Milestone 2

Team 7 - 2023/03/30
Richard Kosta
Hunter Masur
Aidan Mazany
Evan Sellers

Executive Summary

As a team the goal of milestone 2 was to get our hands dirty in the codebase, learn how to work in someone else's system, and start important refactors to make the system more maintainable. Because this project was develop for a software quality assurance (SQA-CSSE376) project the codebase was already extremely clean and there are limited code “bad code smells” as alot of these are caught by Spot Bug and other tools employed for the SQA project. This ment specifically alot of our changes came in the form of structural refactoring and “bad code smells” had to be nit picky and found during structural refactoring. One of the biggest issues with the project was the coupling (divergent change + Large Class) between the game loop and the user interface, which we have started the slow and meticulous process of unwraveling these two slowly.

Administrative

To better manage our development process we have put alot of time into our ticket management (Trello). Each task is broken up into two separate categories either a milestone for the ticket to be completed on, or Grabbag - these are not connected to a required feature and can be grabbed to complete at any time. From there the tasks are labeled into one (or multiple depending on scale) categories:

- **Feature** - Planned Feature that add additional features to the game.
- **Bad Code Smell** - These are bad code smells which must be fixed to implement.
- **Refactor Change** - These are changes which are not bad code smells, are not good.
- **QoL Improvements** - These are Quality of Life Improvements, small items like renaming unclear variable names or moving a method.
- **Administrative** - These are tasks that are not code based and are admin actions.

Plans

A more detailed plan can be viewed on our ticket management system (trello).

Milestone 2

- Add Names to each user and let them select their color.
- Ability to have different maps
- Separation of Game from other game objects using MVC
- Add “Wild Cards”
- Updated User Interface
    - Provide an explanation of the game at the beginning.
    - Provide explanation dialog during the first round of each stage.
    - Provide dialog to indicate switching to the next player's turn.

Milestone 3

- Add “Secret Mission Mode”
    - Will require adding mission cards
- Further Separation of Game and User Interface
- Update User Interface
    - Reorganization play screen to better inform users.
    - Provide information for attack, indicate selected territories, draw a line between.

Milestone 4

- Perform final testing.
- Clean up documentation.
- Prepare to deliver the project.

Fixed Bad Code Smells

(The roughly ~200 unit tests we started with are still all passing in addition to the new ones - many had to be altered to accommodate structural changes to our codebase)

(1) Data Clumping Continent Attributes

This was one of the issues that needed to be addressed to be able to support the ability of adding new maps. All the data about each continent was individual and was hardcoded linked to a string. Each continent had a string, amount of territories and a bonus amount. These data attributes when everywhere to be able to calculate the total continent bonus. We decided to abstract these attributes into a class. This allow us fix multiple other things and ment the territory could rely on a continent class instead of just a string to represent the continent.
Example Tests: WorldTest.LoadedContinentAustralia(), WorldTest.LoadedContinentAfrica(), WorldTest.LoadedContinentBrazil()

```
10      static final int MAX_ATTACKS = 3;
11      static final int MAX_DEFENDS = 2;
12      static final int NORTH_AMERICAN_TERRITORIES = 9;
13      static final int SOUTH_AMERICAN_TERRITORIES = 4;
14      static final int AFRICAN_TERRITORIES = 6;
15      static final int AUSTRALIAN_TERRITORIES = 4;
16      static final int ASIAN_TERRITORIES = 12;
17      static final int EUROPEAN_TERRITORIES = 7;
18
19      static final int NORTH_AMERICAN_BONUS = 5;
20      static final int SOUTH_AMERICAN_BONUS = 2;
21      static final int AFRICAN_BONUS = 3;
22      static final int AUSTRALIAN_BONUS = 2;
23      static final int ASIAN_BONUS = 7;
24      static final int EUROPEAN_BONUS = 5;
```

```
7   public class Continent {
8
9
10      public final String name;
11      public final int territories;
12      public final int bonus;
13
14
15      public Continent(String name, int territories, int bonus) {
16          this.name        = name;
17          this.territories = territories;
18          this.bonus       = bonus;
19      }
```

Other logic was moved to the continent class that made more sense to be connected to the continent class. The values were then loaded in by the Map Loader.

