```java
/**
 * Creates a Player object.
 * @param name the name of the player
 * @param credits the number of credits they should start with
 * @param currentLocation where the player starts
 * @param pilotPoints their pilot points
 * @param fighterPoints their fighter points
 * @param merchantPoints their merchant points
 * @param engineerPoints their engineer points
 */
Player(String name, int credits, Region currentLocation, int pilotPoints,
       int fighterPoints, int merchantPoints, int engineerPoints) {
    this.name = name;
    this.credits = credits;
    this.currentLocation = currentLocation;
    this.pilotPoints = pilotPoints;
    this.fighterPoints = fighterPoints;
    this.merchantPoints = merchantPoints;
    this.engineerPoints = engineerPoints;

    newShip(ShipType.STARSHIP);

    playerInventory = new Inventory();

    if (Game.difficulty.getDifficulty() == 1) {
        this.setStats( initialFuel: 1000,  initialWater: 50,  initialFood: 50);
    } else if (Game.difficulty.getDifficulty() == 2) {
        this.setStats( initialFuel: 800,  initialWater: 30,  initialFood: 30);
    } else {
        this.setStats( initialFuel: 600,  initialWater: 10,  initialFood: 10);
    }
}
```

Creator

Player creates the Ship object because the player “owns” the Ship and holds the reference to that Ship.

```java
/**
 * Gets a random item from the player's inventory.
 * @return the item
 */
protected Item getRandomItem() {
    Random rand = new Random();
    int index = rand.nextInt( bound: 10) + 1;
    return itemList.get(index);
}
```

Information Expert

Inventory has the information to allow it to choose a Random good within an inventory, so Inventory performs that task.

```java
/**
 * Represents a Region object.
 * Each Region has an economic level--a TechLevel, a name, and an (x, y)
 * location.
 */
public class Region {

    /**
     * The x-coordinate of the Region's location.
     */
    private int xCord;

    /**
     * The y-coordinate of the Region's location.
     */
    private int yCord;

    /**
     * The region's TechLevel--what kind of economy it has.
     */
    private TechLevel tech = TechLevel.random();

    /**
     * The name of the Region.
     */
    private String name;

    /**
     * The Inventory of the Region's marketplace.
     */
    private Inventory inventory;
```

High Cohesion

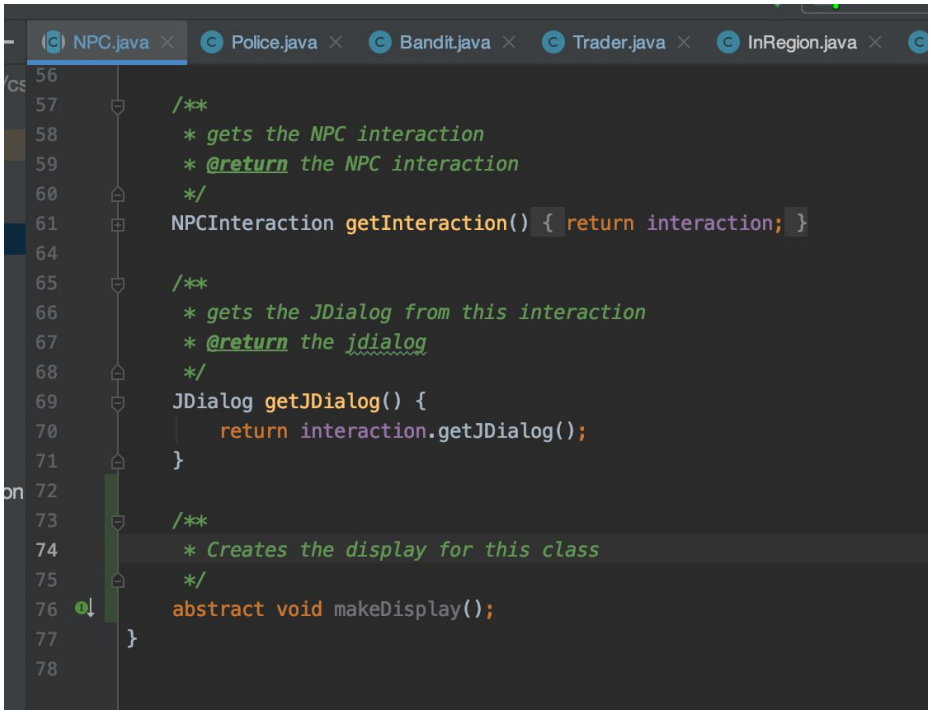
Region has a very specific job of recording all the information for an individual region, including its location, TechLevel, and Marketplace.

```java
public enum ShipType {
    STARSHIP( cargoCapacity: 20,  health: 50),
    BFR( cargoCapacity: 50,  health: 100);

    private int cargoCapacity;
    private int health;

    /**
     * constructor
     *
     * @param cargoCapacity the capacity of the ship
     * @param health the health of the ship
     */
    ShipType(int cargoCapacity, int health) {
        this.cargoCapacity = cargoCapacity;
        this.health = health;
    }

    /**
     * getter for cargo capacity
     *
     * @return the cargo capacity
     */
    public int getCargoCapacity() { return cargoCapacity; }
```

