

Doctor Who Ticket to Ride

Milestone 2 – Team F

Authored by: Gregory Nathan, Spencer Wright, Timothy Anderson,
Emily Richardson

March 30th, 2016

New Features

List of Features Developed:

Card Drawing Clarity:

We added in a popup window that tells the user what color card they drew, to clarify game play and improve user experience. Beforehand it was not obvious what color was drawn unless the user looked at the numbers of each cards at the bottom of the screen and compared them to what previously existed.

Real Time Instructions:

In order to further assist the user and improve quality of gameplay and experience, we provided instructions on the screen that facilitate the player through each step of their turn. If the player isn't sure what to do at a specific point during their turn, we wanted them to be able to see information that explains to them what they need to do. This is now located along the right side of the GUI.

Resolution Adjusting Graphics:

Due to past conflicts with playing the game on differing screen resolutions, we decided to make the code flexible so that the dimensions of the graphics automatically adjusted to the player's screen resolution. That way, no matter what resolution screen the user decides to run the game on, the graphics that make up the interface will show themselves appropriately.

Testing New Features:

Our new features are purely graphical, therefore typical unit tests are inadequate for these changes. In order to test our changes, we did extensive manual exploratory testing to verify that the graphical changes were successfully completed in the desired manner.

Refactoring Changes

Switch Statements

Below are a few snippets of switch statements (there were 7 total that needed fixed) that were refactored during Milestone two. The repetitive code within these statements were fixed by querying a HashMap for the desired data. Due to the nature of the game, there is no reason that there would ever be new types of data added to these statements, so it is safe to use a non-expandable data structure to resolve this code smell.

Before (in class Path):

```
public Color getPathColor() {
    switch (pathColor) {
        case Red: {
            return Color.RED;
        }
        case Pink: {
            return Color.PINK;
        }
        case Orange: {
            return Color.ORANGE;
        }
        case Yellow: {
            return Color.YELLOW;
        }
        case Green: {
            return Color.GREEN;
        }
        case Blue: {
            return Color.BLUE;
        }
        case White: {
            return Color.WHITE;
        }
        case Black: {
            return Color.BLACK;
        }
        case Rainbow: {
            return Color.GRAY;
        }
    }
    return null;
}

public Color getOwnedColor() {
    if (this.ownedColor != null) {
        switch (ownedColor) {
            case Green: {
                return Color.GREEN;
            }
            case Blue: {
                return Color.BLUE;
            }
            case Magenta: {
                return Color.MAGENTA;
            }
            case Red: {
                return Color.RED;
            }
            case Yellow: {
                return Color.YELLOW;
            }
        }
        return null;
    } else {
        return null;
    }
}
```

After refactoring:

```
private static HashMap<TrainColor, Color> pathColorMap = new HashMap<TrainColor, Color>();
private static HashMap<PlayerColor, Color> playerColorMap = new HashMap<PlayerColor, Color>();

static {
    pathColorMap.put(TrainColor.Red, Color.RED);
    pathColorMap.put(TrainColor.Pink, Color.PINK);
    pathColorMap.put(TrainColor.Orange, Color.ORANGE);
    pathColorMap.put(TrainColor.Yellow, Color.YELLOW);
    pathColorMap.put(TrainColor.Green, Color.GREEN);
    pathColorMap.put(TrainColor.Blue, Color.BLUE);
    pathColorMap.put(TrainColor.White, Color.WHITE);
    pathColorMap.put(TrainColor.Black, Color.BLACK);
    pathColorMap.put(TrainColor.Rainbow, Color.GRAY);



    playerColorMap.put(PlayerColor.Red, Color.RED);
    playerColorMap.put(PlayerColor.Yellow, Color.YELLOW);
    playerColorMap.put(PlayerColor.Green, Color.GREEN);
    playerColorMap.put(PlayerColor.Blue, Color.BLUE);
    playerColorMap.put(PlayerColor.Magenta, Color.MAGENTA);
}

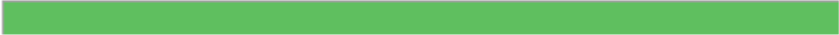
public Color getPathColor() {
    if(pathColor != null){
        return pathColorMap.get(pathColor);
    }
    return null;
}

public Color getOwnedColor() {
    if (this.ownedColor != null) {
        return playerColorMap.get(ownedColor);
    }
    return null;
}
```


