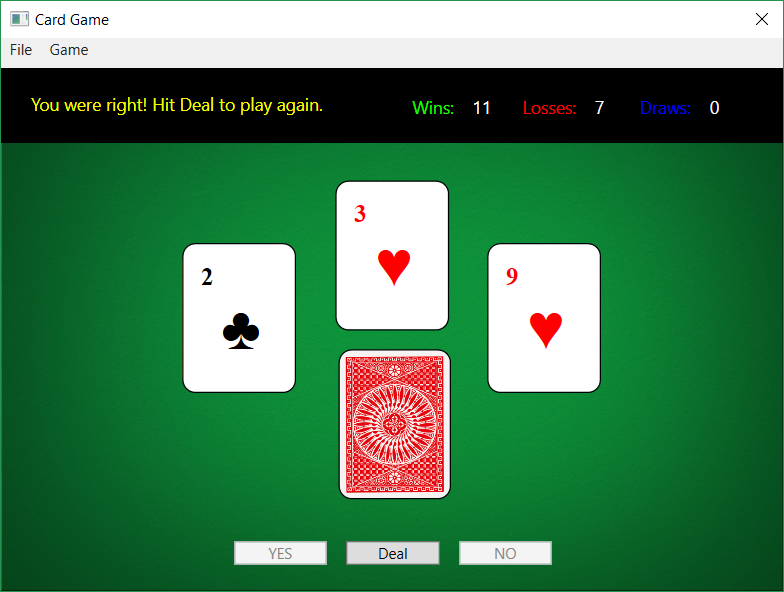
GUI Lab – Windows Presentation Foundation (WPF) and Windows Designer (Part 2)

In Part 1 of the lab, we set up all the GUI components of the Acey Deucey card game. In Part 2, we will set up the data and game logic.



# 1. Adding the Game’s Data Model

A well-designed application separates its data from the presentation (GUI) and logic as much as possible. Here, our data are the scores – wins, losses, and draws. We’ll create a fairly simple ScoresModel class to manage the scores, but with one important additional feature: it will implement the INotifyPropertyChanged interface. This will allow the class to notify subscribed GUI components when the number of wins, losses, or draws changes.

We’ll also set up a DataContext element in our XAML to help bind the score labels to our ScoresModel. Anytime our code updates the ScoresModel, the score labels in the GUI will automatically reflect those changes.

## Adding the ScoresModel Class

1. Right-click the “Card Game” project icon and choose **Add > Class**.
2. Name the class ScoresModel, then hit **Add**.
3. Add the following using statement to the list of imports at the top of the new source code file:

using System.ComponentModel;

1. Update the first line of the class so that ScoresModel implements the INotifyPropertyChanged interface.
2. If necessary, delete the empty constructor inside the ScoresModel class.
3. Add all of the following code to the *body* of the ScoresModel class:

private int wins, losses, draws = 0;

// Get/sets the number of draws

public int Draws

{

get

{

return this.draws;

}

set

{

this.draws = value;

OnPropertyChanged("Draws");

}

}

// Get/sets the number of losses

public int Losses

{

get { return losses; }

set {

losses = value;

OnPropertyChanged("Losses");

}

}

// Get/sets the number of wins

public int Wins

{

get { return wins; }

set {

wins = value;

OnPropertyChanged("Wins");

}

}

// Resets the wins, losses, and draws to 0

public void Reset()

{

Wins = 0;

Losses = 0;

Draws = 0;

}

// propertyChanged property of INotifyPropertyChanged

// interface MUST be implemented

public event PropertyChangedEventHandler PropertyChanged;

// OnPropertyChanged property of INotifyPropertyChanged

// interface MUST be implemented

private void OnPropertyChanged(string propertyName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(propertyName));

}

}

1. **IMPORTANT**: Build the solution (**Build** menu > **Build Solution**, or hit the **F7** key). We need Visual Studio to compile the ScoresModel class for the next section of this lab.

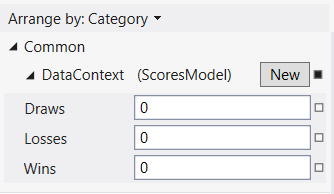
The ScoresModel class has three fields, four properties, and two methods. Note that the Wins, Losses, and Draws properties call the private OnPropertyChanged() method implemented from the INotifyPropertyChanged interface. This is how the ScoresModel notifies event handlers of changes to its data.

## Creating a Data Context

Let’s return to our XAML code to set up a <Window.DataContext> element. When you bind data to a particular element in your GUI, you need to define the *source* of that data – that’s what a **data context** is.

Setting up a simple Data Context is relatively easy:

1. Go to your *MainWindow.xaml* file and select your GUI’s Window component.
2. In the **Properties** panel, search for DataContext.
3. Hit the **New** button next to the DataContext property.
4. Select ScoresModel in the list of classes.
5. Hit **OK**. When you’re finished, you should see “(ScoresModel)” added to the **Properties** panel, as well as the three properties of the ScoresModel class itself.



