Next-Gen Solitaire Tutorials

Written By:
Daniel Duff, Andrew Levy

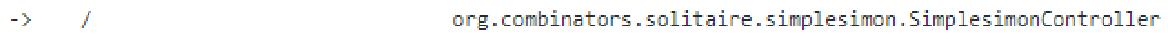
Advised By:
George T. Heineman

Creating a New Solitaire Family/Variation

1. **Setting Up Work Environment and Test Running Variations**
    a. *Install latest version of IntelliJ*
        i. Include all Scala tools when prompted
        ii. If IntelliJ is already installed, install Scala plugin
    b. *Open Next-Gen Solitaire*
        i. Open up the Next-Gen Solitaire project which can be found and cloned/downloaded at: https://github.com/combinators/nextgen-solitaire/tree/Dan-TestCases-Perturb
    c. *Configure IntelliJ*
        i. In the top right corner click the drop down and go to edit configurations
        ii. Choose to add a new SBT Task
        iii. Change the tasks field to be just "run"
        iv. Rename the command if desired and save and close configuration
        v. Got to File>install settings and select the IntelliJSettings.jar in the demo folder
    d. *Run/Test the service you created in B*
        i. In a web browser go to http://localhost:9000/narcotic (you can replace narcotic with any existing variations, note variations in families include their family name before the variation name, such as http://localhost:9000/fan/shamrocks
        ii. Wait for the page to load
            1. May take up to a minute
            2. If page never stops loading then there may be an error (check IntelliJ terminal for errors)
        iii. Press the compute button on the web page
        iv. Copy the git clone line from the git tab that appears when you compute
    e. *Run/Test generated Java code*
        i. Open a terminal and navigate to a folder where you want to store your variations
        ii. Run the git clone command
        iii. In IntelliJ open the project structure and import the generated code from the folder in step i as a new module.
        iv. Link the standalone.jar found in demo/ folder of the generating-solitaire project
        v. The code should be runnable and allow you to run any downloaded variations

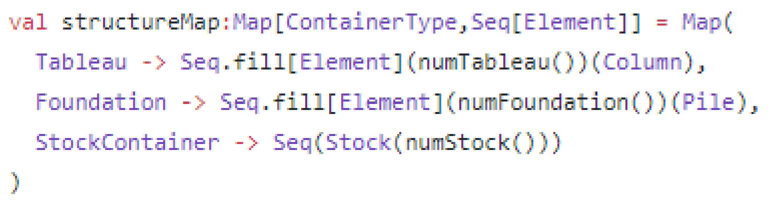
2. **Creating a New Family/Variation**
    a. *Intro*
        i. Solitaire variations can either be a part of a family or be alone as part of a standalone variation.
        ii. This section will focus on creating a family which can contain several variations based on the family.
        iii. This intro will be following the creation of the Simple Simon family.
    b. *Setup*
        i. In src/main/scala/org.combinators/solitaire/ create a folder named "simplesimon"
        ii. Inside this folder, create the following files:
            1. Controllers - Scala trait class

        a. Defines the family’s controllers and their behaviors.
    2. Simplesimon - Solitaire Target
        a. Handles variation controllers for all variations in the family..
    3. Simplesimon - Package object
        a. The Scala model for the Simple Simon variation.
        b. Pulls characteristics of the family from `variationPoints.`
    4. simplesimonDomain - Scala class
        a. Requesting additional imports for generating code.
        b. Custom constraints are defined here within `ExtraMethods.`
    5. variationPoints - Scala trait class
        a. Defines characteristics of the Simple Simon family that all variation models can pull from (for example, number of number of tableaus)
    6. These files can all be created from scratch, from the currently existing templates (File, New, scroll to find the template of each file above), or copied from other working variations and then renamed and refactored to match the new variation.
    7. Each variation that is created must be registered in the routes file located in the resources folder (src/main/resources/routes). This will allow the synthesizer to find it during generation. Each variation should be registered in routes as shown in the image below.



```
->    /                                 org.combinators.solitaire.simplesimon.SimplesimonController
```