Unit tests:

Test showing colors are selected correctly still passes:

Runs: 1/1  Errors: 0  Failures: 0



doctorWhoGame.PathTester [Runner: JUnit 4] (0.006 s)

-  testColorsReturnedCorrectlyForInputTrainColors (0.006 s)

Duplicated Code

There were instances where methods were exceptionally long due to unnecessary code duplication. In cases such as these, the repetitive code was extracted into its own method, and that method was then called in place of the extracted code. Show below is an example.

Before (in class PathComponent):

```
public void checkHighlight(int mouseX, int mouseY) {
    if (!purchasing && !routeGetting) {
        float xBox = mouseX - LINE_WIDTH;
        float yBox = mouseY - LINE_WIDTH;
        boolean found = false;
        for (Path p : pathArray) {
            if (p.getOwnedColor() == null && !Game.getCurrentPlayer().isPathOwned(p)) {
                Line2D.Double pathLine = p.getLine();
                if (pathLine.intersects(xBox, yBox, 2 * LINE_WIDTH, 2 * LINE_WIDTH)) {
                    if (!found) {
                        p.setHighlighted(true);
                        this.removeAll();
                        this.revalidate();
                        this.repaint();
                        found = true;
                    } else {
                        p.setHighlighted(false);
                        this.removeAll();
                        this.revalidate();
                        this.repaint();
                        break;
                    }
                } else {
                    boolean tempHighlight = p.getHighlighted();
                    p.setHighlighted(false);
                    if (tempHighlight == true) {
                        this.removeAll();
                        this.revalidate();
                        this.repaint();
                        break;
                    }
                }
            } else {
                p.setHighlighted(false);
            }
        }
    }
}
```

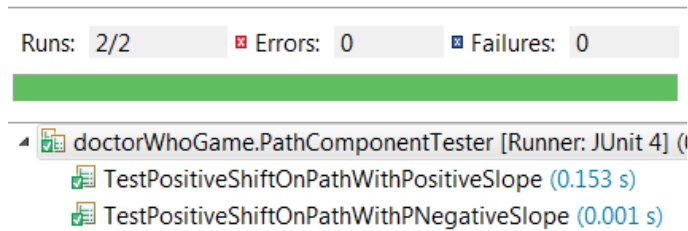
After refactoring:

```
public void checkHighlight(int mouseX, int mouseY) {
    if (!purchasing && !routeGetting) {
        float xBox = mouseX - LINE_WIDTH;
        float yBox = mouseY - LINE_WIDTH;
        boolean found = false;
        for (Path p : pathArray) {
            if (p.getOwnedColor() == null && !Game.getCurrentPlayer().isPathOwned(p)) {
                Line2D.Double pathLine = p.getLine();
                if (pathLine.intersects(xBox, yBox, 2 * LINE_WIDTH, 2 * LINE_WIDTH)) {
                    if (!found) {
                        p.setHighlighted(true);
                        removeRevalidateRepaint();
                        found = true;
                    } else {
                        p.setHighlighted(false);
                        removeRevalidateRepaint();
                        break;
                    }
                } else {
                    boolean tempHighlight = p.getHighlighted();
                    p.setHighlighted(false);
                    if (tempHighlight == true) {
                        removeRevalidateRepaint();
                        break;
                    }
                }
            } else {
                p.setHighlighted(false);
            }
        }
    }
}

private void removeRevalidateRepaint() {
    this.removeAll();
    this.revalidate();
    this.repaint();
}
```

Unit tests:

All tests still pass :