# 2. Code: Adding the fields and constructor to the code-behind file

Next on the agenda is to set up the fields and constructor code in our *MainWindow.xaml.cs* code-behind file. Let’s start with the fields.

## Adding the fields

* In the **Solution Explorer**, expand the *MainWindow.xaml* file so that the *MainWindow.xaml.cs* code-behind file appears.
* Open the *MainWindow.xaml.cs* code-behind file.
* Add the following code between the very first line of the MainWindow class and its constructor:

// Data model

private ScoresModel scores = new ScoresModel();

private Deck myDeck;

private Card card1, card2, midCard; // vars to store dealt cards

private int cardNumber, cardsDealt; // card counters

// Constants for the display text

const string WIN\_TEXT = "You were right! ";

const string LOSS\_TEXT = "WRONG. You lose. ";

const string DRAW\_TEXT = "It's a draw. ";

const string DEAL\_AGAIN\_TEXT = "Hit Deal to play again.";

const string GUESS\_TEXT = "Will the next card be between these two?";

Visual Studio generates errors, telling you it can’t find the Deck and Card classes. Add the following to the list of using statements at the top of the file:

using CS225CardLibrary;

So, what do we have here? We declared and initialized our scores model, which serves as the MainWindow’s data context. We also set up fields for a Deck of cards, as well as for the three cards dealt each hand. There are card counters for the current hand (cardNumber) and the total number of cards dealt (cardsDealt.)

Below the card-related fields are constant strings for the message displayed whenever the user takes an action. *Get in the habit of using constant strings for display messages*. Not only does it make your code more modular, but it saves you from having to scroll through hundreds of lines of code to update a particular string.

## Updating the Constructor

After initializing all GUI components, the MainWindow constructor should initialize and shuffle the Deck of cards, as well as set up the GUI’s Data Context.

Add the following lines of code *after* the InitializeComponents() statements in the body of the constructor:

// Initialize and shuffle the deck

myDeck = new Deck();

myDeck.Shuffle();

// Set the data context to our ScoresModel

base.DataContext = scores;

# 3. Code: Painting a Card

Now we’re starting to get into the meat of the code.

Our GUI has a generic ContentControl object that uses a template to create a graphic of a card. The following private method takes a ContentControl and a Card dealt from the deck, then sets the Card’s rank and suit as the text of the template’s labels. Add it to your MainWindow class:

/// <summary>

/// Paints a card on the GUI

/// </summary>

private void PaintCard(ContentControl cc, Card card)

{

// Find the rank and suit Labels in cardTemplate

Label rank = cc.Template.FindName("rankLabel", cc) as Label;

Label suit = cc.Template.FindName("suitLabel", cc) as Label;

// Set the labels with the card's rank and suit

rank.Content = card.GetRankAsString();

suit.Content = card.GetSuitAsString();

// If the card is a diamond or heart, paint the rank and suit red

if (card.suit == Suits.Diamonds || card.suit == Suits.Hearts)

{

rank.Foreground = new SolidColorBrush(Colors.Red);

suit.Foreground = new SolidColorBrush(Colors.Red);

}

}

Some points worth mentioning about the method:

* The first two lines have a lot going on. Each line asks the ContentControl’s Template to find a child element whose name matches the given string, using the ContentControl (cc) itself as the starting “root” point of the search. Because FindName() returns a generic object, we have to use downcasting (as Label) to force C# to treat the object as a Label.
* The next two lines return the Card’s rank and suit as strings using the GetRankAsString() and GetSuitAsString() methods of the Card class. We assign each to the Content property of the appropriate Labels.
* Finally, if the Card’s suit is a diamond or heart, we repaint the rank and suit Label’s Foreground property to red using a SolidColorBrush.

**YOUR TURN**: This is a good time to Save your work. Hit **Ctrl+Shift+S**, do **File** menu > **Save All**, or hit the  Save All button in the main toolbar.

# 4. Code: Dealing a Card

The next method, DealCard(), is responsible for dealing a Card.

The method takes one parameter (@seq) that represents the card’s sequence in the hand – i.e., is it the 1st (left) card dealt, 2nd (right) card, or 3rd (middle) card? Note that this is NOT the event handler that the “Deal” button calls when a user hits it…we’ll add that later in the lab.