(2) Switch Statment Territories Occupied in each Continent

This section of code is used to get the total amount of territories a users has in each continent, and is used to calculate the continent bonus. The increment correct field was called on each territory a player owned by the calculate continent bonus. This then used a switch statement to figure out which class local variable to update. The switch statement and the temporary local fields were removed because we were able to abstract the process using a hashmap of the territories occupied by the player on each continent. This logic was also moved to the continent class.
Example Tests: ContinentTest.testCalculateContinentBonusWithAllTerritories(), ContinentTest.testCalculateContinentBonusMissingOneTerritoryFromEachContinent(), ContinentTest.testCalculateContinentBonusWithSingleContinent()

```
155     private static void resetStaticCounts() {
156         northAmericanTerritories = 0;
157         southAmericanTerritories = 0;
158         africanTerritories = 0;
159         australianTerritories = 0;
160         asianTerritories = 0;
161         europeanTerritories = 0;
162     }
163
164     private static void incrementCorrectField(String continent) {
165         switch (continent) {
166             case "North America":
167                 northAmericanTerritories++;
168                 break;
169             case "South America":
170                 southAmericanTerritories++;
171                 break;
172             case "Africa":
173                 africanTerritories++;
174                 break;
175             case "Australia":
176                 australianTerritories++;
177                 break;
178             case "Asia":
179                 asianTerritories++;
180                 break;
181             case "Europe":
182                 europeanTerritories++;
183                 break;
184         }
185     }
```

```
22      public static int calculateContinentBonus(List<Territory> controlledTerritories, List<Continent> continents) {
23          HashMap<Continent, Integer> byContinent;
24          byContinent = Continent.getTerritoryCountByContinent(controlledTerritories);
25          int bonus = 0;
26
27          for (Continent continent : continents) {
28              if (!byContinent.containsKey(continent)) continue;
29              if (byContinent.get(continent) != continent.territories) continue;
30              bonus += continent.bonus;
31          }
32
33          return bonus;
34      }
35
36
37      private static HashMap<Continent, Integer> getTerritoryCountByContinent(List<Territory> territories) {
38          HashMap<Continent, Integer> byContinent = new HashMap<>();
39          for (Territory territory : territories) {
40              Continent continent = territory.getContinent();
41              if (!byContinent.containsKey(continent))
42                  byContinent.put(continent, 0);
43              byContinent.put(continent, byContinent.get(continent) + 1);
44          }
45          return byContinent;
46      }
```

