TEAM 78

SOLID

1. SOLID Principle 1 - Single Responsibility

```java
public class Coordinate {

    private int x;
    private int y;

    /** Creates a new coordinate object with random coordinates ...*/
    public Coordinate() { this( x: (int) (Math.random() * 400) - 200,  y: (int) (Math.random() * 400) - 200); }

    /** Creates a new coordinate object ...*/
    public Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /** Gets the x-coordinate ...*/
    public int getX() { return x; }

    /** Gets the y-coordinate ...*/
    public int getY() { return y; }

    /** Calculates the distance between two coordinates ...*/
    public static double distance(Coordinate coordinate1, Coordinate coordinate2) {
        return Math.sqrt(Math.pow(coordinate1.x - coordinate2.x, 2)
                + Math.pow(coordinate1.y - coordinate2.y, 2));
    }
```

The Coordinate class contains just X and Y coordinates and allows for calculations of distance between two coordinates.

2. SOLID Principle 2 - Open/Closed

```java
public class MarketItem {

    /** Name of item. */
    protected String officialItemName;
```

The MarketItem class is used for all items the Player can buy from the Market

```java
public class UniverseDeed extends MarketItem {
    public UniverseDeed(String playerName) {
        super(MarketGoods.UNIVERSE);
        officialItemName = playerName + "'s Universe";
    }
}
```

The UniverseDeed class is used for the win condition item. Here we extend the MarketItem to add the functionality of including the Player's name in the item name without modifying the closed MarketItem class.

3. SOLID Principle 3 - Dependency Inversion

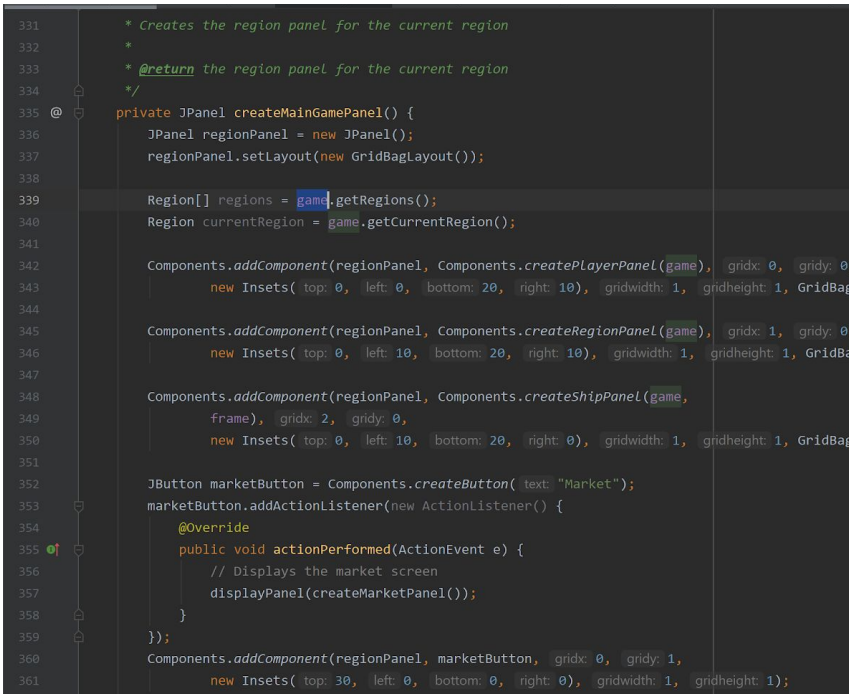
```java
for (MarketItem marketItem : currentMarket.getMarketItemsInRegion()) {

    JLabel inventoryOfItem = Components.createHeader2( text: "Inventory: "
            + game.getCurrentCount(marketItem));

    Components.addComponent(marketPanel,
            Components.createHeader2(marketItem.getOfficialItemName()), gridx: 0, y,
            new Insets( top: 0, left: 0, bottom: 0, right: 5));
    int itemPrice = game.getCost(marketItem);
    Components.addComponent(marketPanel, Components.createHeader2( text: "$" + itemPrice),
            gridx: 1, y, new Insets( top: 0, left: 5, bottom: 0, right: 5));
```

Here we loop through every MarketItem in the current Region's Market to display all MarketItems. This depends only on the iterator giving us MarketItems, which can include a MarketItem or the UniverseDeed object.