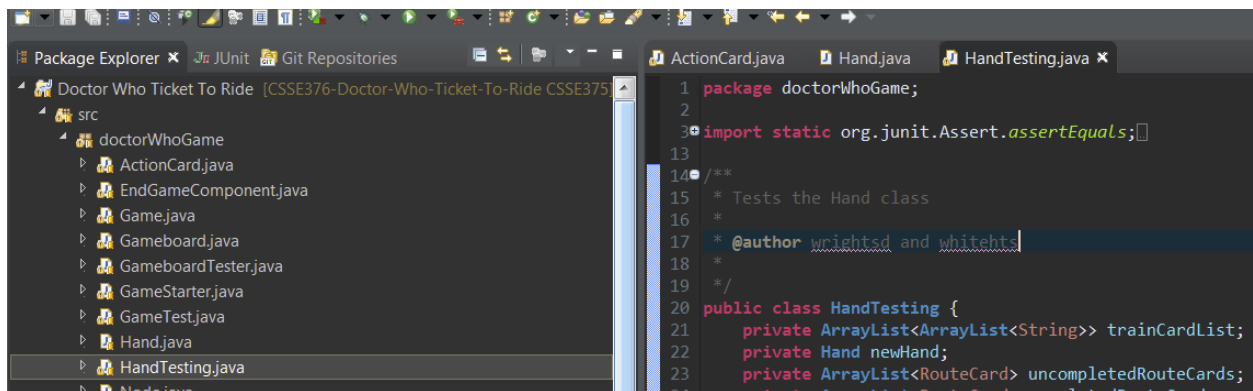


This refactoring of remove, revalidate, and repaint into its own method to reduce duplicate code was performed across several of the component classes.

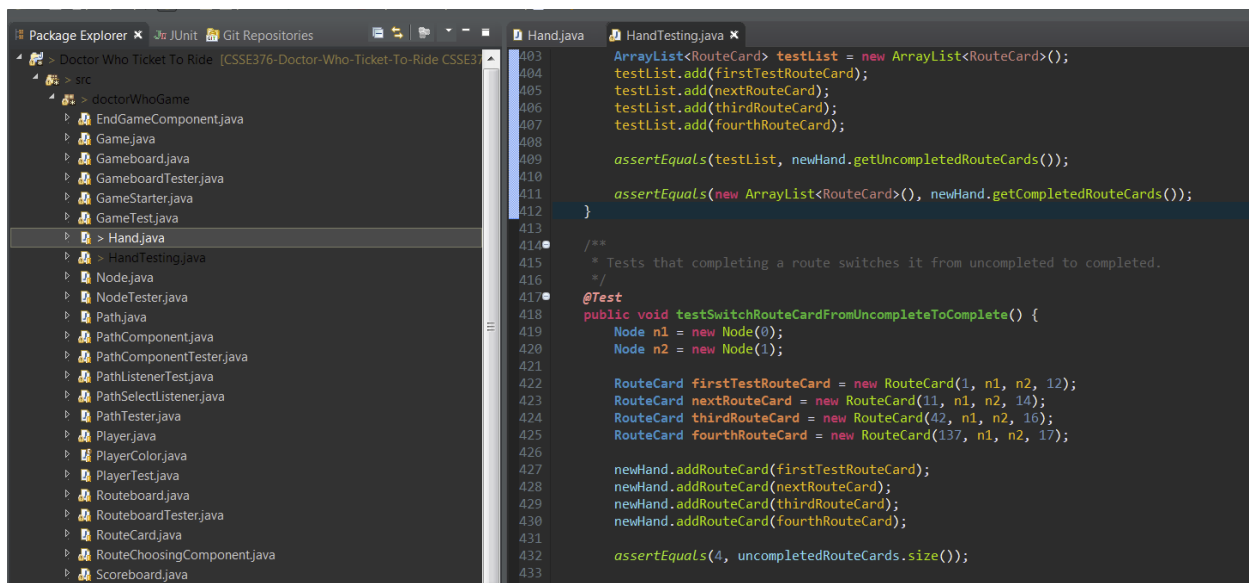
Speculative Generality

Originally, it was intended that action cards would be part of the game. Action cards are a new addition to the Ticket To Ride game that would do things, such as skip someone's turn or block another player. They were not implemented, but the ActionCard class and some calls to it were created before for future use. These are Speculative Generality, and as such, were deleted.

