GRASP Principle #1: Creator

```
Marketplace.java   Region.java
marketplace

    /** Returns the array of Region objects in the game
    public static Region[] getRegions() { return region

    public static void populateRegions() {...}

    /** Returns a random TechLevel from the enums array
    private TechLevel randomEnum() {...}

    /** Used to generate coordinates for each Region. E
    private void generateCoordinates() {...}

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

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

    /** Gets the tech level of the Region ...*/
    public TechLevel getTechLevel() throws AbsentInform

    /** Gets the name of the Region ...*/
    public String getName() {...}

    /** Gets the description of the Region ...*/
    public String getDescription() {...}

    /** Gets whether the Region is visited or not ...*/
    public boolean isVisited() { return visited; }

    /**
     * Sets the value of visited
     *
     * @param visited true or false
     */
    public void setVisited(boolean visited) {
        this.visited = visited;
        marketplace = new Marketplace( r: this);
    }
```

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

```java
public class TravelPageController {

    private Main mainApp;
    private Region region;

    @FXML
    private Label lblName;
    @FXML
    private Label lblDesc;
    @FXML
    private Label lblTech;
    @FXML
    private Label lblCoord;
    @FXML
    private Label lblDist;

    @FXML
    public void handleTravel() {...}

    private String setNPCName(int npcSelect) {...}

    private String determineNPCEncounter(int npcSelect) {...}

    public void showInfo() {
        lblName.setText(region.getName());
        lblDesc.setText(region.getDescription());
        try {
            lblTech.setText(region.getTechLevel().getStr());
        } catch (AbsentInformationException as) {
            lblTech.setText("Unknown");
        }

        lblCoord.setText("X: " + region.getX() + ", Y: " + region.getY());
        lblDist.setText(Region.distance(Main.getPlayer().getRegion(),
            region) + " Light-Minutes");
    }

    void setMainApp(Main mainApp) { this.mainApp = mainApp; }

    public void setRegion(Region region) { this.region = region; }
}
```

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

```java
/** Class that describes a Player in the Space Trader Remake ...*/
public class Player {

    private int pilot;
    private int fighter;
    private int merchant;
    private int engineer;
    private int credits;
    private String name;
    private Difficulty diff;
    private Region region;
    private Ship ship;

    /** Constructs a new Player */
    public Player() {
        pilot = 0;
        fighter = 0;
        merchant = 0;
        engineer = 0;
        name = "";
        diff = Difficulty.MEDIUM;
        credits = diff.getStartingCredits();
        ship = new Ship();

    }
```

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

```java
import java.util.HashMap;

/** Class that describes a Marketplace within a Region */
public class Marketplace {
    private Furs furs;
    private Ore ore;
    private Food food;
    private Firearms firearms;
    private Games games;
    private Machines machines;
    private Medicine medicine;
    private Narcotics narcotics;
    private Water water;
    private Robots robots;

    /** Constructs a new Marketplace ...*/
    public Marketplace(Region r) {...}

    /**
     * Takes in a good, and sells it to the player if they have sufficient
     * credits.
     *
     * @param good the good that is being sold
     */
    public void buyGood(Good good) {
        int credits = Main.getPlayer().getCredits();
        HashMap<String, Integer> inventory =
            Main.getPlayer().getShip().getInventory();
        String goodName = good.getName();

        if (good.getBuyPrice() < credits) { // enough money for the
            // player to buy
            if (Main.getPlayer().getShip().getInventoryCount()
                != Main.getPlayer().getShip().getInventoryCap()) {

                inventory.put(good.getName(), inventory.get(goodName) + 1);
                Main.getPlayer().setCredits(
                    credits - good.getBuyPrice());
                Main.getPlayer().getShip().incrementInventoryCount();
            }
        }
    }
```

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

```java
    public String getName() { return name; }

    /**
     * Calculates the sell to player price for the good
     *
     * @param region        the region in which the Good is being sold
     * @param merchantSkill the merchant skill of the Player
     * @return the sell to player price
     */
    public int calculateBuyPrice(Region region, int merchantSkill) {
        try {
            double price = basePrice * ((1.0 + minTechLevel)
                / (region.getTechLevel().getTechVal() + 1));
            price += (rand.nextInt( bound: 10) / 10.0) * (basePrice / 2.0);
            price *= (1 + (48.0 / (merchantSkill + 32)));
            return (int) price;
        } catch (AbsentInformationException e) {
            return 0;
        }
    }

    /**
     * Calculates the buy from player price for the good
     *
     * @param region        the region which is buying the Good
     * @param merchantSkill the merchant skill of the Player
     * @return the buy from player price
     */
    public int calculateSellPrice(Region region, int merchantSkill) {
        try {
            double price = basePrice * ((1.0 + minTechLevel)
                / (1 + region.getTechLevel().getTechVal()));
            price += (rand.nextInt( bound: 10) / 10.0) * (basePrice / 2.0);
            price = price * (1 + ((merchantSkill + 32.0) / 48));
            return (int) price;
        } catch (AbsentInformationException e) {
            return 0;
        }
    }
```