iii. Create the family - variationPoints
    1. Structure Map
        a. Structure Map (usually called `structureMap` in `variationPoints`) is where the type of card containers are defined. For example, Simple Simon uses tableaus, foundation piles, and a stock.



```
val structureMap:Map[ContainerType,Seq[Element]] = Map(
  Tableau -> Seq.fill[Element](numTableau())(Column),
  Foundation -> Seq.fill[Element](numFoundation())(Pile),
  StockContainer -> Seq(Stock(numStock()))
)
```

        b. The tableau takes two values, the number of tableaus and the type of piles that the tableaus can be. These types include: piles, buildable piles, and columns. The pile type determines how cards are shown on the tableau, such as whether the underlying cards are visible below the top of the stack.
        c. Foundation takes the same type of values as tableau, but for Simple Simon the foundation uses piles instead of columns.
        d. All the stock needs is the number of stocks since all cards in a stock move in the same way and therefore the type does not have to be defined.
    2. Layout Map

a. Layout Map (usually called `layoutMap` or `map` in `variationPoints`) defines the location of the containers defined in the structure map.

```
val map:Map[ContainerType, Seq[Widget]] = Map (
  Tableau -> horizontalPlacement(15, 200, numTableau(), 13*card_height),
  StockContainer -> horizontalPlacement(15, 20, numStock(), card_height),
  Foundation -> horizontalPlacement(293, 20, numFoundation(), card_height)
)
```

b. In the layout map, all of the containers take an x position, y position, the number of that type of container, and the size of the containers. `card_height` represents the height of one card so that the containers are large enough to fit the cards that they are containing.

3. Deal

a. Now that the layout has been set, the next step is to deal the cards into their containers. Each game deals a different amount of cards to a different amount of locations so the `getDeal` function defines how many cards should go to each container. In the example below, 8 cards are dealt to the first two tableaus, with one less card being added to the subsequent piles. (ex: 8, 8, 8, 7, 6, …).

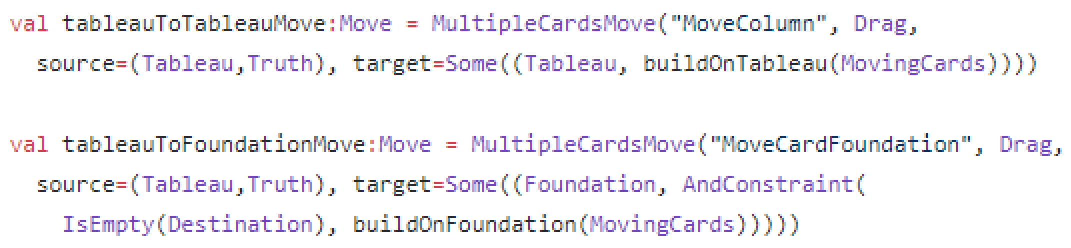
```
def getDeal: Seq[DealStep] = {
  var colNum:Int = 0
  var cardNumCounter:Int = 8
  var dealSeq:Seq[DealStep] = Seq()// doesn't like me declaring it without initializing

  for(colNum <- 0 to 1){
    dealSeq = dealSeq :+ DealStep(ElementTarget(Tableau, colNum),
      Payload(faceUp = true, numCards = 8))
  }
  for(colNum <- 2 to 9){
    dealSeq = dealSeq :+ DealStep(ElementTarget(Tableau, colNum),
      Payload(faceUp = true, numCards = cardNumCounter))
    cardNumCounter -= 1
  }
  dealSeq
}
```

b. The `faceUp` variable is a boolean that states whether or not the cards are dealt face up. The `numCards` variable states how many cards are dealt to the specified tableau. Any remaining cards not dealt in `dealStep` are placed into the stock.

c. In Simple Simon, the deal step deals 8 cards face up to tableau 0 and 1. For each remaining tableau the number of cards decreases by 1 after dealing. Simple Simon does not have any leftover cards that are placed into the stock, however the container still exists for Simple Simon variations to place cards into.