For now, add the following private method to your MainWindow class. **Don’t worry about compiler errors. We’ll fix those in the next section.**

// Deals the @seq (1, 2, 3) Card in the hand

private Card DealCard(int seq)

{

// Create a new ContentControl; apply the cardTemplate to make it look like a playing card

ContentControl card = new ContentControl();

card.Template = Resources["cardTemplate"] as ControlTemplate;

card.ApplyTemplate();

// Draw a new card from the deck

Card myCard = myDeck.GetCard(cardNumber++);

// Paint the rank & suit on the card

this.PaintCard(card, myCard);

// Set the starting region of the card animation

// - The first card moves to the left

// - The second card moves to the right

string canvasAnchor = "";

switch (seq)

{

case 1: canvasAnchor = "(Canvas.Left)"; break;

case 2: canvasAnchor = "(Canvas.Right)"; break;

default: canvasAnchor = "(Canvas.Left)"; break;

}

// Add the card to the GUI

playArea.Children.Add(card);

// Set the animation starting and ending points, depending on which card we're dealing

if (seq == 3)

{

AnimateCard(card, playArea.ActualWidth / 2 - 45, playArea.ActualWidth / 2 - 45, canvasAnchor);

AnimateCard(card, card.ActualHeight + 20, card.ActualHeight + 150, "(Canvas.Bottom)");

}

else

{

AnimateCard(card, playArea.ActualWidth / 2 - 45, 20, canvasAnchor);

AnimateCard(card, card.ActualHeight + 20, card.ActualHeight + 100, "(Canvas.Bottom)");

}

return myCard;

}

Some points worth mentioning about this method:

* The first three lines essentially create the “shell” of a new card.
  + The 1st line creates a new, empty ContentControl named card.
  + The 2nd line sets the ContentControl’s Template property to cardTemplate
  + The 3rd line is critical. You would think setting the card’s Template property would be enough to apply the cardTemplate to it…but it’s not. You have to call the ApplyTemplate() method separately to actually construct all of the template’s elements: i.e., the Rectangle and the two Labels
* The switch-case block sets up the card animations.
  + The first card in the hand sequence starts on top of the “deck” and moves to the left, hence the use of “Canvas.Left” as the attached property for the ContentControl
  + Same for the second card in the hand sequence, but it moves to the right.
  + By default, any remaining cards will be anchored to Canvas.Left
* playArea.Children.Add(card) adds the card graphic to the list of child elements of the playArea Canvas.
* The if-else() block is responsible for managing the Card animations correctly.
  + If it’s the third card in the sequence, the card moves from the deck straight up between the other two cards. The first call to AnimateCard() controls the starting and ending x-coordinates of the card. The second call to AnimateCard() controls the y-coordinates.
  + Otherwise, move the card left or right depending on the value of the canvas Anchor
* Finally return the dealt Card object

# 5. Code: Animating a Card

The DealCard() method makes calls to an AnimateCard() method, which is the method responsible for moving the cards on the screen. Before we add the method to the MainWindow class, we need to add a pair of namespaces to our list of using statements at the top of our source code file:

using System.Windows.Media.Animation;

using System.Windows.Threading;

After adding those two statements, scroll back down to the bottom of the file and add the AnimateCard() method inside the body of the class:

/// <summary>

/// Animates the given @card ContentControl's @propertyToAnimate, starting

/// at the @from value and ending at the @to value.

/// </summary>

private void AnimateCard(ContentControl card, double from, double to, string propertyToAnimate)

{

// Create a new Storyboard with a one-way animation

Storyboard storyboard = new Storyboard() { AutoReverse = false };

// Create a new Animation object, setting its from, to, and duration properties.

DoubleAnimation animation = new DoubleAnimation()

{

From = from,

To = to,

Duration = new Duration(TimeSpan.FromSeconds(0.4)),

};

// Use the card and animation together on the Storyboard

Storyboard.SetTarget(animation, card);

// Set up the animation path using @propertyToAnimate as the guide

Storyboard.SetTargetProperty(animation, new PropertyPath(propertyToAnimate));

// Add the animation to the Storyboard and start it

storyboard.Children.Add(animation);

storyboard.Begin();

}

Some points worth mentioning about this method:

* Storyboard is the built-in C# class for defining an animation sequence.
* DoubleAnimation creates a container for the animation’s properties. An animation has two states: a starting point and an ending point. The “Double” means we’ll store all numerics as doubles.
  + From property sets the starting value
  + To property sets the ending value
  + Duration property takes a TimeSpan object. Here we use the TimeSpan object to set the duration to 0.4 seconds
* SetTarget applies the animation to the card object
* SetTargetProperty applies the specific propertyToAnimate to the animation. For us, the property is one of the attached Canvas properties – either Canvas.Left or Canvas.Right.
* storyboard.Children.Add(animation) adds the animation to our storyboard, and the call to Begin() sets it in motion.

# 6. Code: Comparing Cards

A key part of the game logic is to compare the first two cards that are dealt in each hand. If the difference between the ranks is at least 2, we’ll ask the user to guess what the third card will be, then deal the third card. Otherwise, we declare a draw and set up the GUI so the user can deal two cards again.

