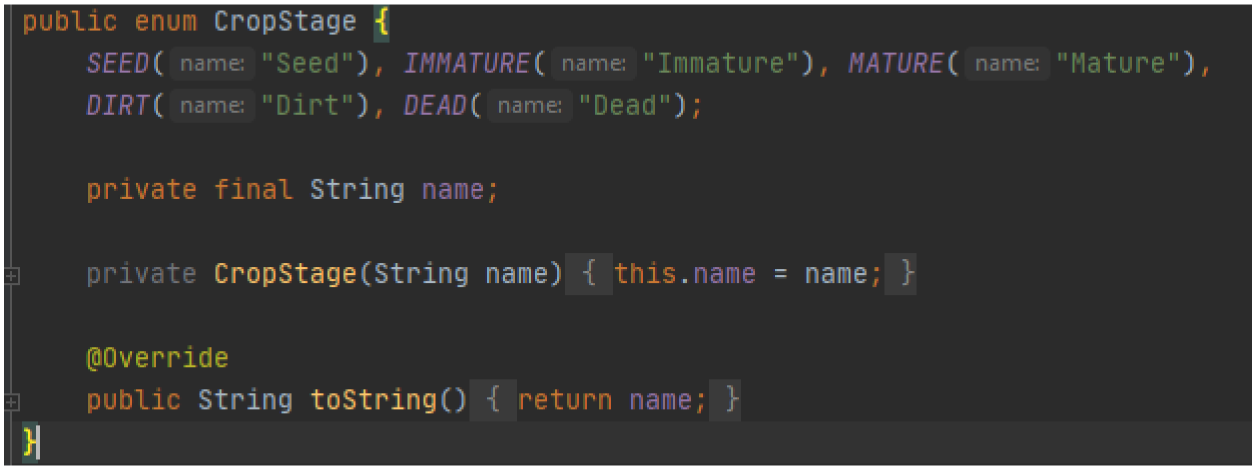


```
controller
    Controller
model
    Crop
    CropStage
    Farm
    FarmMachine
    FarmWorker
    Fertilizer
    Inventory
    Irrigation
    Item
    Market
    Pesticide
    Player
    Plot
    ConfigScreen
    FarmScreen
    GameOverScreen
    IScreen
    MarketScreen
    WelcomeScreen
```

```java
public class Controller extends Application {
    private static Stage mainWindow;
    private static final int WIDTH = 800;
    private static final int HEIGHT = 880;
    private static String currentView; //"Welcome", "Config", "Farm", "Market"
    private static String name;

    @Override
    public void start(Stage primaryStage) {
        mainWindow = primaryStage;
        primaryStage.setTitle("FARM GAME");
        initWelcomeScreen();
    }
```

The code for the controller utilized only a small portion of the code, as a controller should. It demonstrates the controller principle of GRASP by having multiple methods other than the controller (Screen methods) to control the code. It delegates to other objects work to be done as well as coordinates and controls high level activity.

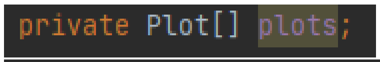
Pure Fabrication (advanced):