4. Moves
    a. Each game has a set of moves that are allowed within the game. These moves define which cards can be moved into which locations and when.
    b. Each move needs to be given several properties. The type of move, how the move is performed (dragging or clicking), the source and destination of the cards in the move, and a set of constraints that define if the move is allowed. Each move may take many constraints in order to be a successful move and therefore these constraints are usually defined in a helper function. Constraints will be discussed more in the next subsection.



```
val tableauToTableauMove:Move = MultipleCardsMove("MoveColumn", Drag,
  source=(Tableau,Truth), target=Some((Tableau, buildOnTableau(MovingCards))))

val tableauToFoundationMove:Move = MultipleCardsMove("MoveCardFoundation", Drag,
  source=(Tableau,Truth), target=Some((Foundation, AndConstraint(
    IsEmpty(Destination), buildOnFoundation(MovingCards)))))
```

    c. Simple Simon uses two moves, tableau to tableau and tableau to foundation. Both moves are exactly as they sound as tableau to tableau is moving cards from one tableau to another and tableau to foundation is moving cards from a tableau to a foundation.

5. Constraints
    a. Constraints can be used to check a specific condition and returns true if that condition is met. For example `isAce(card)` would return true if the defined card is an ace.
    b. Constraints can be used to define what makes a specific move valid. Using the same example, if a move can only be valid if the card is an ace, the move can check that by determining if the `isAce()` constraint returns true.
    c. There are a list of predefined constraints however since all games are very different, another constraint that does not exist may need to be created to meet a specific condition. Custom constraints can be defined in Java within the Domain in an `extraMethods` function.

```
def buildOnTableau(cards: MovingCards.type): Constraint = {
  val topDestination = TopCardOf(Destination)
  val bottomMoving = BottomCardOf(cards)
  val isEmpty = IsEmpty(Destination)
  val descend = Descending(cards)
  val suit = AllSameSuit(cards)
  AndConstraint( AndConstraint(descend, suit), OrConstraint(isEmpty,
    NextRank(topDestination, bottomMoving, true)) )
}
```

d. Simple Simon uses the two methods `buildOnTableau` and `buildOnFoundation` to check constraints for moving cards to the tableau and the foundation.
e. The tableau to tableau move only allows piles of cards to move onto another pile if the bottom card of the moving pile is the same suit and one rank below the top card of the destination pile.
f. The tableau to foundation move only allows a pile of a complete set of descending cards (King through Ace) of the same suit to be moved into an empty foundation.

iv. Create the model - package
   1. Setting model
      a. In the package folder of each variation, the model is created where each variable for a solitaire game is defined. Many of these come from `variationPoints` so it must be extended in the package.

```
val simplesimon:Solitaire = {
  Solitaire(name = "Simplesimon",
    structure = structureMap,
    layout = Layout(map),
    deal = getDeal,
    specializedElements = Seq.empty,
    moves = Seq(tableauToTableauMove, tableauToFoundationMove),
    logic = BoardState(Map(Foundation -> 52)),
    solvable = false,
    testSetup = Seq(),
  )
}
```

b. `structure, layout, deal, and moves` are the same as what is defined within `variationPoints`.
c. `logic` refers to how the user can win the game of solitaire. In the Simple simon variation example, the user wins when the game is in a board state where all 52 cards are all in the foundation.
d. `solvable` an optional value as when it is set to false then it will not be utilized in the variation. If the value is set to true, then an additional button is available to the user upon launching a variation: *solve*. This button then iterates through all possible