Single Responsibility Principle

ShipType is a clean Enum with a limited scope. It only defines the types of ships that may exist and those ships' healths and cargo capacities.

```
        }
        if (npc == null) {
            return null;
        } else {
            npc.makeDisplay();
            return npc.getJDialog();
        }
    }
}
```



```
    /**
     * Creates a new Police officer.
     */
    Police() {
        super( name: "Police Officer");
    }

    @Override
    void makeDisplay() {
        getInteraction().policeDisplay();
    }
}
```

Polymorphism

The NPC is an abstract class with three concrete child classes: Police, Bandit, and Trader. Our code doesn't do explicit checks to see if we are dealing with a Police, Bandit, or Trader encounter--instead, we take advantage of polymorphism so that the required methods can be called on any NPC and create the correct pop-up.

```
flee.addActionListener((ActionEvent e) -> {
    banditDialog.dispose();
    KarmaController.goodAction();
    if (NPC.pilotCheck(Game.playerOne)) {
        goBack = true;
        JOptionPane.showMessageDialog(mainFrame, message: "You escaped "
                + "from the bandit!");
```

```
class KarmaController {

    /**
     * Handles a Bad Action occurring.
     */
    static void badAction() {
        Game.playerOne.lowerKarma();
    }

    /**
     * Handles a Good Action occurring.
     */
    static void goodAction() {
        Game.playerOne.increaseKarma();
    }

    /**
     * Checks if the Player has good Karma.
     * @return if the Player's karma is at least 5
     */
    static boolean hasGoodKarma() {
        return (Game.playerOne.getKarma() >= 5);
    }
}
```

Controller

For keeping track of the Karma a player accumulates and managing the transfer of information from the frontend to the backend, we use a controller class called KarmaController. As shown in the screenshots, whenever a

player does something good, KarmaController is notified, and KarmaController then makes the appropriate backend modifications. The frontend does not directly interact with the backend.

```
public class EncounterGenerator {

    /**
     * A method to determine if an encounter will happen during travel.
     * @param difficulty the difficulty of the game
     * @param player the player for this game
     * @return the NPC that the player encountered; or null for no encounter
     */
    static JDialog determineEncounter(
            GameDifficulty difficulty, Player player) {
        Random rand = new Random();
        NPC npc = null;
        int encounterNum = rand.nextInt( bound: 30);
        if (difficulty == GameDifficulty.EASY) {
            if (encounterNum == 0
                    && player.getPlayerInventory().hasItems()) {
                npc = new Police();
```

Pure Fabrication

The EncounterGenerator class doesn't represent any kind of physical "real-world" object. It only exists to generate encounters, which makes the code cleaner and increases cohesion.

```
JButton button0 = new RoundButton(currentList.get(0).getName(),
        currentList.get(0).getX(), currentList.get(0).getY());
mapPanel.add(button0);
button0.setBounds( x: currentList.get(0).getX() + adjustX,  y: adjustY
        - currentList.get(0).getY(),  width: 40,  height: 40);
```

Liskov Substitution Principle

Within the InRegion class, we use RoundButton (a class that is derived from JButton) to form our map panel. RoundButton is a subtype of JButton.

This code shows that JButton objects can be replaced with RoundButton objects, without altering the properties of the program. Hence, it is an example of the Liskov Substitution Principle.

```
/**
 * Makes a screen for when the Player is in a region, taking in a Game,
 * an initialRegion, and a list of all the Regions.
 * @param initialRegion the first region the player should be in
 * @param game the game the player is in
 * @param regions the list of all the regions in this game
 */
protected InRegion(Region initialRegion, Game game,
                   ArrayList<Region> regions) {
    mainFrame.setVisible(true);
    mainFrame.setTitle("Regions");
    this.initialRegion = initialRegion;
    Game.playerOne.setCurrentLocation(initialRegion);
    this.game = game;
    this.regions = regions;
    screenSetup();
}
```

Dependency Inversion Principle

An instance of Game and Region is passed into the constructor which is an example of constructor injection, a type of dependency injection. This means that instead of the Region class needing to know which Game and Region it will use, whoever creates the Region class provides it with a Game and a Region.