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

```java
import java.util.HashMap;

/** Class that describes a Marketplace within a Region */
public class Marketplace {
    private Furs furs;
    private Ore ore;
    private Food food;
    private Firearms firearms;
    private Games games;
    private Machines machines;
    private Medicine medicine;
    private Narcotics narcotics;
    private Water water;
    private Robots robots;

    /** Constructs a new Marketplace ...*/
    public Marketplace(Region r) {...}

    /**
     * Takes in a good, and sells it to the player if they have sufficient
     * credits.
     *
     * @param good the good that is being sold
     */
    public void buyGood(Good good) {
        int credits = Main.getPlayer().getCredits();
        HashMap<String, Integer> inventory =
            Main.getPlayer().getShip().getInventory();
        String goodName = good.getName();

        if (good.getBuyPrice() < credits) { // enough money for the
            // player to buy
            if (Main.getPlayer().getShip().getInventoryCount()
                != Main.getPlayer().getShip().getInventoryCap()) {

                inventory.put(good.getName(), inventory.get(goodName) + 1);
                Main.getPlayer().setCredits(
                    credits - good.getBuyPrice());
                Main.getPlayer().getShip().incrementInventoryCount();
            }
        }
    }

    /**
     * Takes in a good, and buys it from the player if the player has that
     * good in their inventory.
     *
     * @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

```java
import java.util.Random;

/**
 * Abstract class providing common logic for all NPC subclasses
 */
public abstract class NPC {
    private final int successFactor = 50;
    private Random rand = new Random();

    /**
     * Rolls to determine whether an attempt based off of the pilot skill is
     * successful
     * @param player the player in the game
     * @return true or false
     */
    protected boolean rollPilot(Player player) { return (roll(player.getPilot(), player.getDiff()) >= successFactor); }

    /**
     * Rolls to determine whether an attempt based off the fighter skill is
     * successful
     * @param player the player in the game
     * @return true or false
     */
    protected boolean rollFighter(Player player) {
        return (roll(player.getFighter(), player.getDiff()) >= successFactor);
    }

    /**
     * Rolls to determine whether an attempt based off the merchant skill is
     * successful
     * @param player the player in the game
     * @return true or false
     */
    protected boolean rollMerchant(Player player) {
        return (roll(player.getMerchant(), player.getDiff()) >= successFactor);
    }

    /**
     * Rolls to determine whether an attempt based off the engineer skill is
     * successful
     * @param player the player in the game
     * @return true or false
     */
    protected boolean rollEngineer(Player player) {
        return (roll(player.getEngineer(), player.getDiff()) >= successFactor);
    }

    /**
```

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

```java
    /**
     * Method used to setup labels and buttons when the player encounters a
     * TraderNPC
     */
    private void setupTrader() {
        trader.startInteraction(player);
        npcType.setText("You have encountered a trader! He would like to sell "
            + trader.getTraderInventory() + " " + trader.getTradeGood()
            + " for "
            + trader.getPrice() + " cr.");
        action1.setText("Buy");
        action1.setOnAction(e -> traderBuy(trader));
        action2.setText("Ignore");
        action2.setOnAction(e -> {
            travel = true;
            finishEncounter();
        });
        action3.setText("Rob");
        action3.setOnAction(e -> traderRob(trader));
        action4.setText("Negotiate");
        action4.setOnAction(e -> traderNegotiate(trader));
    }

    /**
     * Method used to set up the buttons and labeling for when the player
     * encounters a BanditNPC
     */
    private void setupBandit() {
        bandit.startInteraction(player);
        npcType.setText(String.format("You have encountered a bandit! They "
            + "request %d credits!", bandit.getPayAmount()));
        action1.setText("Pay");
        action1.setOnAction(e -> banditPay());
        action2.setText("Flee");
        action2.setOnAction(e -> banditFlee(player));
        action3.setText("Fight");
        action3.setOnAction(e -> banditFight(player));
        action4.setText("");
        action4.setDisable(true);
    }

    /**
     * Method used to set up the buttons when the interaction is with a
     * PoliceNPC
     */
    private void setupPolice() {
        police.startInteraction(player);
        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.