(3) Shotgun Surgery Implement Map

One issue that needed to be address if we were going to be able to import different maps was how the data about that map moved around. There was hardcoded information in the for the user interface buttons, user interface background, territory calculations, and in the player class. So, if we wanted to update the map (or in this case) add a new map, we would of had to update all those places, instead we built a class to handle all this information. This class also loaded the data from a YAML file. With the new update to add a new map, no code needs to change you can just add a new YAML file and image.
Example Tests: WorldTest.LoadedCoordinatesWAustralia(), WorldTest.LoadedCoordinatesScandinavia(), WorldTest.LoadedCoordinatesBrazil()

```
93      public void addButtons(List<Territory> territoryList, Game game) {
94          // CHECKSTYLE:OFF
95          int[] buttonXValues = new int[]{45,190,490,170,280,390,180,290,
96                  270,290,430,310,350,820,690,860,790,810, 1020,1260,1420,
97                  1480,1320,1120,1250,1400,1060,1220,1020,1250,1420,1210,910,
98                  1120,1260,620,790,890,580,740,550,750};
99          int[] buttonYValues = new int[]{150,140,90,220,200,240,320,360,
100                 430,520,605,610,740,540,590,590,695,820, 800,630,650,780,
101                 760,100,110,110,220,225,340,320,330,410,460,480,520,
102                 140,170,230,290,260,400,400};
103         // CHECKSTYLE:ON
104         for (int i = 0; i < territoryList.size(); i++) {
105             JButton button = new TerritoryButton(messages.getString("armies") + " 0", territoryList.get(i), game);
106             button.setBounds(buttonXValues[i], buttonYValues[i],  100, 20);
107             board.add(button);
108         }
109
110         frame.setVisible(true);
111     }
```

```
96      private void addButtons(List<Territory> territoryList,
97                              List<MapLoader.Coordinate> coordinates,
98                              int width, int height, Game game) {
99          for (int i = 0; i < territoryList.size(); i++) {
100             JButton button = new TerritoryButton(messages.getString("armies") + " 0", territoryList.get(i), game);
101             int buttonWidth  = (int) (width * coordinates.get(i).getX());
102             int buttonHeight = (int) (height * coordinates.get(i).getY());
103             button.setBounds(buttonWidth, buttonHeight,  100, 20);
104             board.add(button);
105         }
106         frame.setVisible(true);
107     }
```

```
82      private void createBoard() {
83          try {
84              File file = new File(getClass().getResource("/RiskExampleMap.jpg").toURI());
85              BufferedImage image = ImageIO.read(file);
86              ImageIcon icon = new ImageIcon(image.getScaledInstance(1600, 900, Image.SCALE_SMOOTH));
87              board = new JLabel(icon);
88          } catch (Exception e) {
89              e.printStackTrace();
90          }
91      }
```

```
86      private void createBoard(BufferedImage background) {
87          try {
88              ImageIcon icon = new ImageIcon(background);
89              board = new JLabel(icon);
90          } catch (Exception e) {
91              e.printStackTrace();
92          }
93      }
```

```java
public static int calculateArmiesFromContinentBonus(List<Territory> controlledTerritories) {
    resetStaticCounts();
    Stream<String> continents = controlledTerritories.stream().map(Territory::getContinent);
    continents.forEach(Territory::incrementCorrectField);
    int total = 0;
    if (northAmericanTerritories == NORTH_AMERICAN_TERRITORIES) {
        total += NORTH_AMERICAN_BONUS;
    }
    if (southAmericanTerritories == SOUTH_AMERICAN_TERRITORIES) {
        total += SOUTH_AMERICAN_BONUS;
    }
    if (africanTerritories == AFRICAN_TERRITORIES) {
        total += AFRICAN_BONUS;
    }
    if (australianTerritories == AUSTRALIAN_TERRITORIES) {
        total += AUSTRALIAN_BONUS;
    }
    if (asianTerritories == ASIAN_TERRITORIES) {
        total += ASIAN_BONUS;
    }
    if (europeanTerritories == EUROPEAN_TERRITORIES) {
        total += EUROPEAN_BONUS;
    }
    return total;
}
```

```java
public static int calculateContinentBonus(List<Territory> controlledTerritories, List<Continent> continents) {
    HashMap<Continent, Integer> byContinent;
    byContinent = Continent.getTerritoryCountByContinent(controlledTerritories);
    int bonus = 0;

    for (Continent continent : continents) {
        if (!byContinent.containsKey(continent)) continue;
        if (byContinent.get(continent) != continent.territories) continue;
        bonus += continent.bonus;
    }

    return bonus;
}
```

