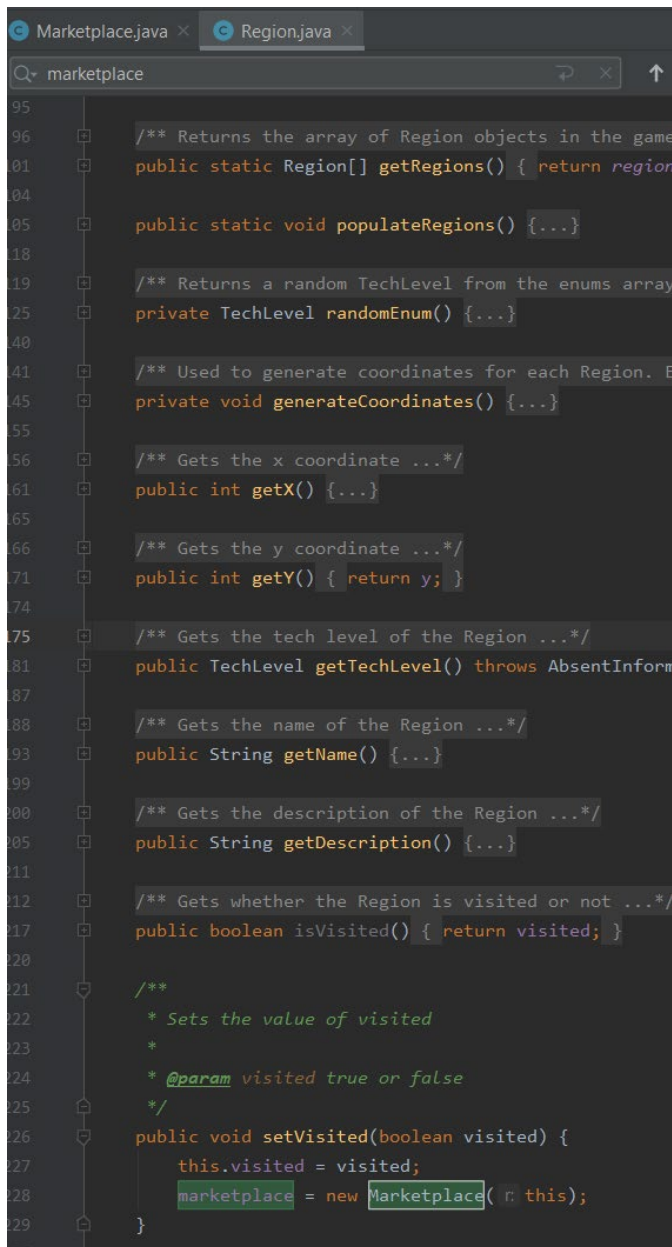


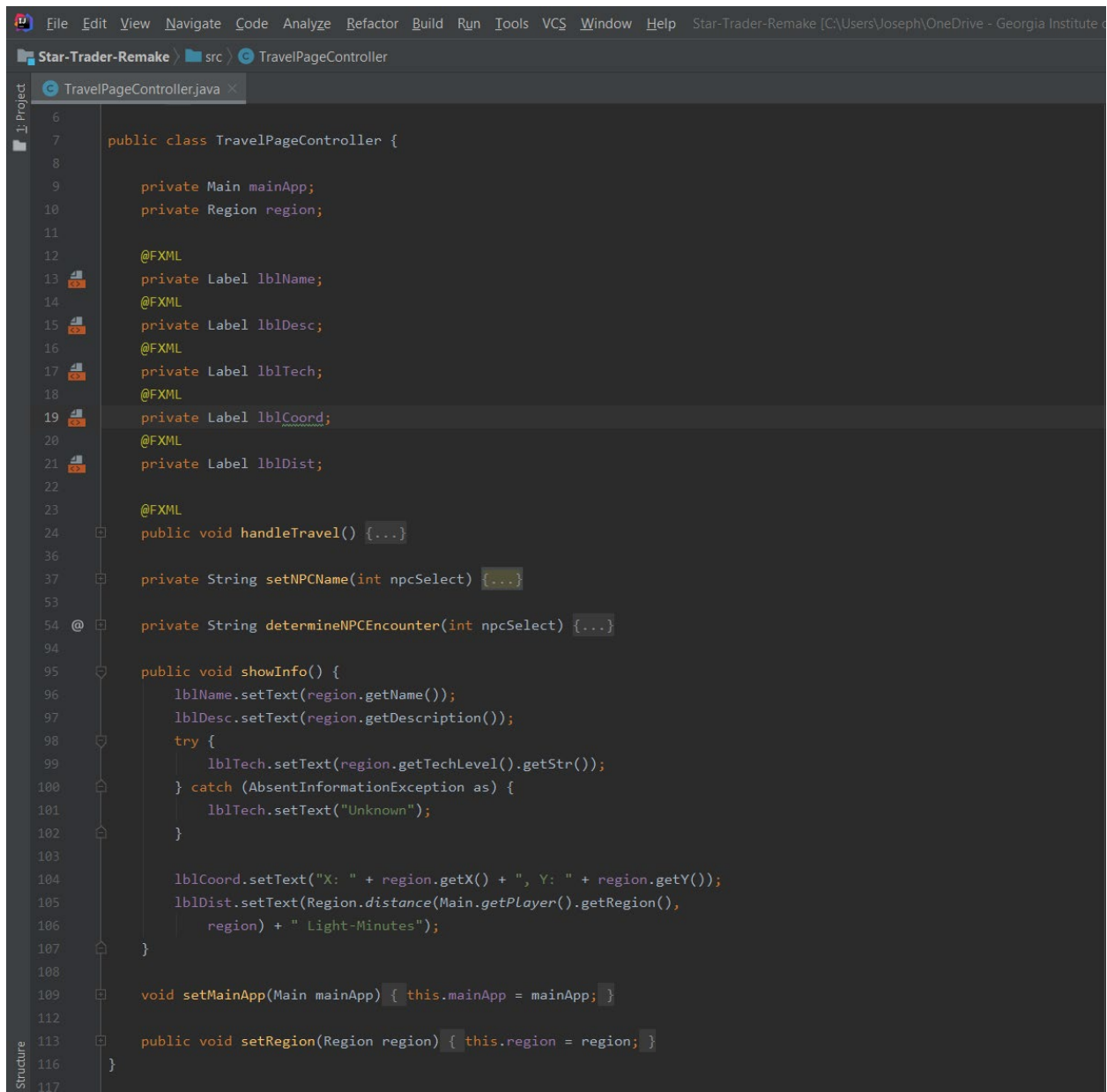
## GRASP Principle #1: Creator



```
95
96  /** Returns the array of Region objects in the game
101  public static Region[] getRegions() { return region
104
105  public static void populateRegions() {...}
118
119  /** Returns a random TechLevel from the enums array
125  private TechLevel randomEnum() {...}
140
141  /** Used to generate coordinates for each Region. E
145  private void generateCoordinates() {...}
155
156  /** Gets the x coordinate ...*/
161  public int getX() {...}
165
166  /** Gets the y coordinate ...*/
171  public int getY() { return y; }
174
175  /** Gets the tech level of the Region ...*/
181  public TechLevel getTechLevel() throws AbsentInform
187
188  /** Gets the name of the Region ...*/
193  public String getName() {...}
199
200  /** Gets the description of the Region ...*/
205  public String getDescription() {...}
211
212  /** Gets whether the Region is visited or not ...*/
217  public boolean isVisited() { return visited; }
220
221  /**
222   * Sets the value of visited
223   *
224   * @param visited true or false
225   */
226  public void setVisited(boolean visited) {
227      this.visited = visited;
228      marketplace = new Marketplace( r, this);
229  }
```

This is a screenshot of the Region class for our game. As shown, a Region object creates/initializes a Marketplace object. This is the Creator GRASP pattern because the Region object contains a Marketplace object.

