```java
package Solitaire.Model;

import java.util.List;
import java.util.ArrayList;
import java.util.Random;

/* Models a playing card */

public class Card {

  /* Abs: (1) number = 1 => card is an Ace of 'suit
          (2) 2 <= number <= 10 => card is a 'number' of 'suit'
          (3) number = 11 => card is a Jack of 'suit'
          (4) number = 12 => card is a Queen of 'suit'
          (5) number = 13 => card is a King of 'suit' */

  /* DTI: (1) 1 <= number <= suitSize
          (2) color = suit = Suit.Clubs || suit. Suit.Spades */

  public final static int suitSize = 13;
  public enum Suit { Clubs, Diamonds, Hearts, Spades }

  public final int number;
  public final Suit suit;
  public final boolean color;

  // Constructor private because cards can only be obtained as part of a deck
  private Card(Suit s, int n) {
    assert n >= 1 && n <= suitSize;
    suit = s; number = n;
    color = suit == Suit.Clubs || suit == Suit.Spades;
  }

  // Post: returns [Card(s,n) | s <- Suits, n <- [1..13]]
  public static List<Card> fullDeck(){
    List<Card> deck = new ArrayList<Card>(52);
    for (Suit suit : Suit.values()) {
      for (int cardnumber = 1; cardnumber <= suitSize; cardnumber++)
        deck.add(new Card(suit, cardnumber));
    }
    return deck;
  }

  // Post: returns fullDeck() shuffled using the Fischer-Yates algorithm
  public static List<Card> shuffledDeck(){
    List<Card> deck = fullDeck();
    Random rgen = new Random();
    for (int n = deck.size(); n > 1; n--){
      int random = (int) (n * rgen.nextFloat());
      // Switching deck[random] with deck[n-1]
      Card d = deck.get(random);
      deck.set(random, deck.get(n-1));
      deck.set(n-1,d);
    }
    return deck;
  }

  /* Needed to use cards as keys in HashMap */
  @Override
  public int hashCode(){
```

```
    return (suit.hashCode() ^ number);
  }

  @Override
  public boolean equals(Object o){
    if (o == null || !(o instanceof Card))
      return false;
    Card c = (Card) o;
    return suit == c.suit && number == c.number;
  }
}
```

```
package Solitaire.Model;

import java.util.HashSet;

/* 'Mixin' providing a set of event handlers and methods to add and remove
    from it */

public class EventBroadcaster<EventHandler> {

  /* DTI: listeners != null */

  protected HashSet<EventHandler> listeners;

  // Post: listeners = {}
  public EventBroadcaster(){
      listeners = new HashSet<EventHandler>();
  }

  // Post: listeners = listeners0 U {l}
  public void addListener(EventHandler l){
    if (l != null) listeners.add(l);
  }

  // Post: listener = listeners0 \ {l}
  public void removeListener(EventHandler l){
    if (l != null) listeners.remove(l);
  }
}
```

```java
package Solitaire.Model;

import java.util.List;

/* Models a Foundation stack, with the DTI enforcing the rules of the game */

public class Foundation extends Stack {

  /*
    Additions to Stack's specification:
    DTI: (1) 0 <= i < j < size => cards[i].suit = cards[j].suit
         (2) 0 <= i < size => cards[i].number = i + 1
  */

  @Override
  // Post: returns the number of visible cards, at most two (if top card is
      being dragged, two cards are visible)
  public int visible(){
    return Math.min(size(), 2);
  }

  @Override
  // Pre: n > 0
  // Post: returns whether removing n cards from this stack is a valid move
      according to the rules of the game
  public boolean validRemove(int n){
    assert n > 0;
    return n == 1 && size() > 0;
  }

  @Override
  // Pre: cs != null && cs != []
  // Post: returns whether adding cs to cards is a valid move according to the
      rules of the game
  public boolean validAdd(List<Card> cs){
    assert cs != null && cs.size() > 0;
    Card c = cs.get(0);

    // Checking DTI.1: if not empty, suit has to match and cs has to only be