(4) Long Paramater List Attack Territory

Starting in Game, the attackTerritory method in Player is called and requires 4 separate parameters to be passed in. The Player class then calls the attackTerritory method in the Territory class requiring 3 of those 4 original parameters, and that method then calls handleAttackRolls with all of those 3 parameters again. It seems that the integer arrays representing the dice rolls, originally created in Game, start to get pretty far from home being passed from method call to method call. We decided it'd be best to simplify things by creating a parameter class that can not only handle actually making the rolls, but store all of the other requisite data so only one object needs to be passed along in this sequence as opposed to 4.
Example Tests: testAttackTerritoryCapture1Integration(), testAttackTerritoryNotCapture1Integration(), testAttackTerritoryAttackerHas0Armies()

```java
public int attackTerritory(Territory attacking, Territory defending,
        int[] attackerRolls, int[] defenderRolls) throws InvalidAttackException {
    Player attackingPlayer = attacking.getOccupant();
    Player defendingPlayer = defending.getOccupant();
    if (attackingPlayer.getColor() != this.color) {
        throw new InvalidAttackException("Cannot attack with a Territory in another's control.");
    } else if (attackingPlayer.getColor() == defendingPlayer.getColor()) {
        throw new InvalidAttackException("Cannot attack own Territory.");
    }
    if (attacking.attackTerritory(defending, attackerRolls, defenderRolls)) {
        this.captureDefeatedTerritory(defendingPlayer, attacking, defending);
        return attacking.getArmies() - 1;
    }
    return 0;
}
```

```java
public int attackTerritory(AttackData data) throws InvalidAttackException {
    Player attackingPlayer = data.getAttacker().getOccupant();
    Player defendingPlayer = data.getDefender().getOccupant();
    if (attackingPlayer.getColor() != this.color) {
        throw new InvalidAttackException("Cannot attack with a Territory in another's control.");
    } else if (attackingPlayer.getColor() == defendingPlayer.getColor()) {
        throw new InvalidAttackException("Cannot attack own Territory.");
    }

    if (data.getAttacker().attackTerritory(data)) {
        this.captureDefeatedTerritory(defendingPlayer, data.getAttacker(), data.getDefender());
        return data.getAttacker().getArmies() - 1;
    }
    return 0;
}
```

```java
// Returns true if attacked territory has lost all troops
public boolean attackTerritory(Territory defender, int[] attackerRolls, int[] defenderRolls)
        throws InvalidAttackException {

    if (attackerRolls.length == 0 || defenderRolls.length == 0
            || attackerRolls.length > MAX_ATTACKS || defenderRolls.length > MAX_DEFENDS) {
        throw new InvalidAttackException("Invalid roll values");
    }
    if (attackerRolls.length >= this.armies) {
        throw new InvalidAttackException(
                "Attacker does not possess enough troops for that many rolls");
    } else if (defenderRolls.length > defender.armies) {
        throw new InvalidAttackException(
                "Defender does not possess enough troops for that many rolls");
    }
    Arrays.sort(attackerRolls);
    Arrays.sort(defenderRolls);
    handleAttackRolls(defender, attackerRolls, defenderRolls);

    return defender.armies == 0;
}
```

```java
public boolean attackTerritory(AttackData data)
        throws InvalidAttackException {
    int[] attackerRolls = data.getADice();
    int[] defenderRolls = data.getDDice();
    Territory defender = data.getDefender();
    if (attackerRolls.length == 0 || defenderRolls.length == 0
            || attackerRolls.length > MAX_ATTACKS || defenderRolls.length > MAX_DEFENDS) {
        throw new InvalidAttackException("Invalid roll values");
    }
    if (attackerRolls.length >= this.armies) {
        throw new InvalidAttackException(
                "Attacker does not possess enough troops for that many rolls");
    } else if (defenderRolls.length > defender.armies) {
        throw new InvalidAttackException(
                "Defender does not possess enough troops for that many rolls");
    }
    Arrays.sort(attackerRolls);
    Arrays.sort(defenderRolls);
    handleAttackRolls(data);

    return defender.armies == 0;
}
```

(5) Duplicated Code in Player Constructor

Player makes use of multiple constructors mostly for testing purposes to allow the system of choosing player names and colors to be bypassed when it is so desired. In this case, the constructors are nested in order to prevent code duplication and make it easier to change Player in the future since only one constructor would need to be modified.
Example Tests: testSetPlayers(), testDefaultPlayers()

```
22          public Player(PlayerColor color, Random random, CardTrader cardTrader) {
23                  this.color = color;
24                  this.name = color.toString();
25                  this.random = random;
26                  this.cardTrader = cardTrader;
27                  this.occupiedTerritories = new ArrayList<Territory>();
28                  this.cards = new HashSet<Card>();
29                  this.armiesAvailable = 0;
30                  this.capturedThisTurn = false;
31          }
32          public Player(PlayerColor color, String name, Random random, CardTrader cardTrader) {
33                  this.color = color;
34                  this.name = name;
35                  this.random = random;
36                  this.cardTrader = cardTrader;
37                  this.occupiedTerritories = new ArrayList<Territory>();
38                  this.cards = new HashSet<Card>();
39                  this.armiesAvailable = 0;
40                  this.capturedThisTurn = false;
41          }
```

```
22          public Player(PlayerColor color, Random random, CardTrader cardTrader) {
23                  this(color, color.toString(), random, cardTrader);
24          }
25          public Player(PlayerColor color, String name, Random random, CardTrader cardTrader) {
26                  this.color = color;
27                  this.name = name;
28                  this.random = random;
29                  this.cardTrader = cardTrader;
30                  this.occupiedTerritories = new ArrayList<Territory>();
31                  this.cards = new HashSet<Card>();
32                  this.armiesAvailable = 0;
33                  this.capturedThisTurn = false;
34          }
```