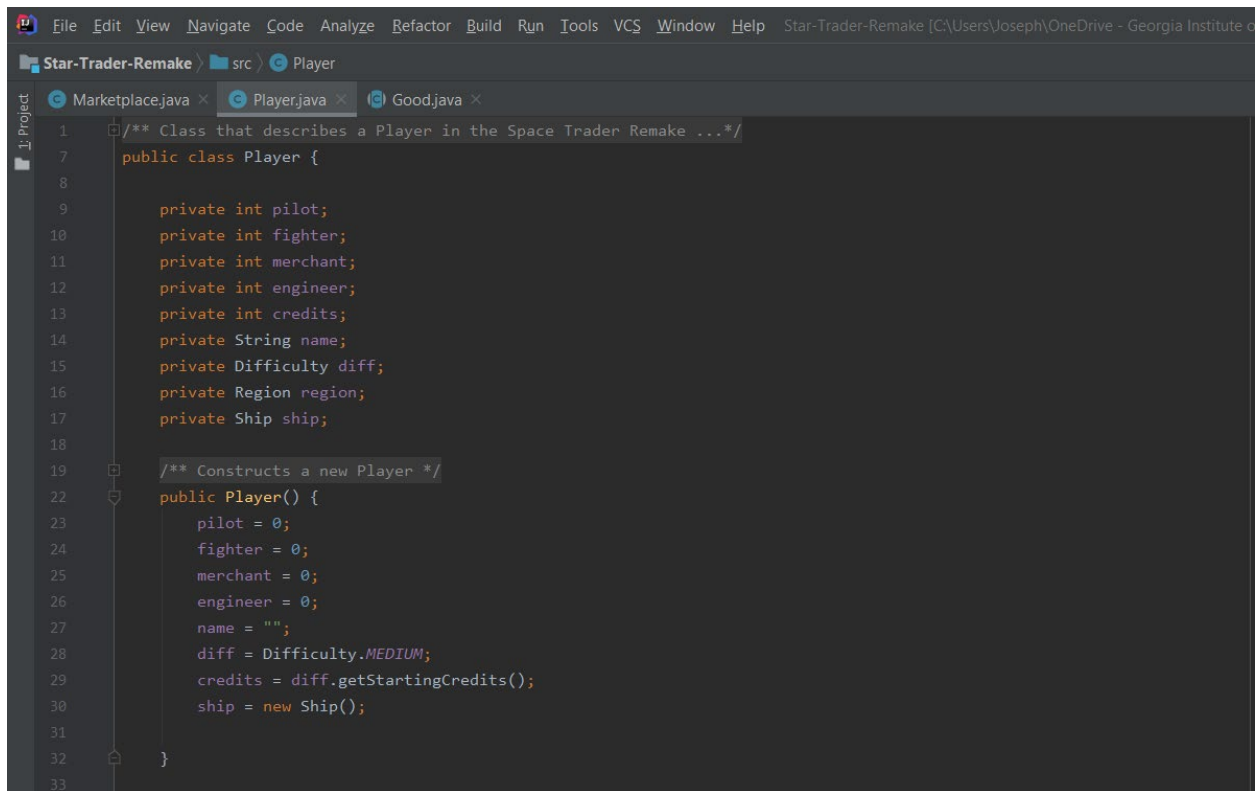
## GRASP Principle #2: Controller



```
6
7 public class TravelPageController {
8
9     private Main mainApp;
10    private Region region;
11
12    @FXML
13    private Label lblName;
14    @FXML
15    private Label lblDesc;
16    @FXML
17    private Label lblTech;
18    @FXML
19    private Label lblCoord;
20    @FXML
21    private Label lblDist;
22
23    @FXML
24    public void handleTravel() {...}
25
26
27    private String setNPCName(int npcSelect) {...}
28
29
30    private String determineNPCEncounter(int npcSelect) {...}
31
32
33    public void showInfo() {
34        lblName.setText(region.getName());
35        lblDesc.setText(region.getDescription());
36        try {
37            lblTech.setText(region.getTechLevel().getStr());