The code before the deletion of ActionCard can be seen below:



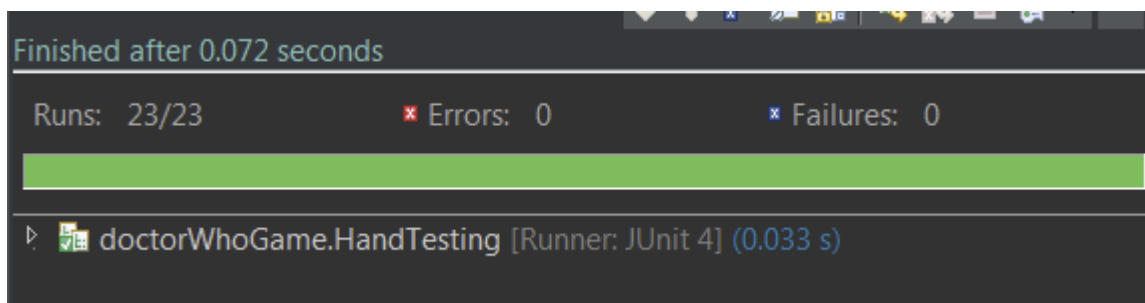
The code after ActionCard was deleted is shown below, showing that ActionCard is no longer a class, and there are no errors from its deletion:



The screenshot shows an IDE with the Package Explorer on the left and the HandTesting.java file open on the right. The Package Explorer shows a project structure with a 'src' folder containing several Java files, including 'Hand.java' and 'HandTesting.java'. The 'HandTesting.java' file is selected, and its code is visible in the editor. The code includes imports for 'ArrayList' and 'RouteCard', and a test method 'testSwitchRouteCardFromUncompleteToComplete()' that uses 'newHand' to add route cards and check their status.

```
403 ArrayList<RouteCard> testList = new ArrayList<RouteCard>();
404 testList.add(firstTestRouteCard);
405 testList.add(nextRouteCard);
406 testList.add(thirdRouteCard);
407 testList.add(fourthRouteCard);
408
409 assertEquals(testList, newHand.getUncompletedRouteCards());
410
411 assertEquals(new ArrayList<RouteCard>(), newHand.getCompletedRouteCards());
412 }
413
414 /**
415  * Tests that completing a route switches it from uncompleted to completed.
416  */
417 @Test
418 public void testSwitchRouteCardFromUncompleteToComplete() {
419     Node n1 = new Node(0);
420     Node n2 = new Node(1);
421
422     RouteCard firstTestRouteCard = new RouteCard(1, n1, n2, 12);
423     RouteCard nextRouteCard = new RouteCard(11, n1, n2, 14);
424     RouteCard thirdRouteCard = new RouteCard(42, n1, n2, 16);
425     RouteCard fourthRouteCard = new RouteCard(137, n1, n2, 17);
426
427     newHand.addRouteCard(firstTestRouteCard);
428     newHand.addRouteCard(nextRouteCard);
429     newHand.addRouteCard(thirdRouteCard);
430     newHand.addRouteCard(fourthRouteCard);
431
432     assertEquals(4, uncompletedRouteCards.size());
433 }
```

The tests that were affected all still pass, as Hand was the only other place ActionCard was referenced:



The screenshot shows a test runner output window. At the top, it says 'Finished after 0.072 seconds'. Below that, it shows 'Runs: 23/23', 'Errors: 0', and 'Failures: 0'. A green progress bar is visible. At the bottom, it shows the test class 'doctorWhoGame.HandTesting' with a green checkmark icon, indicating that all tests passed successfully.

```
Finished after 0.072 seconds
Runs: 23/23      Errors: 0      Failures: 0
doctorWhoGame.HandTesting [Runner: JUnit 4] (0.033 s)
```


Long Methods / Interleaved Logic

The Game class had many methods that tried to do too many things. These were cut up into separate methods where applicable. This also led to a little bit of removal of duplicate code as an added bonus.