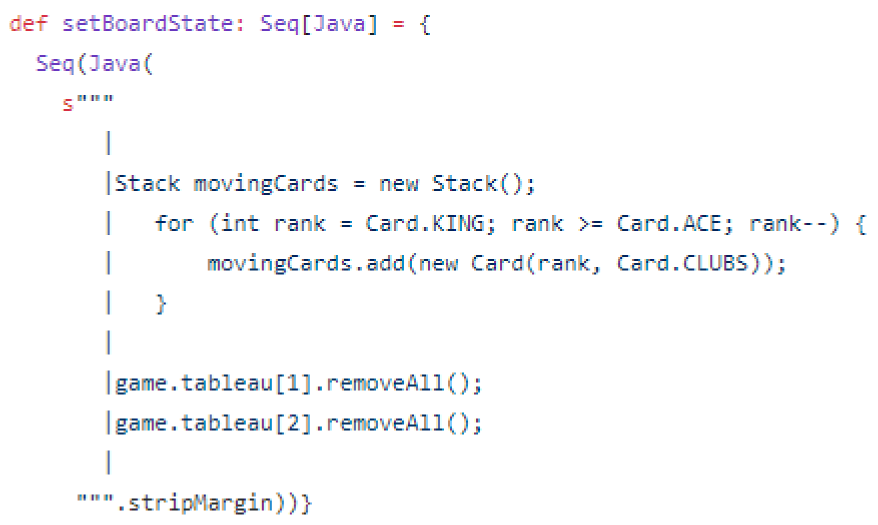
potential moves for a given board state, effectively brute-forcing all available options until the most progress is made.

e. `specializedElements` is a part of the model that is used to set special elements on the board display. Elements that are "specialized" are any elements that are not contained within the standard board components of decks, tableaus, and foundations. When set to an empty sequence, no special elements are used.

2. Test Setup - Set Board State

a. `SetBoardState` is a function that defines an example board state. This board state should represent how the cards should be set up so that moves can be tested easily after generating test cases.

b. The Simple Simon variation does not utilize this function (and as of yet does not utilize generating test cases) and therefore is just set as an empty sequence. However, the Simplevar variation in the Simple Simon family does utilize `setBoardState`. Simplevar has the same rules as Simple Simon but some of the cards begin in the stock and can be dealt to the tableau.



```
def setBoardState: Seq[Java] = {
  Seq(Java(
    s"""
       |
       |Stack movingCards = new Stack();
       |   for (int rank = Card.KING; rank >= Card.ACE; rank--) {
       |       movingCards.add(new Card(rank, Card.CLUBS));
       |   }
       |
       |game.tableau[1].removeAll();
       |game.tableau[2].removeAll();
       |
     """.stripMargin))}
```

c. Simplevar's `setBoardState` creates a scenario where the moving pile is a set of cards king to ace that are all the same suit. Then tableaus 1 and 2 are cleared. This organization of cards allows for a successful tableau to tableau move and tableau to foundation move to be tested.