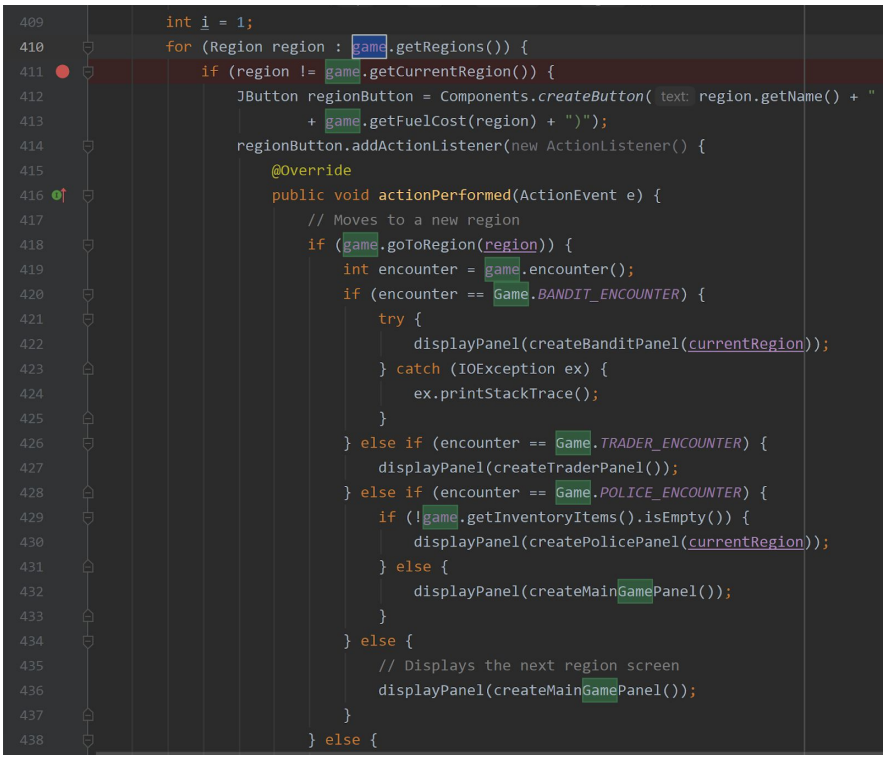
GRASP

1. **GRASP Principle 1 - Controller (Game)**

Game class acts as the main intermediate class between the player operating in SpaceTrader.java with most of the other classes in the project. It is the class that controls and delegates the input of the user to the correct classes.



```
331         * Creates the region panel for the current region
332         *
333         * @return the region panel for the current region
334         */
335 @       private JPanel createMainGamePanel() {
336             JPanel regionPanel = new JPanel();
337             regionPanel.setLayout(new GridBagLayout());
338 
339             Region[] regions = game.getRegions();
340             Region currentRegion = game.getCurrentRegion();
341 
342             Components.addComponent(regionPanel, Components.createPlayerPanel(game),  gridx: 0,  gridy: 0
343                     new Insets( top: 0,  left: 0,  bottom: 20,  right: 10),  gridwidth: 1,  gridheight: 1, GridBag
344 
345             Components.addComponent(regionPanel, Components.createRegionPanel(game),  gridx: 1,  gridy: 0
346                     new Insets( top: 0,  left: 10,  bottom: 20,  right: 10),  gridwidth: 1,  gridheight: 1, GridBa
347 
348             Components.addComponent(regionPanel, Components.createShipPanel(game,
349                     frame),  gridx: 2,  gridy: 0,
350                     new Insets( top: 0,  left: 10,  bottom: 20,  right: 0),  gridwidth: 1,  gridheight: 1, GridBag
351 
352             JButton marketButton = Components.createButton( text: "Market");
353             marketButton.addActionListener(new ActionListener() {
354                 @Override
355 o               public void actionPerformed(ActionEvent e) {
356                     // Displays the market screen
357                     displayPanel(createMarketPanel());
358                 }
359             });
360             Components.addComponent(regionPanel, marketButton,  gridx: 0,  gridy: 1,
361                     new Insets( top: 30,  left: 0,  bottom: 0,  right: 0),  gridwidth: 1,  gridheight: 1);
```



```
409             int i = 1;
410             for (Region region : game.getRegions()) {
411                 if (region != game.getCurrentRegion()) {
412                     JButton regionButton = Components.createButton( text: region.getName() + "
413                             + game.getFuelCost(region) + ")");
414                     regionButton.addActionListener(new ActionListener() {
415                         @Override
416 o                       public void actionPerformed(ActionEvent e) {
417                             // Moves to a new region
418                             if (game.goToRegion(region)) {
419                                 int encounter = game.encounter();
420                                 if (encounter == Game.BANDIT_ENCOUNTER) {
421                                     try {
422                                         displayPanel(createBanditPanel(currentRegion));
423                                     } catch (IOException ex) {
424                                         ex.printStackTrace();
425                                     }
426                                 } else if (encounter == Game.TRADER_ENCOUNTER) {
427                                     displayPanel(createTraderPanel());
428                                 } else if (encounter == Game.POLICE_ENCOUNTER) {
429                                     if (!game.getInventoryItems().isEmpty()) {
430                                         displayPanel(createPolicePanel(currentRegion));
431                                     } else {
432                                         displayPanel(createMainGamePanel());
433                                     }
434                                 } else {
435                                     // Displays the next region screen
436                                     displayPanel(createMainGamePanel());
437                                 }
438                             } else {
```