        one card
    if (size() != 0 && (c.suit != cards.get(0).suit || cs.size() > 1))
      return false;

    // Checking DTI.2: has to be the right number
    return c.number == size() + 1;
  }
}
```

```
package Solitaire.Model;

import java.util.List;
import java.util.HashSet;

/* Models an entire game of Solitaire */

public class Game extends EventBroadcaster<GameListener> implements
    StackListener {

  /* Abs: trivial
     DTI: (1) 0 <= i < 4 => foundations[i] != null
          (2) 0 <= i < 7 => tableaus[i]
          (3) talon != null
          (4) 0 <= moves
          (5) 0 <= cardsMoved */

  private Foundation[] foundations;
  private Tableau[] tableaus;
  private Talon talon;
  private int moves,cardsMoved;

  public Game(GameListener l){

    // Shuffling the cards
    List<Card> deck = Card.shuffledDeck();

    // Creating the 4 empty foundations
    foundations = new Foundation[4];
    for (int i = 0; i < 4; i++){
      foundations[i] = new Foundation();
      foundations[i].addListener(this);
    }

    // Creating the talon with the first 24 cards (the cards are shuffled, so
        the dealing order does not matter)
    talon = new Talon(deck.subList(0,24));
    talon.addListener(this);

    // Creating the 7 tableaus with the rest of the cards
    tableaus = new Tableau[7];
    for (int i = 0, offset = 24; i < 7; offset += ++i){
      tableaus[i] = new Tableau(deck.subList(offset,offset+i+1));
      tableaus[i].addListener(this);
    }

    addListener(l);
  }

  // Pre: 0 <= i < 4
  // Post: returns the ith foundation
  public Foundation foundation(int i){
    assert i >= 0 && i < 4;
    return foundations[i];
  }

  // Post 0 <= i < 7
  // Post: returns the ith tableau
  public Tableau tableau(int i){
    assert i >= 0 && i < 7;
```

```
    return tableaus[i];
  }

  // Post: returns the talon/waste
  public Talon talon(){
    return talon;
  }

  @Override
  // Pre: n >= 0
  // Post: moves = moves0 + 1; cardsMoved = cardsMoved0 + n
  public void stackReceivedCards(Stack s, int n){
    assert n >= 0;
    moves++;
    cardsMoved += n;
    fireStateChanged();
    // If the event source is a foundation we might have won
    if (s instanceof Foundation && finished())
      fireGameOver();
  }

  // Post: returns whether the game is over
  private boolean finished(){
    for (int i = 0; i < 4; i++)
      if (foundations[i].size() < Card.suitSize)
        return false;
    return true;
  }

  protected void fireStateChanged(){
    for (GameListener l : listeners)
      l.stateChanged(moves,cardsMoved);
  }

  protected void fireGameOver(){
    for (GameListener l : listeners)
      l.gameOver(moves);
  }
}
```

```java
package Solitaire.Model;

/* Provides interface to listen to a State's events */

public interface GameListener {
    public void stateChanged(int moves, int cardsMoved);
    public void gameOver(int moves);
}
```

```
package Solitaire.Model;

import java.util.List;
import java.util.ArrayList;

/* Models a Tableau stack */

public class Tableau extends Stack {

  /*
    Additons to Stack's specification:
    Abs: cards[0..hidden) are upside down
    DTI: (1) size > 0 => hidden < size
         (2) size > 0 => cards[hidden..size) satisfy Tableau requirement
         (3) size > 0 && hidden = 0 => cards[0].number = 13
         (4) size = 0 => hidden = 0

    A sequence of cards xs satisfies the Tableau requirement iff:
      0 <= i < size -1 => xs[i].number = xs[i+1].number + 1 && xs[i].color !=
          xs[i+1].color
      (i.e alternating colors and ascending values)
  */

  private int hidden;

  // Pre: cs != null
  // Post: cards = cs && hidden = size() - 1;
  public Tableau(List<Card> cs){
    assert cs != null;
    cards.addAll(cs);