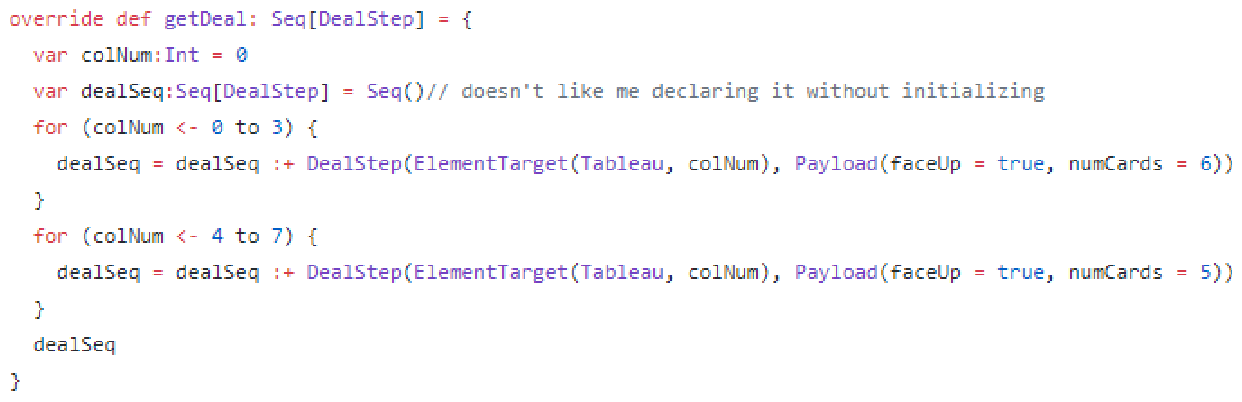
v. Controllers

1. Controller are responsible for adding combinators relating to in-game actions, such as dragging. The initial template covers most controllers that will be needed for each variation, however occasionally new controllers will need to be added to accommodate specific actions in games. For example, clicking on a buildable pile causes the top card to flip over (as shown in the spider variation controller).

c. *Adding more variations*

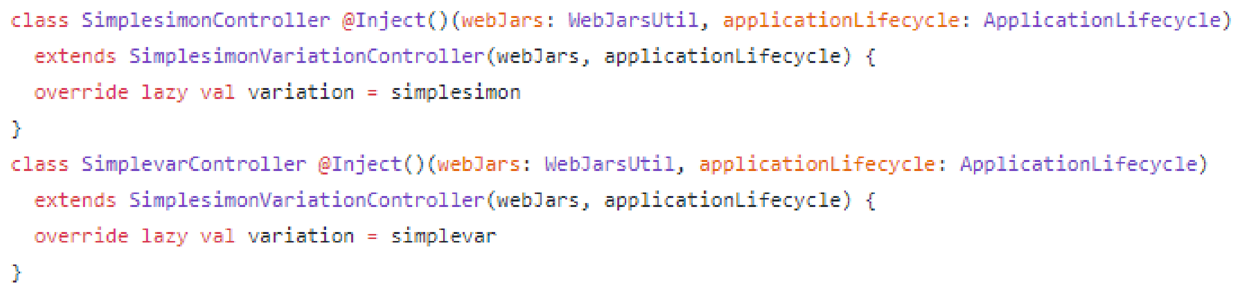
i. To add more variations to the family, create a new package named after the new variation. This variation will be modeled in the same way and will also extend `varitionPoints` to take info from the family.

ii. Since new variations are different from the variations within their families, they also have different attributes. Values can be overwritten so that the model utilizes the game mechanics of the new variation instead of those defined for the main variation (defined in `variationPoints`). The override method can be used as shown below. This override is used by Simplevar to override the number of cards dealt to the tableau at the start of a game.



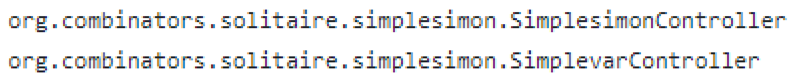
```
override def getDeal: Seq[DealStep] = {
  var colNum:Int = 0
  var dealSeq:Seq[DealStep] = Seq()// doesn't like me declaring it without initializing
  for (colNum <- 0 to 3) {
    dealSeq = dealSeq :+ DealStep(ElementTarget(Tableau, colNum), Payload(faceUp = true, numCards = 6))
  }
  for (colNum <- 4 to 7) {
    dealSeq = dealSeq :+ DealStep(ElementTarget(Tableau, colNum), Payload(faceUp = true, numCards = 5))
  }
  dealSeq
}
```

iii. New variations must have their controllers added to the family's variation controllers as shown in the image below.



```
class SimplesimonController @Inject()(webJars: WebJarsUtil, applicationLifecycle: ApplicationLifecycle)
  extends SimplesimonVariationController(webJars, applicationLifecycle) {
  override lazy val variation = simplesimon
}
class SimplevarController @Inject()(webJars: WebJarsUtil, applicationLifecycle: ApplicationLifecycle)
  extends SimplesimonVariationController(webJars, applicationLifecycle) {
  override lazy val variation = simplevar
}
```