(6) Long Class Game Class

Game class is storing all of the data for every relevant game object in one class, and performing operations with them that should not be the responsibility of Game. To fix this we relocated methods to different models and controllers where it seemed more appropriate. Below are some examples of methods after they were relocated to separate controllers.
Example Tests: testRemoveDefeatedPlayerFromGameLoop0(), testNextTurn(), testInitGamePlayerArray()

```
public void nextPlayer() {
    if (++currentPlayer == playerArray.size()) {
        currentPlayer = 0;
    }
}
```

```
void nextPlayer() {
    playerController.nextPlayer();
}
```

```
public void removeDefeatedPlayer(int playerNum) {
    Player currentPlayerObject = playerArray.get(currentPlayer);
    if (playerNum < 0 || playerNum > numberOfPlayers - 1) {
        throw new IllegalArgumentException("player number must be between "
                + "0 and the number of players minus one");
    }
    playerArray.remove(playerNum);
    numberOfPlayers--;
    currentPlayer = playerArray.indexOf(currentPlayerObject);
}
```

```
public void removeDefeatedPlayer(int playerNum) throws IllegalArgumentException {
    playerController.removeDefeatedPlayer(playerNum);
}
```

(7) Divergent Change in Game

The Game class directly calls fields in the data objects that stores. This leaves the two classes Game and Player highly coupled and means that if the data structure in Player was changed, both classes would need to change. We added controllers for Player, Territory, and Continent so that Game calls so that if we change how those classes work, Game will not have to change any of its method calls. Below Is an example. If we were to change how we

were selecting a player, such as if it were random or changes to be stored as a linked list, Game no longer has to change as it is handled by the controller.
Example Tests: testNextTurn(), testSetPlayers(), testSetFirstPlayer()

```
void nextPlayer() {
    if (++current == playerArray.size()) {
        current = 0;
    }
}
```

```
  void nextPlayer() {
      playerController.nextPlayer();
  }
```

```
  public Player getCurrentPlayer() {
      return playerArray.get(current);
  }
```

```
public Player getCurrentPlayer() {
    return playerController.getCurrentPlayer();
}
```

(8) Long Method to trade in cards

This code handles both the logic of determining whether a selected set of cards is valid, and awarding armies to the player (including any bonuses from those specific cards) if it is. Seeing as we were implementing wild cards that can take the place of any card type, it was only getting bigger with our addition and seemed prime for a refactor. This method was quite long and contained multiple distinct sections that were doing different things, so we extracted the separate sections into multiple private methods with descriptive names. As you can see from the below images, the overall logic of tradeInCardSet(...) is now much clearer. We also got rid of the switch statement as an added bonus.
Example Tests: CardTraderTest.testValidCombinationOneEachWithWildCard(), CardTraderTest.testValidCombinationMultipleWildCards(), CardTraderTest.testInvalidSizeIncludingWildCard()

```
public boolean tradeInCardSet(Player player, Set<Card> cardSet) {
    if (cardSet.size() != 3) {
        return false;
    }
    int numInfantry = 0;
    int numCavalry = 0;
    int numArtillery = 0;

    boolean isValid = false;
    for (Card card : cardSet) {
        switch (card.getSymbol()) {
            case INFANTRY:
                numInfantry++;
                break;
            case CAVALRY:
                numCavalry++;
                break;
            case ARTILLERY:
                numArtillery++;
                break;
```

```
public boolean tradeInCardSet(Player player, Set<Card> cardSet) {
        if (this.cardSetIsValid(cardSet)) {
                if (this.getsTerritoryBonus(player, cardSet)) {
                        player.giveArmies(2);
                }
                player.giveArmies(this.getCurrentSetValue());
                this.numSetsTurnedIn++;
                return true;
        }
        return false;
}
```

```java
private boolean cardSetIsValid(Set<Card> cardSet) {
        if (cardSet.size() != 3) {
                return false;
        }
        CardSymbol[] setSymbols = new CardSymbol[3];

        int index = 0;
        for (Card card : cardSet) {
                CardSymbol currentSymbol = card.getSymbol();
                if (currentSymbol == CardSymbol.WILD) {
                        return true;
                }
                setSymbols[index] = currentSymbol;
                index++;
        }
        boolean isValid = cardSetIsThreeOfAKind(setSymbols) || cardSetIsOneOfEach(setSymbols);
        return isValid;
}
```