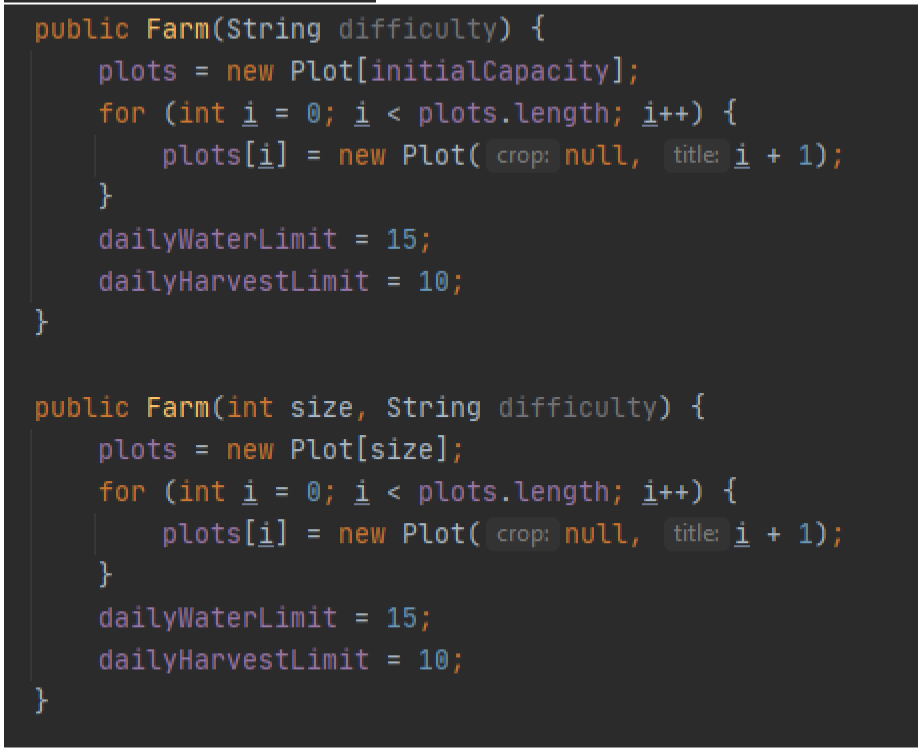
```java
public enum CropStage {
    SEED( name: "Seed"), IMMATURE( name: "Immature"), MATURE( name: "Mature"),
    DIRT( name: "Dirt"), DEAD( name: "Dead");

    private final String name;

    private CropStage(String name) { this.name = name; }

    @Override
    public String toString() { return name; }
}
```

In attempting to add a crop stage variable that would keep track of a crops various growth stages we had no current method that could be used to hold this information without convoluting the method. We decided to create a new class that is solely responsible for holding the crop Stages. This follows the GRASP principle of pure fabrication since there was not an appropriate class to house the different crop stages, resulting in a new fabricated class called CropStage.

Creator (basic):



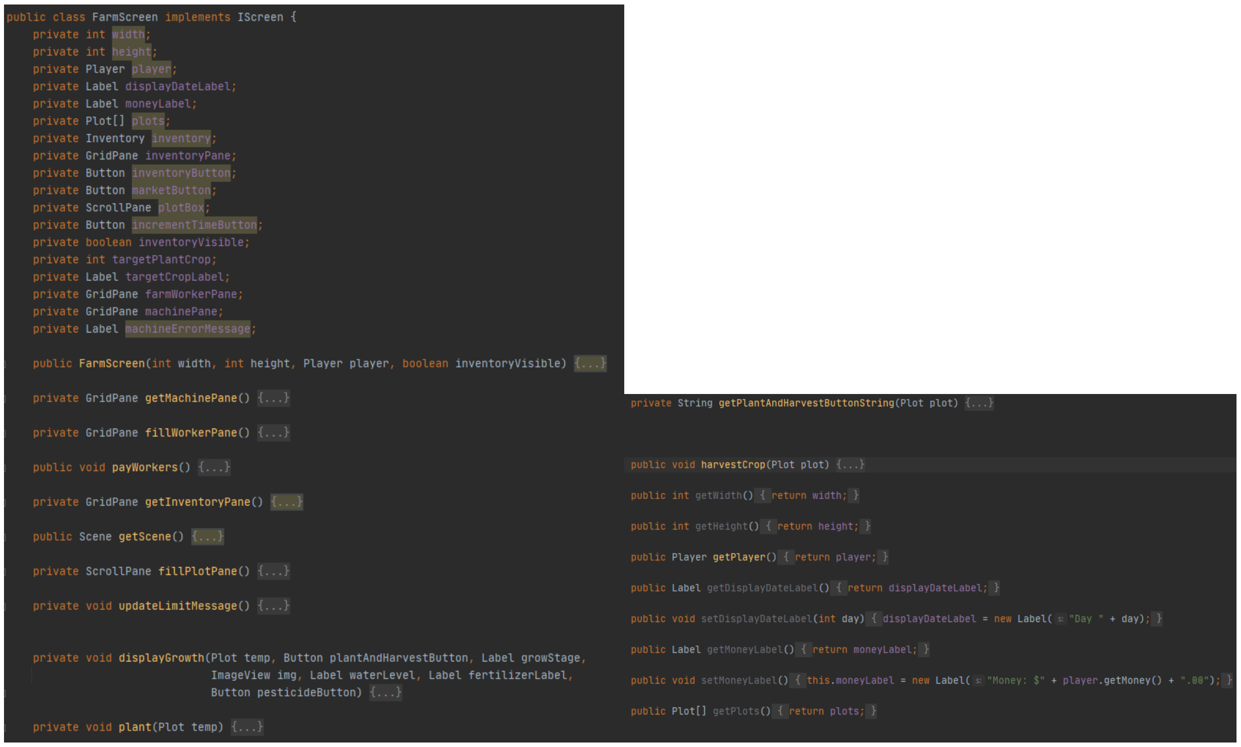
```
private Plot[] plots;
```



```
public Farm(String difficulty) {
    plots = new Plot[initialCapacity];
    for (int i = 0; i < plots.length; i++) {
        plots[i] = new Plot( crop: null,  title: i + 1);
    }
    dailyWaterLimit = 15;
    dailyHarvestLimit = 10;
}

public Farm(int size, String difficulty) {
    plots = new Plot[size];
    for (int i = 0; i < plots.length; i++) {
        plots[i] = new Plot( crop: null,  title: i + 1);
    }
    dailyWaterLimit = 15;
    dailyHarvestLimit = 10;
}
```