38        } catch (AbsentInformationException as) {
39            lblTech.setText("Unknown");
40        }
41
42        lblCoord.setText("X: " + region.getX() + ", Y: " + region.getY());
43        lblDist.setText(Region.distance(Main.getPlayer().getRegion(),
44            region) + " Light-Minutes");
45    }
46
47    void setMainApp(Main mainApp) { this.mainApp = mainApp; }
48
49    public void setRegion(Region region) { this.region = region; }
50
51 }
```

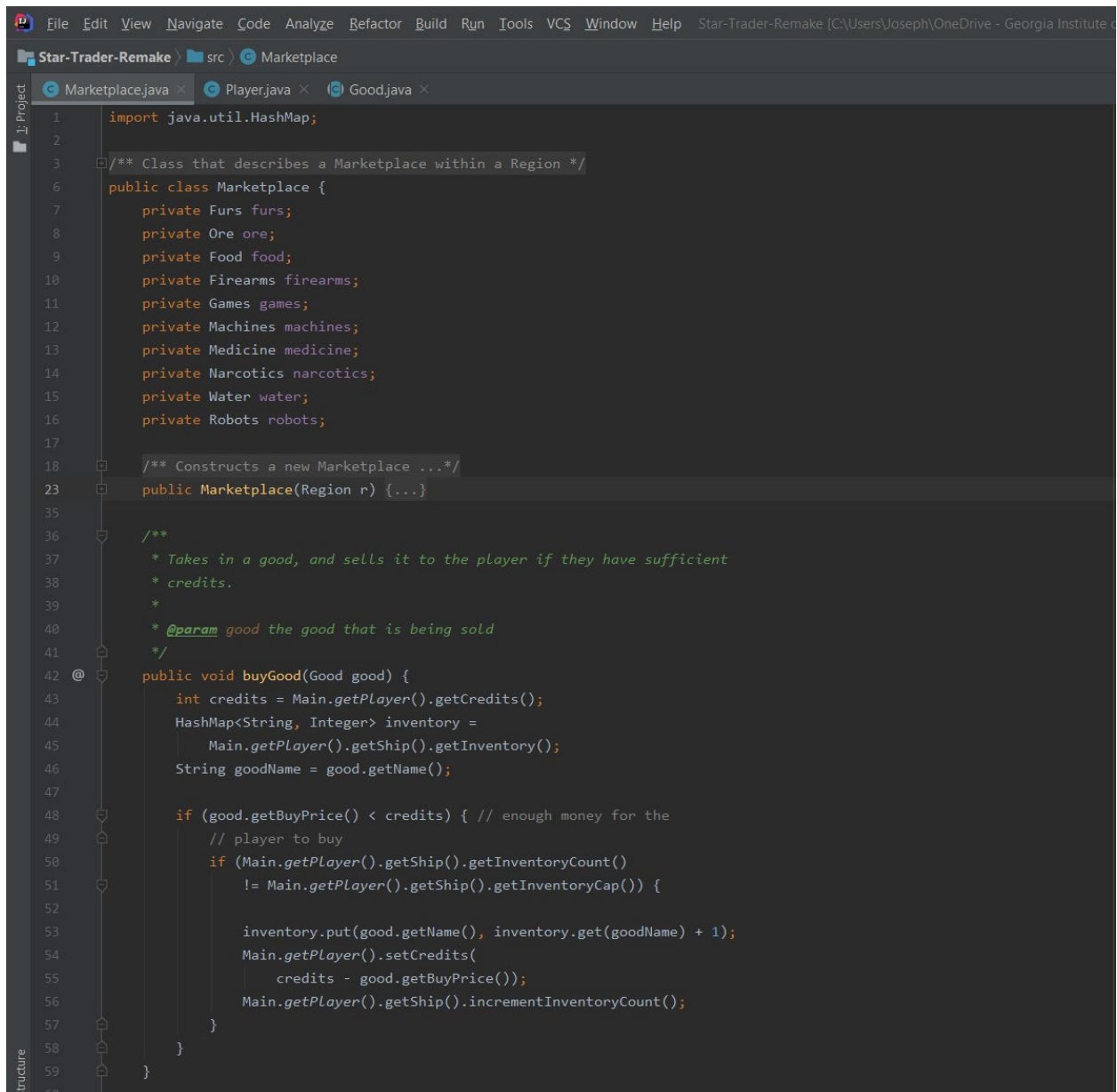
This is a screenshot of the TravelPageController class. As can be seen, this is a class that handles UI logic. Because it is used to pass on information and coordinate input to the model classes, it fulfills the controller pattern.