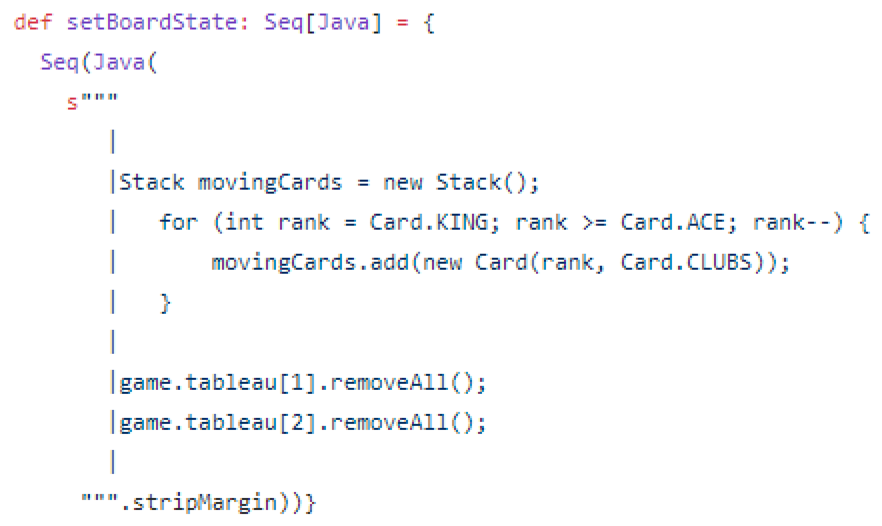
iv. It is important that any new variations be added to routes so that they can be generated as well.



```
org.combinators.solitaire.simplesimon.SimplesimonController
org.combinators.solitaire.simplesimon.SimplevarController
```

Creating New Generated Test Cases for a Solitaire Variation

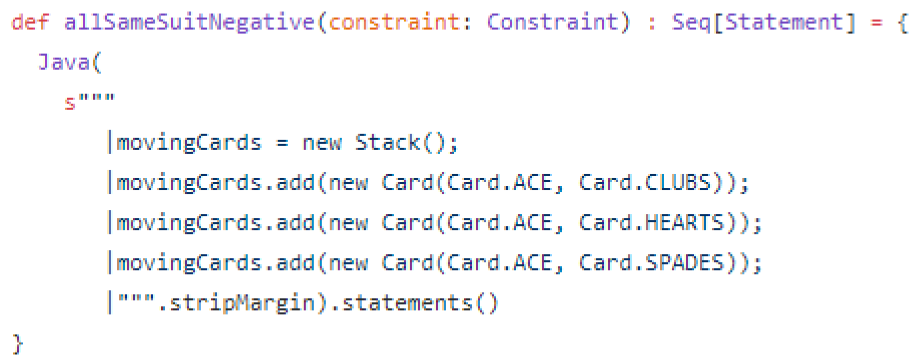
1. Set Board State
    a. The first step to creating test cases for a variation is to set an initial board state. The initial board state is an orientation of cards where the various moves of a game can be tested. This involves creating a scenario that when a stack of cards (`movingCards`) is moved it should be able to successfully move on to the tableau or the foundation. This scenario is defined within the variation's package object and is written in Java using Scala's built-in Java method



```
def setBoardState: Seq[Java] = {
  Seq(Java(
    s"""
       |
       |Stack movingCards = new Stack();
       |   for (int rank = Card.KING; rank >= Card.ACE; rank--) {
       |       movingCards.add(new Card(rank, Card.CLUBS));
       |   }
       |
       |game.tableau[1].removeAll();
       |game.tableau[2].removeAll();
       |
     """.stripMargin))}
```