There are many examples of the creator principle of GRASP in our project's code. One simplistic example of this is the Farm class. The Farm class creates a Plot array and functions as a Creator for those plots. Farm contains instances of the class Plot thus the class Farm functions as a creator for the Plot[].

Information Expert (basic):



```
public class FarmScreen implements IScreen {
    private int width;
    private int height;
    private Player player;
    private Label displayDateLabel;
    private Label moneyLabel;
    private Plot[] plots;
    private Inventory inventory;
    private GridPane inventoryPane;
    private Button inventoryButton;
    private Button marketButton;
    private ScrollPane plotBox;
    private Button incrementTimeButton;
    private boolean inventoryVisible;
    private int targetPlantCrop;
    private Label targetCropLabel;
    private GridPane farmWorkerPane;
    private GridPane machinePane;
    private Label machineErrorMessage;

    public FarmScreen(int width, int height, Player player, boolean inventoryVisible) {...}

    private GridPane getMachinePane() {...}

    private GridPane fillWorkerPane() {...}

    public void payWorkers() {...}

    private GridPane getInventoryPane() {...}

    public Scene getScene() {...}

    private ScrollPane fillPlotPane() {...}

    private void updateLimitMessage() {...}


    private void displayGrowth(Plot temp, Button plantAndHarvestButton, Label growStage,
                               ImageView img, Label waterLevel, Label fertilizerLabel,
                               Button pesticideButton) {...}

    private void plant(Plot temp) {...}
```

```
private String getPlantAndHarvestButtonString(Plot plot) {...}


public void harvestCrop(Plot plot) {...}

public int getWidth() { return width; }

public int getHeight() { return height; }

public Player getPlayer() { return player; }

public Label getDisplayDateLabel() { return displayDateLabel; }

public void setDisplayDateLabel(int day) { displayDateLabel = new Label( s: "Day " + day); }

public Label getMoneyLabel() { return moneyLabel; }

public void setMoneyLabel() { this.moneyLabel = new Label( s: "Money: $" + player.getMoney() + ".00"); }

public Plot[] getPlots() { return plots; }
```

The FarmScreen class serves as an Information Expert for most of the code in each milestone. It contains several of the child classes and most other classes will branch off this overarching class. It follows the GRASP information expert principle since many other classes are referencing based of the information stored in this class. In other words it contains a significant responsibility of the code since it also contains such a high amount of information stored from the code.