### GRASP Principle #3: Information Expert



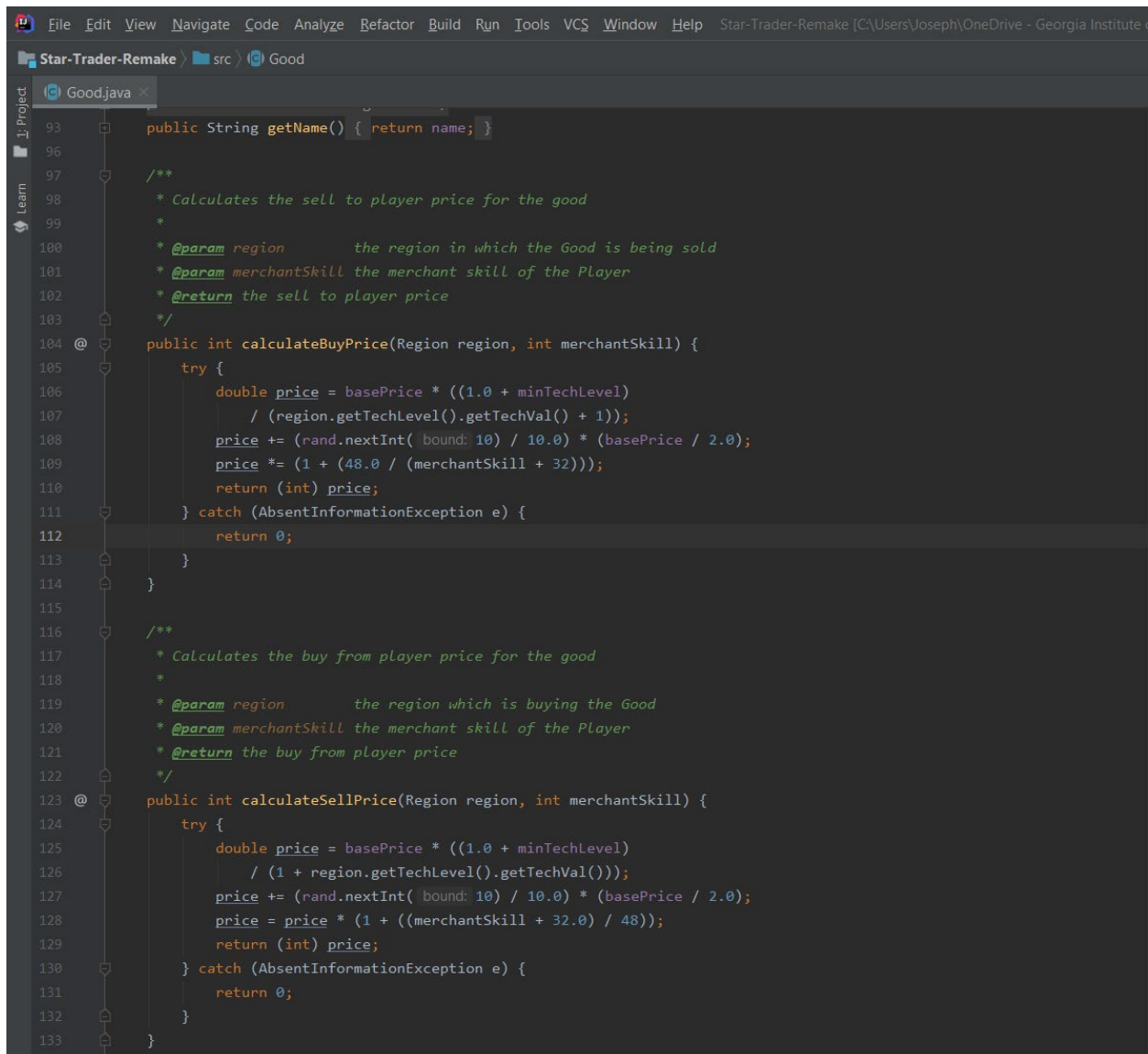
This screenshot shows the Player class. This class has all the information necessary to contain a Ship class. In fact, it contains the only Ship object in the entire game. Thus, it fulfills the information expert pattern.