(9) Speculative Generality for World Loader

This World/Map load had an interface. But this interface was only for type of loader the YAML loader, and we have no plans of adding different types of world loaders. This was actually my fault I was thinking back to software design and thought for future changes we would want this, but we have no plans to add any features like this in the future so it was just complicated bloat, and I reconized that and took it out. This was done by simply merging removing the interface and moving the static method from the interface to the new World class.
Example Tests: WorldTest.GetMaps(), WorldTest.getBackground()

```java
6
7   public interface MapLoader {
8
```

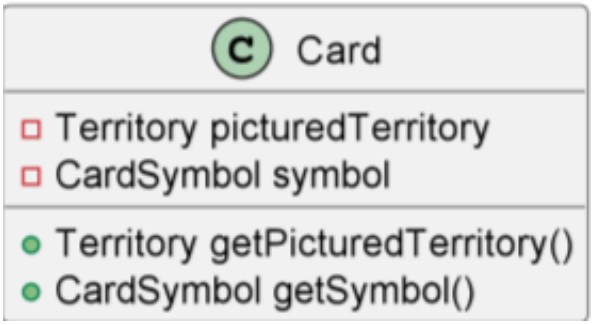
```java
13
14  public class MapLoaderYAML implements MapLoader {
15
```

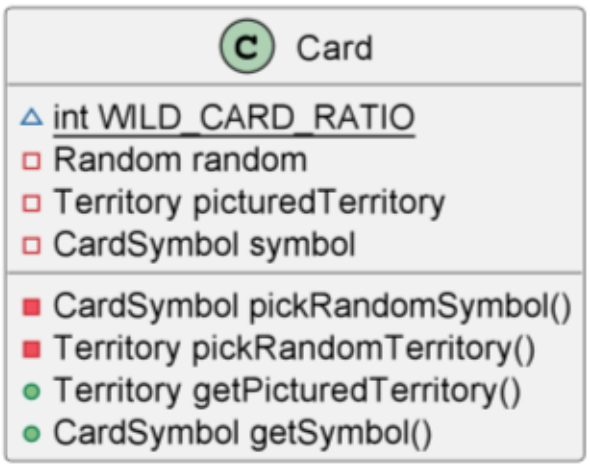
```java
10
11  public class World {
12
```

MapLoader was renamed World, because it makes more sense in the context and it's not confused with a hashmap.

(10) Card is a data class

Seeing as the CardTrader class already handled the logic of trading in card sets (as its name would suggest), it seemed off that it also handled the generation of new cards when players draw. To remedy this, we moved the generation logic inside of Card so that it could generate itself and have some actual behavior to speak of. On the left was the only functionality of Card prior to the refactor, and on the right is the refactored version.
Example Tests: CardTraderTest.generateWildCard(), all previous generation tests still passing





(11) Message Chaining Background Size

The Map Loader loads the background in for use by the user interface, but for the user interface to get the size of the background, it must request the world object, then request the background object from that, and then finally ask for the height and width. I removed this message changing by allowing someone to ask the map object for the size of the map directly.

Example Tests: WorldTest.LoadedBackground()

```
36          int width = map.getBackground().getWidth();
37          int height = map.getBackground().getHeight();
38          this.addButtons(map.getTerritories(), map.getCoordinates(),
39                  width, height, game);
```

```
36          int width = world.getWidth();
37          int height = world.getHeight();
38          this.addButtons(world.getTerritories(), world.getCoordinates(),
39                  width, height, game);
```

```
165     public int getWidth() {
166         return this.background.getWidth();
167     }
168
169
170     public int getHeight() {
171         return this.background.getHeight();
172     }
```

(12) CardTrader's generateNewCard method is a needless Middle Man

Seeing as the player simply wants to add a new card to their hand, it's strange that they need to ask the CardTrader object to generate one and give it back to them, first. This was remedied in the same way as the data class smell - by putting the power of generation into the Card class itself so as to conceal and encapsulate that behavior.