    // At beginning of game, only top card is visible
    hidden = size()-1;
  }

  @Override
  // Post: returns the number of visible cards
  public int visible(){
    return size()-hidden;
  }

  @Override
  // Post: returns whether removing n cards from this stack is a valid move
      according to the rules of the game
  public boolean validRemove(int n){
    // Checking DTI.1
    return hidden <= size()-n;
  }

  @Override
  // Pre: n > 0
  // Post: if validRemove(n): cards = cards[0..size-n)
  public void removeCards(int n){
    super.removeCards(n);
    // If top card is hidden, reveal it (maintaining DTI.1 and DTI.4)
    if (size() > 0 && hidden >= size())
      hidden = size() - 1;
  }

  @Override
  // Pre: cs != null && cs != []
```

```
  // Post: returns whether adding cs to cards is a valid move according to the
      rules of the game
  public boolean validAdd(List<Card> cs){
    assert cs != null && cs.size() > 0;
    Card c = cs.get(0);

    // Checking DTI.3: only a king may be added to an empty tableau
    if (size() == 0)
      return c.number == 13;

    // Checking DTI.2: new cards have to be a tableau and compatible
    return isTableau(cs) && compatible(cards.get(size()-1),c);
  }

  // Pre: bottom != null && top != null
  // Post: returns whether [bottom,top] satisfies Tableau requirement
  private boolean compatible(Card bottom, Card top){
    assert bottom != null && top != null;
    return (bottom.number == top.number + 1 && bottom.color ^ top.color);
  }

  // Pre: cs != null && cs != []
  // Post: returns whether cs satisifies Tableau requirement
  private boolean isTableau(List<Card> cs){
    assert cs != null && cs.size() > 0;
    // I: cs[0..i) satisfies Tableau requirement
    for (int i = 1; i < cs.size(); i++){
      if (!compatible(cs.get(i-1),cs.get(i)))
        return false;
    }
    return true;
  }
}
```

```java
package Solitaire.Model;

import java.util.List;
import java.util.ArrayList;

/* Models a Talon and the associated Waste */

public class Talon extends Stack {

  /*
    Additions to Stack's specification:
    Abs: (1) Talon = talon
         (2) Waste = Stack (= cards)
         (3) 0 <= i < j talon.size => talon[i] is conceptually ABOVE talon[j]
    DTI: (1) talon != null
  */

  private List<Card> talon;

  // Post: talon = cs && waste = []
  public Talon(List<Card> cs){
    talon = new ArrayList<Card>(cs);
  }

    // Post: returns the number of visible cards of the waste
  public int visible(){
    return Math.min(3,size());
  }

  // Post: returns talon == []
  public boolean talonEmpty(){
    return talon.size() == 0;
  }

  // Post: if talon != []: talon = drop 3 talon0 && waste = waste0 ++ take 3
      talon
  //       else: waste = talon0 && talon = waste0
  public void reveal(){
    // If talon is empty, switch talon and waste (this works because the lists
        grow in opposite directions)
    if (talonEmpty()){
      List<Card> d = talon;
      talon = cards;
      cards = d;
    }
    // Otherwise, reveal up to three cards
    else
      for (int i = 0; i < 3 && talon.size() > 0; i++)
        cards.add(talon.remove(0));
    // This should register as a move with 0 cards being moved
    fireStackReceivedCards(0);
  }

  @Override
  // Pre: n > 0
  // Post: returns whether removing n cards from this stack is a valid move
      according to the rules of the game
  public boolean validRemove(int n){
    assert n > 0;
    return n == 1 && size() > 0;
```

```
    }

    @Override
    // Pre: cs != null && cs != []
    // Post: returns whether adding cs to cards is a valid move according to the
        rules of the game
    public boolean validAdd(List<Card> cs){
      assert cs != null && cs.size() > 0;
      return false;
    }
  }
```

```
package Solitaire.Model;