## GRASP Principle #4: Polymorphism



This screenshot shows the Marketplace class. Inside this class, there are multiple methods that take in a Good, which is an abstract class. Because the actual instances are more specific classes that extend Good, this displays polymorphism.

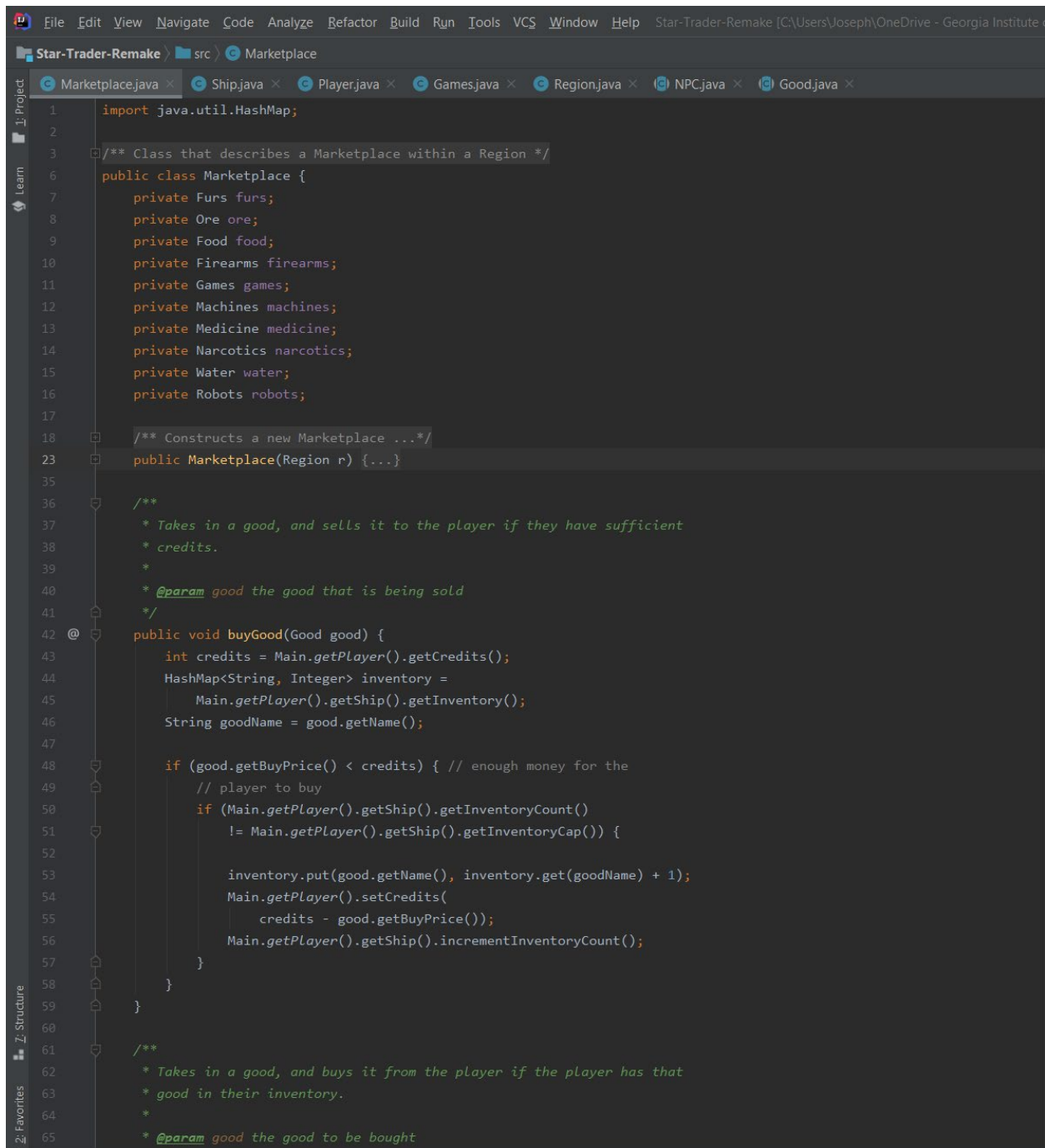
## GRASP Principle #5: Low Coupling



```
Star-Trader-Remake [C:\Users\Joseph\OneDrive - Georgia Institute of Technology\Documents\Star-Trader-Remake]
src \ Good
Good.java x
93 public String getName() { return name; }
96
97 /**
98  * Calculates the sell to player price for the good
99  *
100  * @param region the region in which the Good is being sold
101  * @param merchantSkill the merchant skill of the Player
102  * @return the sell to player price
103  */
104 @ public int calculateBuyPrice(Region region, int merchantSkill) {
105     try {
106         double price = basePrice * ((1.0 + minTechLevel)
107             / (region.getTechLevel().getTechVal() + 1));
108         price += (rand.nextInt( bound: 10) / 10.0) * (basePrice / 2.0);
109         price *= (1 + (48.0 / (merchantSkill + 32)));
110         return (int) price;
111     } catch (AbsentInformationException e) {
112         return 0;
113     }
114 }
115
116 /**
117  * Calculates the buy from player price for the good
118  *
119  * @param region the region which is buying the Good
120  * @param merchantSkill the merchant skill of the Player
121  * @return the buy from player price
122  */
123 @ public int calculateSellPrice(Region region, int merchantSkill) {
124     try {
125         double price = basePrice * ((1.0 + minTechLevel)
126             / (1 + region.getTechLevel().getTechVal()));
127         price += (rand.nextInt( bound: 10) / 10.0) * (basePrice / 2.0);
128         price = price * (1 + ((merchantSkill + 32.0) / 48));
129         return (int) price;
130     } catch (AbsentInformationException e) {
131         return 0;
132     }
133 }
```