An example of a method that was split up is the main constructor, shown before the refactoring below:

```
public Game(Player[] givenPlayerList, Gameboard givenGameboard, Scoreboard givenScoreBoard,
    Routeboard givenRouteboard, JLayeredPane givenLayeredPane, RouteChoosingComponent givenRouteBuyingScreen,
    TurnShield blockScreen, ArrayDeque<RouteCard> routes, EndGameComponent endGameScreen) {
    if (givenPlayerList != null) {
        ArrayList<Player> playerArrayList = new ArrayList<Player>();
        for (int i = 0; i < givenPlayerList.length; i++) {
            playerArrayList.add(givenPlayerList[i]);
            if (givenPlayerList.length > 2) {
                playerArrayList.get(i).setTrainCount(45 - (givenPlayerList.length - 2) * 5);
            } else {
                playerArrayList.get(i).setTrainCount(45);
            }
        }
        this.playerList = playerArrayList;
        this.currentPlayer = this.playerList.get(0);
    }
    for (int i = 0; i < 4; i++) {
        currentPlayer.getHand().addTrainCard(TrainDeck.draw());
    }
}
```

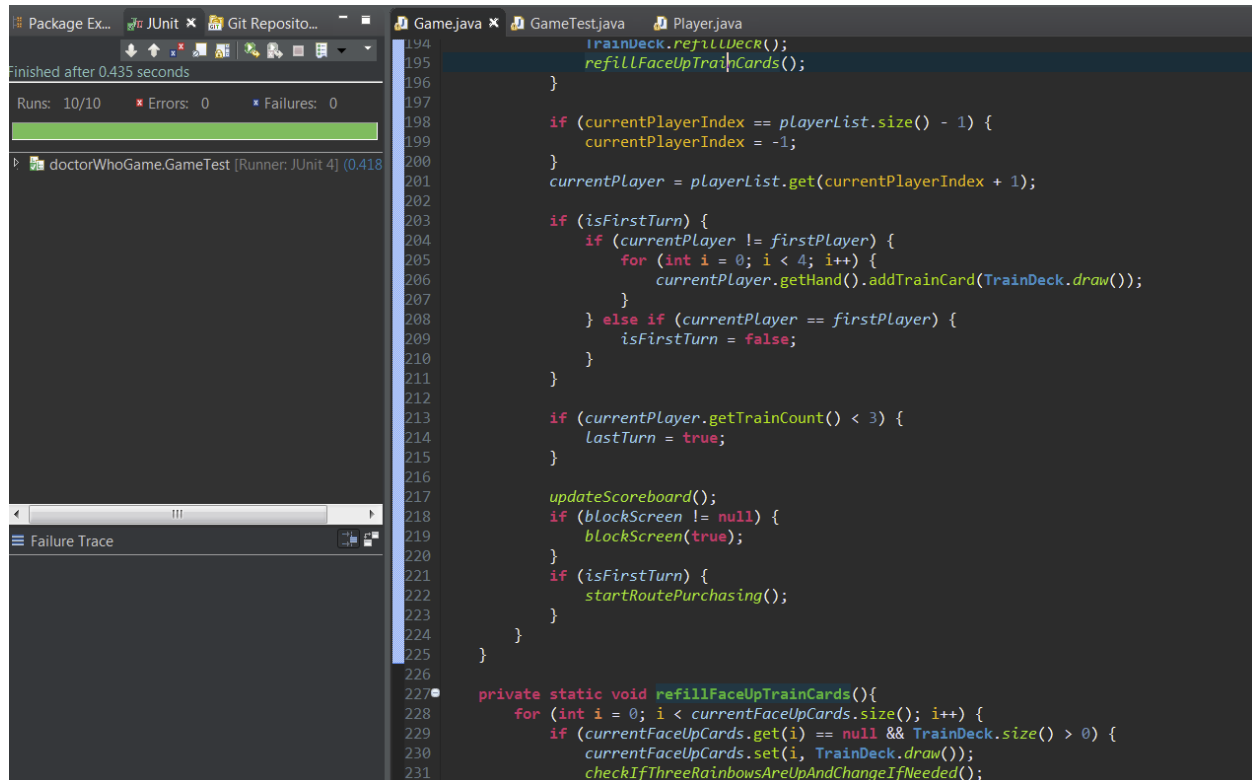
The constructor after refactoring is shown below:

```
    }

    public Game(Player[] givenPlayerList, Gameboard givenGameboard, Scoreboard givenScoreBoard,
        Routeboard givenRouteboard, JLayeredPane givenLayeredPane, RouteChoosingComponent givenRouteBuyingScreen,
        TurnShield blockScreen, ArrayDeque<RouteCard> routes, EndGameComponent endGameScreen)
    {
        ArrayList<Player> playerArrayList = initializePlayerList(givenPlayerList);

        this.playerList = playerArrayList;
        this.currentPlayer = this.playerList.get(0);
        for (int i = 0; i < 4; i++) {
            currentPlayer.getHand().addTrainCard(TrainDeck.draw());
        }
    }
}
```