import java.util.HashSet;
import java.util.List;
import java.util.ArrayList;

/* Models a Stack of cards */

public abstract class Stack extends EventBroadcaster<StackListener> {

  /*
    Abs: (1) Stack = cards
         (2) 0 <= i < j < cards.size => cards[i] is conceptually below
             cards[j]
    DTI: (1) cards != null
  */

  protected List<Card> cards;

  // Post: cards = []
  public Stack(){
    cards = new ArrayList<Card>();
  }

  // Post: returns (length cards)
  public int size(){
    return cards.size();
  }

  // Post: returns the number of visible cards
  public abstract int visible();

  // Pre: n <= visible()
  // Post: returns top n visible cards
  public List<Card> top(int n){
    assert n <= visible();
    return cards.subList(size()-n,size());
  }

  // Pre: n > 0
  // Post: returns whether removing n cards from this stack is a valid move
      according to the rules of the game
  public abstract boolean validRemove(int n);

  // Pre: n > 0 && validRemove(n)
  // Post: if validRemove(n): cards = cards[0..size-n)
  public void removeCards(int n) {
    assert n > 0 && validRemove(n);
    for (int i = 0; i < n; i++)
      cards.remove(size()-1);
  }

  // Pre: cs != null && cs != []
  // Post: returns whether adding cs to cards is a valid move according to the
      rules of the game
  public abstract boolean validAdd(List<Card> cs);

  // Pre: cs != null && cs != []
  // Post: returns validAdd(cs), if validAdd(cs): cards = cards0 ++ cs
  public boolean addCards(List<Card> cs) {
```

```
      if (!validAdd(cs))
        return false;
      cards.addAll(cs);
      fireStackReceivedCards(cs.size());
      return true;
    }

    protected void fireStackReceivedCards(int n){
      for (StackListener l : listeners)
        l.stackReceivedCards(this,n);
    }
  }
```

```java
package Solitaire.Model;

/* Provides interface to listen to a Stack's events */

public interface StackListener {
  public void stackReceivedCards(Stack s, int n);
}
```

```java
package Solitaire.View;
import Solitaire.Model.*;

import java.awt.Graphics;
import java.util.List;

/* Provides the View/Controller for a Foundation stack. */

public class FoundationView extends StackView {

  // The model
  private Foundation model;
  protected Stack model(){ return model; }

  public FoundationView(Foundation m, TableView t, int x, int y){
    super(t,x,y, TableView.CardWidth, TableView.CardHeight);
    model = m;
  }

  @Override
  // Post: g shows this stack at postion (x,y)
  public void draw(Graphics g){

    // If there is no card that's not being dragged, draw outline
    if (model.visible() - dragging < 1)
      Images.outline(x,y,g);
    // Otherwise, draw top card
    else
      Images.front(model.top(1+dragging).get(0),x,y,g);
  }

  @Override
  // Post: returns how many cards would physically be dragged at (px,py)
  public int startDrag(int px, int py){
    table.setOffset(px,py);
    return 1;
  }
}
```

```
package Solitaire.View;
import Solitaire.Model.*;

import java.awt.Image;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.RoundRectangle2D;
import java.util.HashMap;
import javax.imageio.ImageIO;
import java.io.File;

/* This class statically loads all the images and provides static methods to
    draw them. */

public final class Images {

  private static final String directory = "resources/", extension = ".gif",
      cardBackFilename = "back", backgroundName = "felt.jpg";
  private static HashMap<Card,Image> fronts = new HashMap<Card,Image>();
  private static Image back;
  private static Image background;

  static {
    try {
      for (Card c : Card.fullDeck())
        fronts.put(c,ImageIO.read(new File(directory + name(c) + extension)));
      // Random back image everytime the game is run!