This screenshot shows the Good class. In this class, there are two methods used to calculate the buying and selling price for the good. The alternative way of handling this problem would've been to have the buying and selling price calculated inside the Marketplace class. However, the solution presented demonstrates low coupling because it prevents fields of the good class from being populated by the Marketplace class.

## SOLID Principle #1: SRP

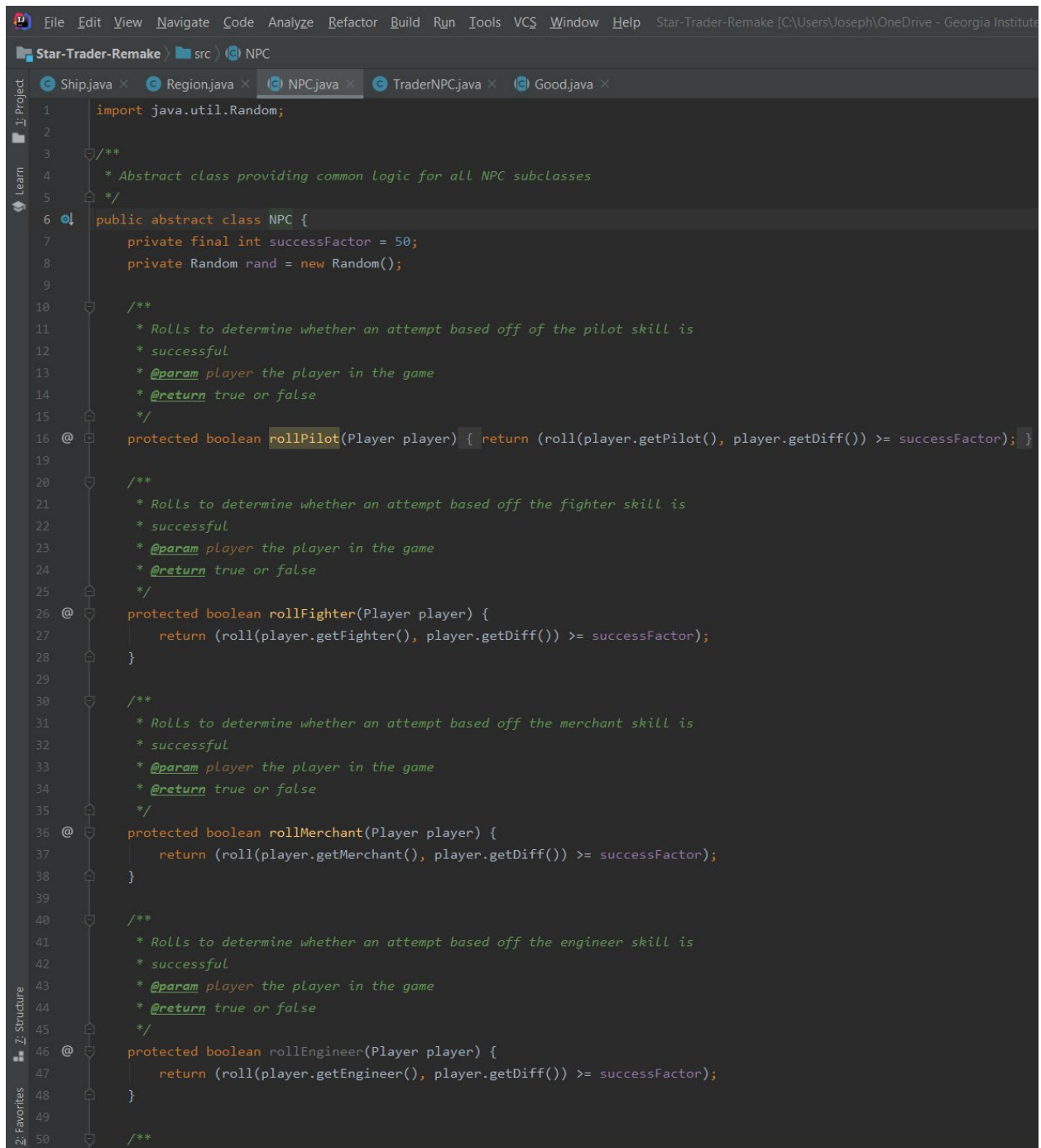


```
1  import java.util.HashMap;
2
3  /** Class that describes a Marketplace within a Region */
6  public class Marketplace {
7      private Furs furs;
8      private Ore ore;
9      private Food food;
10     private Firearms firearms;
11     private Games games;
12     private Machines machines;
13     private Medicine medicine;
14     private Narcotics narcotics;
15     private Water water;
16     private Robots robots;
17
18     /** Constructs a new Marketplace ...*/
23     public Marketplace(Region r) {...}
35
36     /**
37      * Takes in a good, and sells it to the player if they have sufficient
38      * credits.
39      *
40      * @param good the good that is being sold
41      */
42     @
43     public void buyGood(Good good) {
44         int credits = Main.getPlayer().getCredits();
45         HashMap<String, Integer> inventory =
46             Main.getPlayer().getShip().getInventory();
47         String goodName = good.getName();
48
49         if (good.getBuyPrice() < credits) { // enough money for the
50             // player to buy
51             if (Main.getPlayer().getShip().getInventoryCount()
52                 != Main.getPlayer().getShip().getInventoryCap()) {
53
54                 inventory.put(good.getName(), inventory.get(goodName) + 1);
55                 Main.getPlayer().setCredits(
56                     credits - good.getBuyPrice());
57                 Main.getPlayer().getShip().incrementInventoryCount();
58             }
59         }
60
61         /**
62          * Takes in a good, and buys it from the player if the player has that
63          * good in their inventory.
64          *
65          * @param good the good to be bought
```