Example Tests: CardTraderTest.generateWildCard(), all previous generation tests still passing

```
            case WILD:
                isValid = true;
        }
    }
    if (!isValid) {
        isValid = (numInfantry == 3 || numArtillery == 3 || numCavalry == 3)
                || (numInfantry == 1 && numCavalry == 1 && numArtillery == 1);
    }
    if (isValid) {
        if (this.getsTerritoryBonus(player, cardSet)) {
            player.giveArmies( numArmies: 2);
        }
        player.giveArmies(this.getCurrentSetValue());
        this.numSetsTurnedIn++;
        return true;
    }
    return false;
}
```

```
private boolean cardSetIsThreeOfAKind(CardSymbol[] setSymbols) {
        boolean allSame = (setSymbols[0] == setSymbols[1]) && (setSymbols[1] == setSymbols[2]);
        return allSame;
}

private boolean cardSetIsOneOfEach(CardSymbol[] setSymbols) {
        boolean oneOfEach = (setSymbols[0] != setSymbols[1])
                        && (setSymbols[1] != setSymbols[2])
                        && (setSymbols[0] != setSymbols[2]);
        return oneOfEach;
}
```

Seeing as the player simply wants to add a new card to their hand, it's strange that they need to ask the CardTrader object to generate one and give it back to them, first. This was remedied in the same way as the data class smell - by putting the power of generation into the card class itself so that the middleman could be removed from that equation all together. Now, to draw a new card one can simply create a new card object without specifying any parameters (the Java.Random instance passed in is only for the purposes of mocking in tests)

```
void drawCard() {
        this.cards.add(this.cardTrader.generateNewCard());
}
```

```
public Card generateNewCard() {
        CardSymbol symbol = null;
        switch (this.random.nextInt(3)) {
                case 0:
                        symbol = CardSymbol.INFANTRY;
                        break;
                case 1:
                        symbol = CardSymbol.CAVALRY;
                        break;
                case 2:
                        symbol = CardSymbol.ARTILLERY;
                        break;
        }
        return new Card(this.territories.get(this.random.nextInt(this.territories.size())), symbol);
}
```

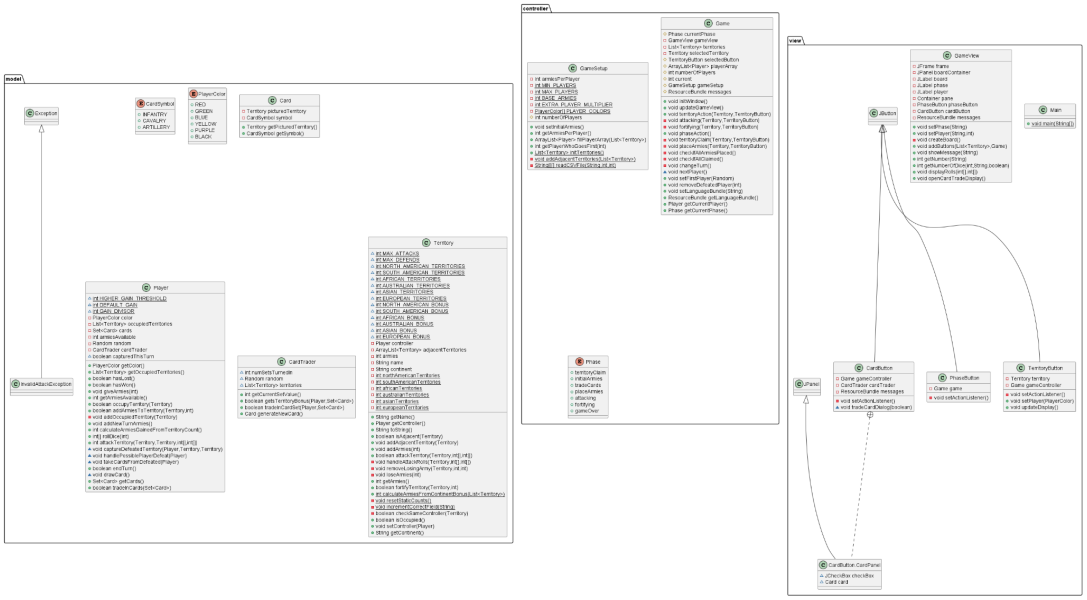
```
void drawCard() {
        this.cards.add(new Card(this.random));
}
```