      back = ImageIO.read(new File(directory + cardBackFilename + extension));
      background = ImageIO.read(new File(directory + backgroundName));
    }
    catch (Exception e){ e.printStackTrace(); }
  }

  private static String name(Card c){
    char suit = ' ';
    switch (c.suit){
      case Clubs: suit = 'c'; break;
      case Diamonds: suit = 'd'; break;
      case Hearts: suit = 'h'; break;
      case Spades: suit = 's'; break;
    }
    return ((c.number < 10) ? "0" : "") + c.number + suit;
  }

  // Draws the picture of c onto g at x,y
  public static void front(Card c, int x, int y, Graphics g){
    g.drawImage(fronts.get(c),x,y,null);
  }

  // Draws a card back onto g at x,y
  public static void back(int x, int y, Graphics g){
    g.drawImage(back,x,y,null);
  }

  // Draws a card outline onto g at x,y
  public static void outline(int x, int y, Graphics g){
    ((Graphics2D) g).draw(new RoundRectangle2D.Float(x, y,TableView.CardWidth,
        TableView.CardHeight,10,10));
  }
```

```
  // Draws the background image onto g at x,y
  public static void background(int x, int y, Graphics g){
    g.drawImage(background,x,y,TableView.TableWidth,TableView.TableHeight,null
        );
  }
}
```

```
package Solitaire.View;
import Solitaire.Model.*;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.JOptionPane;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/* This is the program's main Frame. It contains the TableView as well as
    labels with statistics (such as the number of moves) and a button to start
    a new game. */

public class Solitaire extends JFrame implements GameListener, ActionListener
    {

  private TableView table;
  private JLabel movesLabel, cardMovedLabel;

  public static void main(String[] args){
    new Solitaire();
  }

  public Solitaire(){
    super("Solitaire");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setResizable(false);
    setLayout(new BorderLayout());

    JPanel south = new JPanel();
    south.setLayout(new GridLayout(1,3));
    add(south, BorderLayout.SOUTH);

    movesLabel = new JLabel();
    movesLabel.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
    south.add(movesLabel);

    cardMovedLabel = new JLabel();
    cardMovedLabel.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
    south.add(cardMovedLabel);

    JButton reset = new JButton("New Game");
    reset.addActionListener(this);
    south.add(reset);

    table = new TableView(new Game(this));
    add(table, BorderLayout.CENTER);

    pack();
    setVisible(true);
  }

  @Override
  public void actionPerformed(ActionEvent e){
    newGame();
  }
```

```
  private void newGame(){
    table.setGame(new Game(this));
    stateChanged(0,0);
  }

  @Override
  public void stateChanged(int moves, int cardsMoved){
    movesLabel.setText("Moves: "+moves);
    cardMovedLabel.setText("Cards moved: "+cardsMoved);
    repaint();
  }

  @Override
  public void gameOver(int moves){
    Object[] options = {"Yes, it's so fun!","Quit"};
    if (0 == JOptionPane.showOptionDialog(this, "You won in "+moves+" moves!
        \nPlay another game?", "Congratulations", JOptionPane.
        YES_NO_CANCEL_OPTION, JOptionPane.QUESTION_MESSAGE, null, options,
        options[0]))
      newGame();
    else
      System.exit(0);
  }
}
```

```
package Solitaire.View;
import Solitaire.Model.*;

import java.awt.Graphics;

/* This provides the visual skeleton for a stack, including drawing and mouse
    events */

public abstract class StackView {

    protected int x,y,width,height, dragging;
    protected TableView table;
    protected abstract Stack model();

    public StackView(TableView t, int x, int y, int w, int h){
        table = t;
        this.x = x; this.y = y;
        width = w; height = h;
    }

    // Post: g shows this stack at postion (x,y)
    public abstract void draw(Graphics g);

    // Post: returns how many cards would physically be dragged at (px,py)
        (relative to the stack)
    public abstract int startDrag(int px, int py);

    // Finishes a move and conceptually removes cards from this stack if move
        successful
    public void endDrag(boolean success){
        if (success)
            model().removeCards(dragging);
        dragging = 0;
    }

    /* Responding to mouse events. The px and py coordinates are relative to
        this stack */

    // When this stack is pressed
    protected void handlePress(int px, int py){
        // How many cards are physically being dragged?
        int n = startDrag(px,py);
        // Are we dragging any?
        if (n == 0)
            return;
        // Are we allowed to drag this many?