A picture was not taken of the tests passing at that time, but a picture was taken of the tests passing after another refactor where refilling the faceup train card field was pulled into its own method. A picture of the tests passing is shown below:



The screenshot shows an IDE with two main panels. The left panel displays the results of a JUnit test run for `doctorWhoGame.GameTest`. It indicates that the test finished after 0.435 seconds, with 10/10 runs successful, 0 errors, and 0 failures. The right panel shows the source code for `Game.java`, with line numbers 194 through 231 visible. The code includes a `refillFaceUpTrainCards()` method that iterates over `currentFaceUpCards` and draws new cards from the `TrainDeck` if the current card is null.

```
194    trainDeck.refillDeck();
195    refillFaceUpTrainCards();
196  }
197
198  if (currentPlayerIndex == playerList.size() - 1) {
199    currentPlayerIndex = -1;
200  }
201  currentPlayer = playerList.get(currentPlayerIndex + 1);
202
203  if (isFirstTurn) {
204    if (currentPlayer != firstPlayer) {
205      for (int i = 0; i < 4; i++) {
206        currentPlayer.getHand().addTrainCard(trainDeck.draw());
207      }
208    } else if (currentPlayer == firstPlayer) {
209      isFirstTurn = false;
210    }
211  }
212
213  if (currentPlayer.getTrainCount() < 3) {
214    lastTurn = true;
215  }
216
217  updateScoreboard();
218  if (blockScreen != null) {
219    blockScreen(true);
220  }
221  if (isFirstTurn) {
222    startRoutePurchasing();
223  }
224  }
225  }
226
227  private static void refillFaceUpTrainCards(){
228    for (int i = 0; i < currentFaceUpCards.size(); i++) {
229      if (currentFaceUpCards.get(i) == null && TrainDeck.size() > 0) {
230        currentFaceUpCards.set(i, TrainDeck.draw());
231        checkIfThreeRainbowsAreUpAndChangeIfNeeded();
```

There were many more examples of this in the Game class, enough that anything that had to do with the deck of RouteCard objects was drawn out into its own class.