2. **GRASP Principle 2 - Creator (Universe and Regions)**

The Universe class is responsible for creating the regions, assigning the technology levels, and creating a comprehensive coordinate grid for the regions.

```java
/**
 * Creates a new Universe with regions specified by the region names
 *
 * @param regionNames the names of the regions in the new universe
 */
public Universe(String[] regionNames) {
    regions = new Region[regionNames.length];

    for (int i = 0; i < regions.length; i++) {
        // Creates coordinates for each region randomly, validating that none of the regions are
        // within 5 in either the X or Y direction
        Coordinate coordinate = new Coordinate();
        for (int j = 0; j < i; j++) {
            if (Coordinate.distanceX(coordinate, regions[j].getCoordinate()) <= 5
                    || Coordinate.distanceY(coordinate, regions[j].getCoordinate()) <= 5) {
                coordinate = new Coordinate();
                j = 0;
            }
        }

        // Gets a random tech level for each region
        TechLevel techLevel = TechLevel.values()[(int) (Math.random()
                * TechLevel.values().length)];

        regions[i] = new Region(regionNames[i], coordinate, techLevel);
    }
}
```

3. GRASP Principle 3 - Information Expert (Player)

```java
public class Player {

    private String name;

    private int credits;

    private int pilotPoints;
    private int fighterPoints;
    private int merchantPoints;
    private int engineerPoints;

    private Ship ship;
```

The Player class contains information for the Player's configuration, as well as the health/fuel/inventory contained in the ship.

```java
/**
 * Gets the player's name
 *
 * @return the player's name
 */
public String getPlayerName() { return player.getName(); }

/**
 * Gets the player's pilot points
 *
 * @return the player's pilot points
 */
public int getPilotPoints() { return player.getPilotPoints(); }

/**
 * Gets the player's fighter points
 *
 * @return the player's fighter points
 */
public int getFighterPoints() { return player.getFighterPoints(); }
```

The Player class is referenced from elsewhere to retrieve the information stored in the Player class.

4. **GRASP Principle 4 - High Cohesion**
   Ship and Player have their own specific tasks that have interconnected logic. The player class only contains logic regarding skill points while the ship class only contains logic regarding fuel, inventory, and health. Their connection comes through the interaction of the game as the variables that the ship has are continually, directly and indirectly, impacted by your skill points. Directly, the engineering skill point allows the player to repair the ship which directly coincides with health. Indirectly, all of the skill points help through giving the player better opportunities in randomized situations. For example, a high level of pilot skill points in the player will allow the player to have higher chances to escape malicious NPC's, effectively saving the ship's inventory, health, etc.

```java
/**
 * Creates a new Player
 *
 * @param name the player's name
 * @param pilotPoints pilot points
 * @param fighterPoints fighter points
 * @param merchantPoints merchant points
 * @param engineerPoints engineer points
 */
public Player(String name, int pilotPoints, int fighterPoints, int merchantPoints,
              int engineerPoints) {
    this.name = name;

    this.pilotPoints = pilotPoints;
    this.fighterPoints = fighterPoints;
    this.merchantPoints = merchantPoints;
    this.engineerPoints = engineerPoints;

    this.credits = 0;

    this.ship = new Ship(ShipType.STARTER);
}
```

```java
public class Ship {
    private String type;
    private Inventory inventory;
    private int currentFuel;
    private int maxFuelCapacity;
    private int currentHealth;
    private int maxHealth;
```

5. GRASP Principle 5 - Low Coupling

The Inventory never directly interacts with any of the NPCs since there are not many reasonable connections between those two types. An NPC may clear the inventory but only through the Player's Ship and even then, their access is incredibly limited.

```java
/** Inventory of every item in Ship. Fuel is not counted in Inventory. */
public class Inventory {

    private int size;
    private int usedSpace;
    private HashMap<String, Integer> items;

    public Inventory(int size) {
        this.size = size;
        this.usedSpace = 0;
        items = new HashMap<>();
    }

    public int getSize() { return size; }

    public int getUsedSpace() { return usedSpace; }

    public int getCurrentCount(MarketItem item) {
        if (items.containsKey(item.getOfficialItemName())) {
            return items.get(item.getOfficialItemName());
        }
        return 0;
    }
```

```java
                game.getPlayer().changeCredits( variable: -1 * bandit.getMoneyDemanded());
                JOptionPane.showMessageDialog(frame,  message: "You have paid the Bandit $"
                        + bandit.getMoneyDemanded());
            } else if (game.getPlayer().getShip().getCurrentUsedSpace() > 0) {
                game.getPlayer().getShip().removeAllItems();
                JOptionPane.showMessageDialog(frame,  message: "Because you are not able to pay the"
                        + "Bandit, you have lost all of your inventory.");
            } else {
                // ship gets damaged
                int damage = (int) (Math.sqrt(bandit.getDamageCaused()));
                if (!game.getPlayer().getShip().alterCurrentHealth( variable: -1 * damage)) {
                    displayPanel(createLossPanel());
```