        if (!model().validRemove(n))
            return;
        // Start the drag with top n cards
        table.startDrag(model().top(n), this);
        dragging = n;
    }

    // Pre: table.isDragging();
    protected void handleRelease(int px, int py){
        assert table.isDragging();
        // End the drag with result of adding cards to this stack as the
            success value
        table.endDrag(model().addCards(table.movingStack()));
    }
```

```
    protected void handleClick(int px, int py){}

    /* These calls from TableView correspond to the MouseListener events. If
       they are within this stack, the associated handle-method will be called
       and the method returns true */
    public boolean pressed(int px, int py){
        if (!inside(px,py))
            return false;
        handlePress(px-x,py-y);
        return true;
    }

    public boolean clicked(int px, int py){
        if (!inside(px,py))
            return false;
        handleClick(px-x,py-y);
        return true;
    }

    public boolean released(int px, int py){
        if (!inside(px,py))
            return false;
        handleRelease(px-x,py-y);
        return true;
    }

    private boolean inside(int px, int py){
        return (px >= x && px <= x+width && py >= y && py <= y + height);
    }
}
```

```
package Solitaire.View;
import Solitaire.Model.*;

import java.util.List;
import java.awt.Image;
import java.awt.Graphics;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import javax.swing.JPanel;

/* Provides the visual of a table, controls mouse events and controls moving
    cards */

public class TableView extends JPanel implements MouseListener,
    MouseMotionListener {

  static final int TableWidth = 640, TableHeight = 500,
                   TalonX = 20, TalonY = 20,
                   FoundationX = 269, FoundationY = 20,
                   TableauX = 20, TableauY = 160,
                   CardWidth = 73, CardHeight = 97,
                   Margin = 10, CardShift = 20;

  // Foundations, Tableaus and Talon
  private StackView[] stackViews;

  // What cards are currently being dragged ...
  private List<Card> movingStack;
  // ... where they come from ...
  private StackView source;
  // ... and where they are in relation to the table and the mouse.
  private int dragPointX, dragPointY, dragOffsetX, dragOffsetY;

  public TableView(Game g){
    setLayout(null);
    setPreferredSize(new java.awt.Dimension(TableWidth, TableHeight));

    addMouseListener(this);
    addMouseMotionListener(this);

    setGame(g);
  }

  // Initializes views for this game
  public void setGame(Game game){
    stackViews = new StackView[12];
    for (int i = 0; i < 7; i++)
      stackViews[i] = new TableauView(game.tableau(i),this, TableauX + i *
          (Margin+CardWidth), TableauY);
    for (int i = 0; i < 4; i++)
      stackViews[i+7] = new FoundationView(game.foundation(i),this,
          FoundationX + i * (Margin+CardWidth), FoundationY);
    stackViews[11] = new TalonView(game.talon(),this, TalonX, TalonY);
  }

  @Override
  public void paintComponent(Graphics g) {
    // Draw background
    Images.background(0,0,g);
```

```
    // Draw all stacks
    for (int i = 0; i < stackViews.length; i++)
      stackViews[i].draw(g);

    // Draw the dragged stack
    if (isDragging())
      for (int i = 0; i < movingStack.size(); i++)
        Images.front(movingStack.get(i),dragPointX, dragPointY + i*CardShift,g
            );
  }

  /* Dragging logic */
  public void startDrag(List<Card> cs, StackView src){
    movingStack = cs;
    source = src;
  }

  public void setOffset(int px, int py){
    dragOffsetX = px; dragOffsetY = py;
  }

  public boolean isDragging(){
    return movingStack != null;
  }

  public List<Card> movingStack(){
    return movingStack;
  }

  public void endDrag(boolean success){
    assert isDragging();
    source.endDrag(success);
    movingStack = null; source = null;
    repaint();
  }

  /* Mouse Events */
  // Every StackView will be notified, and because they are disjoint, at most
      one will execute something.