Long Methods / Interleaved Graphics

One severe perpetrator of the long method code smell is the `routesDisplay` method in `Gameboard`. In order to help alleviate this, several sections of code will be pulled out into their own method. One case where this can be done most successfully is with the drawing of the button to access the two route card lists. The code is essentially the same for the addition of both buttons, so a method will be extracted with parameters to handle the differences. The initial code with the nearly duplicated sections is shown below.

```
Color backColor;
if (showCompletedRoutes) {
    routesToShow = Game.getCurrentPlayer().getHand().getCompletedRouteCards();
    backColor = Color.GREEN;

    JButton switchToOtherRoutes = new JButton("View Uncompleted Routes");
    switchToOtherRoutes.setBounds(this.getWidth() - 450, 0, 250, 20);
    this.add(switchToOtherRoutes);
    switchToOtherRoutes.addActionListener(new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent arg0) {
            showCompletedRoutes = false;
            startingRouteIndex = 0;
            removeRevalidateRepaint();
        }
    });
} else {
    routesToShow = Game.getCurrentPlayer().getHand().getUncompletedRouteCards();
    backColor = Color.RED;

    JButton switchToOtherRoutes = new JButton("View Completed Routes");
    switchToOtherRoutes.setBounds(this.getWidth() - 450, 0, 250, 20);
    this.add(switchToOtherRoutes);
    switchToOtherRoutes.addActionListener(new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent arg0) {
            showCompletedRoutes = true;
            startingRouteIndex = 0;
            removeRevalidateRepaint();
        }
    });
}
```

The extracted method with proper method calls is shown below.

```
if (showCompletedRoutes) {
    routesToShow = Game.getCurrentPlayer().getHand().getCompletedRouteCards();
    backColor = Color.GREEN;

    addOppositeRouteCardAccessButton("View Uncompleted Routes", true);
} else {
    routesToShow = Game.getCurrentPlayer().getHand().getUncompletedRouteCards();
    backColor = Color.RED;

    addOppositeRouteCardAccessButton("View Completed Routes", false);
}

private void addOppositeRouteCardAccessButton(String buttonLabel, boolean showCompletedRoutesCheck){
    JButton switchToOtherRoutes = new JButton(buttonLabel);
    switchToOtherRoutes.setBounds(this.getWidth() - 450, 0, 250, 20);
    this.add(switchToOtherRoutes);
    switchToOtherRoutes.addActionListener(new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent arg0) {
            showCompletedRoutes = showCompletedRoutesCheck;
            startingRouteIndex = 0;
            removeRevalidateRepaint();
        }
    });
}
```

By extracting this method, the routesDisplay method is shortened by about 20 lines of code and the functionality of the extracted code is much more clearly expressed in the new method.

Shotgun Surgery

In order to update the positioning of the arrows and purchase count indicators when trying to purchase a route, the value would need to be changed in a large number of places, with different contributing factors, as shown below.

After refactoring to use explaining variables and a query method, the change would only be necessary in one location, either the variable or the query as shown.

```
int upArrowVerticalPlacement = CARD_SPACING_TOP;
int downArrowVerticalPlacement = (int) (CARD_SPACING_TOP + (2.0 / 3) * CARD_SPACE_HEIGHT);
int purchasingCountVerticalPlacement = (int) (CARD_SPACING_TOP + (1.0 / 3) * CARD_SPACE_HEIGHT);
int purchaseWidth = CARD_SPACE_WIDTH / 3;

if (placement == (colorArray.length - 1)) {
    for (int i = 0; i < this.colorArray.length - 1; i++) {
        JButton upArrowButton = new JButton(upA);
        upArrowButton.setBorder(BorderFactory.createEmptyBorder());
        upArrowButton.setBackground(colorArray[i]);
        upArrowButton.setBounds(getHorizontalPurchasingPlacement(i), upArrowVerticalPlacement, purchaseWidth,
                                purchaseWidth);
        this.add(upArrowButton);

        JButton downArrowButton = new JButton(downA);
        downArrowButton.setBorder(BorderFactory.createEmptyBorder());
        downArrowButton.setBackground(colorArray[i]);
        downArrowButton.setBounds(getHorizontalPurchasingPlacement(i), downArrowVerticalPlacement,
                                purchaseWidth, purchaseWidth);
        this.add(downArrowButton);
    }

private int getHorizontalPurchasingPlacement(int locationIndex){
    CARD_SPACE_WIDTH * (locationIndex) + CARD_SPACING_SIDE * (locationIndex+1)
    + CARD_SPACE_WIDTH / 3
}
```

Lazy Classes

Although these were not strictly classes, the PlayerColor enum and TrainColor enum were both created as public enums. However, these just provided an unnecessary indirection to java.awt.Color, and in addition to providing a means of indirection that was unnecessary, there was not a consistent method of accessing the color and converting from PlayerColor to java.awt.Color. The same was true for TrainColor.