Updated Artifacts

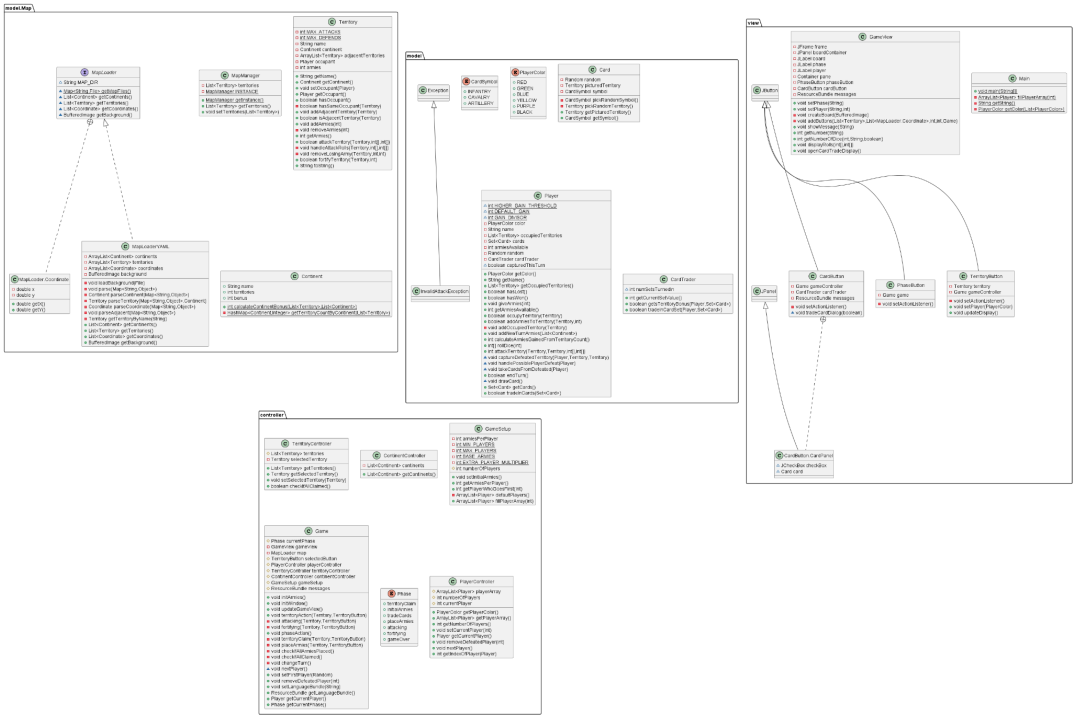
F1. Ability to set Number of Players From 2 - 6
F2. Detect when a player wins the game:
    When a player has control of all territories, they win the game
F3. Ability to place armies on territories during start of game
F4. Ability to attack adjacent territories during turn
F5. Ability to move armies to adjacent territories
F6. Ability to draw a card after capturing territory

F7. Ability to trade in cards on turn
F8. Ability to view the state of the map
F9. Detect when a player has lost the game
    When a player loses control of all of their territories they lost the game
F10. Ability to gain and place armies at the start of your turn
F11. Set number of armies given at start of game depending on number of players
F12. Ability to take cards from another player when you capture their final territory
F13. Ability to have a choice of multiple, mechanically different maps
F14. Ability to choose your color and name
F15. Ability to select alternative Risk Gamemode - Secret Mission Mode
(New requirements in bold - possibly not exhaustive if we happen to think of new ones for future milestone)

Old UML:



New UML:



Key Differences:

- Map methods were moved into their own package and support continents
- Game had a lot of responsibilities moved to controllers
- Added support for new cards and moved the generation logic into the Card class
- Methods for extra player features such as color added