  @Override
  public void mousePressed(MouseEvent e){
    // Try all StackViews and stop once one responded
    for (int i = 0; i < 12 && !stackViews[i].pressed(e.getX(),e.getY()); i++);
    // mouseDragged is not fired before moving the cursor, so we have to do so
        manually
    mouseDragged(e);
  }

  @Override
  public void mouseDragged(MouseEvent e) {
    // No need to do this if we're not actually moving anything
    if (isDragging()){
      dragPointX = e.getX() - dragOffsetX;
      dragPointY = e.getY() - dragOffsetY;
      repaint();
    }
  }

  @Override
```

```
    public void mouseClicked(MouseEvent e){
      for (int i = 0; i < 12 && !stackViews[i].clicked(e.getX(),e.getY()); i++);
    }

    @Override
    public void mouseReleased(MouseEvent e){
      if (isDragging()){
        for (int i = 0; i < 12 && !stackViews[i].released(e.getX(),e.getY()); i+
            +);
        // If user didn't let go above a stack, end drag manually
        if (isDragging())
          endDrag(false);
      }
    }

    // If the mouse exits the window and we are dragging, abort
    @Override
    public void mouseExited(MouseEvent e){
      if (isDragging())
        endDrag(false);
    }

    @Override
    public void mouseMoved(MouseEvent e){}
    @Override
    public void mouseEntered(MouseEvent e) {}
  }
```

```
package Solitaire.View;
import Solitaire.Model.*;

import java.awt.Graphics;

/* This provides the View/Controller for a Tableau stack. Since cards can be
    both moved from and onto a Foundation, it is a TargetView (and by extension
    a SourceView). */

public class TableauView extends StackView {

  // The model
  private Tableau model;
  protected Stack model(){ return model; }

  public TableauView(Tableau m, TableView t, int x, int y){
    super(t,x,y,TableView.CardWidth,TableView.TableHeight - y);
    model = m;
  }

  @Override
  // Post: g shows this stack at postion (x,y)
  public void draw(Graphics g){

    // Outline
    if (model.size() - dragging == 0)
      Images.outline(x,y,g);

    // Hidden cards
    int hidden = model.size() - model.visible();
    for (int i = 0; i < hidden; i++)
      Images.back(x, y + i * TableView.CardShift,g);

    // Open cards (except the top cards if they are being dragged)
    for (int i = 0; i < model.visible() - dragging; i++)
      Images.front(model.top(model.visible()).get(i),x,y + (i+hidden) *
          TableView.CardShift,g);
  }

  @Override
  // Post: returns how many cards would physically be dragged at (px,py)
  public int startDrag(int px, int py){

    // Did we miss the stack?
    if (py > TableView.CardHeight + (model.size()-1) * TableView.CardShift)
      return 0;

    // Did we hit the top card?
    if (py > (model.size()-1) * TableView.CardShift){
      table.setOffset(px,py - (model.size()-1) * TableView.CardShift);
      return 1;
    }

    // How many cards aren't being dragged?
    int n = py / TableView.CardShift;

    table.setOffset(px,py - n * TableView.CardShift);
    return model.size() - n;
  }
}
```

```
package Solitaire.View;
import Solitaire.Model.*;

import java.awt.Graphics;

/* This provides the View/Controller for a Talon stack. Since cards can't be
    moved onto the Talon, it is only a SourceView */

public class TalonView extends StackView {

  // The model
  private Talon model;
  protected Stack model(){ return model; }

  public TalonView(Talon m, TableView t, int x, int y){
    super(t,x,y, 2*(TableView.CardWidth+TableView.CardShift)+TableView.Margin,
        TableView.CardHeight);
    model = m;
  }

  @Override
  // Post: g shows this stack at postion (x,y)
  public void draw(Graphics g){

    // Talon, either the outline or just a card back
    if (model.talonEmpty())
      Images.outline(x,y,g);
    else
      Images.back(x,y,g);

    // Waste, top cards (3 if available), one less if top card is being
        dragged
    for (int i = 0; i < model.visible() - dragging; i++)
      Images.front(model.top(model.visible()).get(i), x + TableView.Margin +
          TableView.CardWidth + i * TableView.CardShift, y,g);
  }

  @Override
  // Post: returns how many cards would physically be dragged at (px,py)
  public int startDrag(int px, int py){
    // Did we miss the waste?
    if (px < TableView.CardWidth+TableView.Margin)
      return 0;

    // Dragging anywhere on the waste will just result in the top card being
        dragged
    table.setOffset(px - TableView.CardWidth - TableView.Margin - TableView.
        CardShift * Math.min(2,model.size()-1), py);
    return 1;
  }

  @Override
  public void handleClick(int px, int py){
    // If the talon is clicked, the reveal() method is called
    if (px < TableView.CardWidth)
      model.reveal();
  }
}
```