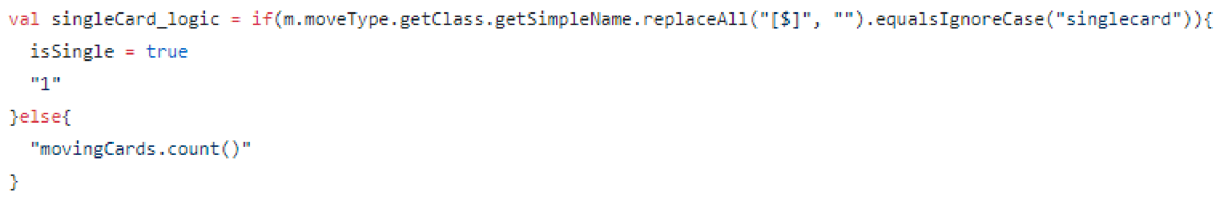
    b. In the example above (Simplevar), a scenario is created where the stack of moving cards is a stack of all clubs ranked in descending order from king to ace. Tableau 1 and 2 are cleared of cards so that the moving cards can be moved from tableau 1 to tableau 2 with no other cards interfering with the tableau to tableau move. In this scenario, all of the conditions for the Simplevar moves are met and therefore the moves are all valid.
    c. Inside the model for the variation, the `testSetup` is set to `setBoardSate` so that the variation keeps track of its valid board state and can call it when needed when generating test cases. If the model is just set to an empty sequence, then it will not generate any test cases.
    d. Once creating the `testSetup`, the variation will be able to generate test cases that pass. However, since each variation uses different constraints, moves, and structures, all of the tests may not pass if the Solitaire Test Suite is encountering a new game mechanic. These changes can be made in `UnitTestCaseGeneration.scala`.
2. Unit Test Case Generation
    a. The `SolitaireTestSuite` class inside of `UnitTestCaseGeneration.scala` generates Java unit test cases for a variation. When running the apply code registry, a default test is created that makes sure that the move being tested is valid given all of the constraints. For each constraint, another test is generated with that constraint negated. These results all should return as false, showing that each constraint must be met in order for the entirety to return true. Each possible move for the variation is run and tested in this way.

b. `UnitTestCaseGeneration.scala` defines methods that negate each constraint (for example `isAceNegative(Constraint)`). These methods have only been defined for the variations that currently generate test cases. If creating test cases for a new variation, then a new method for a negated constraint may need to be created if the new variation uses a currently unused constraint. To make a new method for a negated constraint, a scenario needs to be defined where the constraint would not be met. For example, in `allSameSuitNegative()` a set of cards is created where all of the cards are aces of different suits.



```
def allSameSuitNegative(constraint: Constraint) : Seq[Statement] = {
  Java(
    s"""
       |movingCards = new Stack();
       |movingCards.add(new Card(Card.ACE, Card.CLUBS));
       |movingCards.add(new Card(Card.ACE, Card.HEARTS));
       |movingCards.add(new Card(Card.ACE, Card.SPADES));
       |""".stripMargin).statements()
}
```

c. While running tests for new variations there may be extra additions that need to be made outside of constraints. In certain scenarios there may need to be specific checks added to update the apply method to account for other variations game mechanics. For example, the shamrocks game used movement of single cards instead of stack so a logic check was done to determine if the move was using stacks or single cards. This type of check can be done for other scenarios that have not currently been accounted for in the testing.



```
val singleCard_logic = if(m.moveType.getClass.getSimpleName.replaceAll("[$]", "").equalsIgnoreCase("singlecard")){
  isSingle = true
  "1"
}else{
  "movingCards.count()"
}
```

3. Controller

a. When running the game code for a variation the `Controller.scala` file in the shared folder will create a Solitaire Test Suite for the variation. This is located at the beginning of the `createMoveClasses` method. The method checks to see if a class has a `testSetup` (if the board state has been set) defined in the model and if so it will generate the test cases. If the `testSetup` is not defined then the method will just print that there is no set up found and there are no tests generated for the variation.

```
if(s.testSetup.nonEmpty){
  println("Test setup found for variation: " + s.name)
  updated = updated.addCombinator(new SolitaireTestSuite(s))
}else{
  println("no setup found")
}
```