This screenshot shows the Marketplace class. Its responsibility is to handle buying and selling of any Goods as well as selling fuel/repairs for a Ship. Because its responsibilities do not include anything extraneous, this class demonstrates the Single Responsibility Principle.



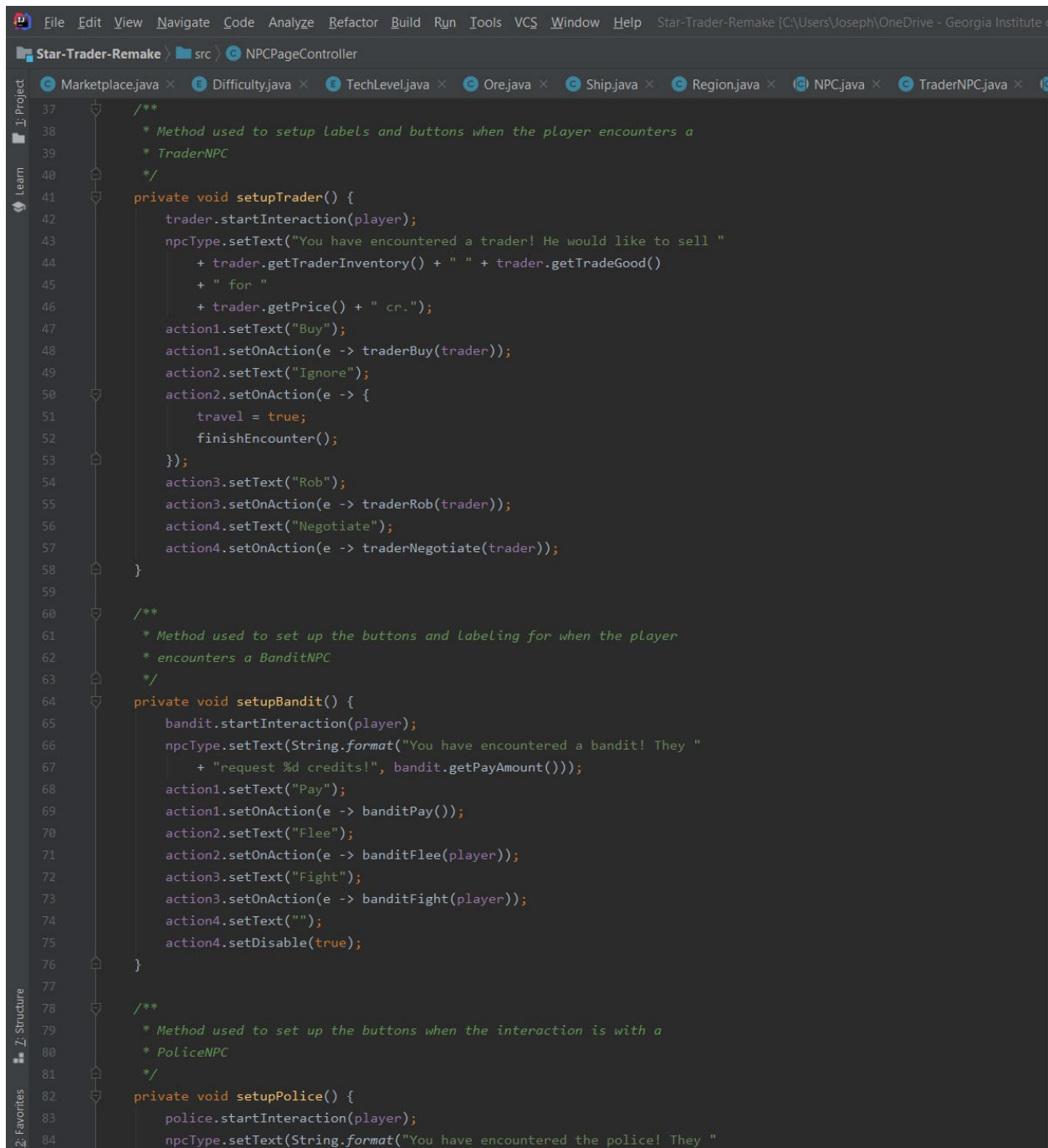
## SOLID Principle #2: OCP



```
1  import java.util.Random;
2
3  /**
4   * Abstract class providing common logic for all NPC subclasses
5   */
6  public abstract class NPC {
7      private final int successFactor = 50;
8      private Random rand = new Random();
9
10     /**
11      * Rolls to determine whether an attempt based off of the pilot skill is
12      * successful
13      * @param player the player in the game
14      * @return true or false
15      */
16     @protected boolean rollPilot(Player player) { return (roll(player.getPilot(), player.getDiff()) >= successFactor); }
17
18     /**
19      * Rolls to determine whether an attempt based off the fighter skill is
20      * successful
21      * @param player the player in the game
22      * @return true or false
23      */
24     @protected boolean rollFighter(Player player) {
25         return (roll(player.getFighter(), player.getDiff()) >= successFactor);
26     }
27
28     /**
29      * Rolls to determine whether an attempt based off the merchant skill is
30      * successful
31      * @param player the player in the game
32      * @return true or false
33      */
34     @protected boolean rollMerchant(Player player) {
35         return (roll(player.getMerchant(), player.getDiff()) >= successFactor);
36     }
37
38     /**
39      * Rolls to determine whether an attempt based off the engineer skill is
40      * successful
41      * @param player the player in the game
42      * @return true or false
43      */
44     @protected boolean rollEngineer(Player player) {
45         return (roll(player.getEngineer(), player.getDiff()) >= successFactor);
46     }
47
48     /**
49      *
50     */
```

This picture is a screenshot of the NPC abstract class. It provides common logic for all NPC types. This demonstrates the Open/Closed principle because the NPC class itself should not need to be changed when a new NPC is added.

### SOLID Principle #3: DIP



```
37  /**
38   * Method used to setup Labels and buttons when the player encounters a
39   * TraderNPC
40   */
41  private void setupTrader() {
42      trader.startInteraction(player);
43      npcType.setText("You have encountered a trader! He would like to sell "
44          + trader.getTraderInventory() + " " + trader.getTradeGood()
45          + " for "
46          + trader.getPrice() + " cr.");
47      action1.setText("Buy");
48      action1.setOnAction(e -> traderBuy(trader));
49      action2.setText("Ignore");
50      action2.setOnAction(e -> {
51          travel = true;
52          finishEncounter();
53      });
54      action3.setText("Rob");
55      action3.setOnAction(e -> traderRob(trader));
56      action4.setText("Negotiate");
57      action4.setOnAction(e -> traderNegotiate(trader));
58  }
59
60  /**
61   * Method used to set up the buttons and Labeling for when the player
62   * encounters a BanditNPC
63   */
64  private void setupBandit() {
65      bandit.startInteraction(player);
66      npcType.setText(String.format("You have encountered a bandit! They "
67          + "request %d credits!", bandit.getPayAmount()));
68      action1.setText("Pay");
69      action1.setOnAction(e -> banditPay());
70      action2.setText("Flee");
71      action2.setOnAction(e -> banditFlee(player));
72      action3.setText("Fight");
73      action3.setOnAction(e -> banditFight(player));
74      action4.setText("");
75      action4.setDisable(true);
76  }
77
78  /**
79   * Method used to set up the buttons when the interaction is with a
80   * PoliceNPC
81   */
82  private void setupPolice() {
83      police.startInteraction(player);
84      npcType.setText(String.format("You have encountered the police! They "
```

This screenshot shows the NPCPageController. In each of its methods, it forwards the handling of actual logic to the appropriate NPC class. However, because the controller work is handled by the controller class, it does not depend heavily on the NPC subclasses.