Pictured below are the PlayerColor and TrainColor enum classes. These were removed, and all references changed to Color.

```
PlayerColor.java  *Player.java  *GameStarte...  *PlayerTest...
1 package doctorWhoGame;
2
3 /*
4  * The enumeration for the different player colors.
5  */
6 public enum PlayerColor {
7
8     Green, Red, Blue, Magenta, Yellow
9
10 }
11
```

```
TrainColor.java
1 package doctorWhoGame;
2
3 public enum TrainColor {
4
5     Pink, White, Blue, Yellow, Orange, Black, Red, Green, Rainbow
6
7 }
8
```

After these were removed, all instances of them were changed to refer to the Color that they represented. As an example, here is the testSetup() method for PlayerTest:

```
@Before
public void testSetup()
    throws NoSuchFieldException, SecurityException, IllegalArgumentException,
        IllegalAccessException {
    String name = "testPlayer";
    PlayerColor playerColor = PlayerColor.Green;
    this.testPlayer = new Player(name, playerColor);
    Field testPlayerTrainCount = Player.class.getDeclaredField("trainCount");
    testPlayerTrainCount.setAccessible(true);
    testPlayerTrainCount.set(testPlayer, 45);
}
```

This was changed to:

```
public void testSetup()
    throws NoSuchFieldException, SecurityException {
    String name = "testPlayer";
    Color playerColor = Color.GREEN;
    this.testPlayer = new Player(name, playerColor);
    Field testPlayerTrainCount = Player.class.getDeclaredField("testPlayerTrainCount");
    testPlayerTrainCount.setAccessible(true);
    testPlayerTrainCount.set(testPlayer, 45);
}
```

In addition to these changes, there was also one small minor change. TrainColor has a Rainbow enumeration, but in the code it mapped to gray. This is a case of speculative generality, since according to the rules that was supposed to be a rainbow card instead of gray, but there was no java representation of rainbow, and creating a gradient was more work than was deemed necessary, so gray was used instead since it was easily differentiable from the other colors and was not yet used.

Feature Envy

RouteCard scoring is done within both Player and Hand. Hand only scores the completed RouteCards, while maintaining a list of both completed and incomplete RouteCards. Player scores RouteCards by getting the list of incomplete RouteCards, totalling those, and then getting the score for the completed RouteCards from Hand. This was changed so that Hand is now merely a holder of RouteCards, and the scoring is done entirely within the Player that owns said Hand.

Before (error is because Hand was refactored before Player):

```
55 public void changeScoreFromRoutes() {
56     int subtractScore = 0;
57     int addRouteScore = 0;
58     ArrayList<RouteCard> uncompletedRoutes = this.hand.getUncompletedRouteCards();
59     for (int i = 0; i < uncompletedRoutes.size(); i++) {
60         subtractScore = subtractScore + uncompletedRoutes.get(i).getPoints();
61     }
62     this.score = this.score - subtractScore;
63     this.score = this.score + this.hand.getCompletedRouteScore();
64 }
```

After:

```
55 public void changeScoreFromRoutes() {  
56     int routeScore = 0;  
57     ArrayList<RouteCard> uncompletedRoutes = this.hand.getUncompletedRouteCards();  
58     for (RouteCard card : uncompletedRoutes) {  
59         routeScore -= card.getPoints();  
60     }  
61     ArrayList<RouteCard> completedRoutes = this.hand.getCompletedRouteCards();  
62     for (RouteCard card : completedRoutes) {  
63         routeScore += card.getPoints();  
64     }  
65     this.score = this.score + routeScore;  
66 }
```

After these changes the Player is entirely responsible for the scoring from the RouteCards instead of delegating only a small portion of the scoring to the Hand class. The test for GameTest set the score for the completed RouteCards inside of Hand class for testing end of game score instead of adding RouteCards that were completed into the Hand class, so this was also refactored to add a completed RouteCard instead of just the points directly.