Another key part of the game is comparing the third dealt card with the other two. We also use the player’s guess to determine if they won the hand or not.

We’ll implement this logic as separate overloaded CompareCards() methods. Here’s the first of the two to add to the bottom of the MainWindow class:

/// <summary>

/// Compares two cards, determining the difference between their ranks;

/// updates the GUI accordingly

/// </summary>

/// <param name="card1">The first Card to compare</param>

/// <param name="card2">The second Card to compare</param>

private void CompareCards(Card card1, Card card2)

{

// If the difference between the 2 ranks is 0 or 1, we have a draw.

if (card1.rank == card2.rank || Math.Abs(card1.rank - card2.rank) == 1)

{

// Update the status label and scores model

statusLabel.Content = DRAW\_TEXT + DEAL\_AGAIN\_TEXT;

scores.Draws++;

}

else

{

// Otherwise, update button states

dealButton.IsEnabled = false;

yesButton.IsEnabled = true;

noButton.IsEnabled = true;

// Ask for a guess

statusLabel.Content = GUESS\_TEXT;

}

}

And this is the second of the two, handling all three cards:

/// <summary>

/// Compare the 3rd card with the other two dealt cards; take into account

/// the user's choice when scoring.

/// </summary>

/// <param name="card1">The first card to compare</param>

/// <param name="card2">The second card to compare</param>

/// <param name="midCard">The third card dealt; compare with @card1 & @card2</param>

/// <param name="choice">The user's choice, YES or NO, as to whether the rank of

/// @midCard falls between card1's and card2's rank.

/// </param>

private void compareCards(Card card1, Card card2, Card midCard, string choice)

{

// Create a temp list of all 3 cards & sort them

List<Card> cards = new List<Card> {

card1, card2, midCard

};

cards.Sort();

// Was the 3rd card between the other two, and did the user guess correctly?

if (cards[0].rank < midCard.rank && midCard.rank < cards[2].rank)

{

if (choice == "yes")

{

statusLabel.Content = WIN\_TEXT + DEAL\_AGAIN\_TEXT;

scores.Wins++;

}

else

{

statusLabel.Content = LOSS\_TEXT + DEAL\_AGAIN\_TEXT;

scores.Losses++;

}

}

else

{

if (choice == "no")

{

statusLabel.Content = WIN\_TEXT + DEAL\_AGAIN\_TEXT;

scores.Wins++;

}

else

{

statusLabel.Content = LOSS\_TEXT + DEAL\_AGAIN\_TEXT;

scores.Losses++;

}

}

}

The second of the two methods determines whether or not the person won or lost the hand.

* We create a temporary sorted List of the three Card objects, helping us to determine scoring.
* The if-else logic seems a bit verbose, but it allows to do just one complex, compound test in the top-level if() block, sparing us error-prone else-ifs later.
  + If the third “midCard” really is the middle card AND the user said “yes”, they win; otherwise, they lose.
  + If the third “midCard” isn’t really the middle card AND the user said “no”, they win; otherwise, they lose.
* In all cases, we display the appropriate text in statusLabel and update the Wins or Losses count.
* **Thanks to data binding, the Wins and Losses Labels on the GUI automatically update in sync with the data.**

**YOUR TURN**: This is a good time to Save your work. Hit **Ctrl+Shift+S**, do **File** menu > **Save All**, or hit the  Save All button in the main toolbar.

# 7. Code: Dealing the 3rd card in a hand

The next method handles the dealing of the third card in a hand. Add the following method to the body of your MainWindow class:

private void DealThirdCard(String choice)

{

// Timer delay: wait 1.2 seconds before comparing the two dealt cards

DispatcherTimer compTimer = new DispatcherTimer { Interval = TimeSpan.FromSeconds(0.6) };

compTimer.Start();

compTimer.Tick += (send, args) =>

{

compTimer.Stop();

compareCards(card1, card2, midCard, choice);

// Update button states

dealButton.IsEnabled = true;

yesButton.IsEnabled = false;

noButton.IsEnabled = false;

};

// Deal the middle card; increment cards dealt

midCard = DealCard(3);

this.cardsDealt += 1;

}

* Starting from the bottom of the method, we call DealCard(), passing an argument of “3” to indicate that this is the 3rd card of the hand. We assign the returned Card to the midCard field of this class.
* The last line increments the cards dealt count by 1
* Returning to the top of the method, we see for the first time an important class for setting the timing of events in the game: DispatcherTimer.

After dealing the third card, we’d like to compare the third card to the other two. The problem is that the animations we use in the game run in separate asynchronous threads. This means that if you set your code up in a normal sequence, the call to CompareCards may occur before the third card is dealt! To prevent this from happening, we use a DispatcherTimer to delay the call to CompareCards() until the animation is finished.

* The compTimer.Tick statement uses a lambda expression to subscribe to a custom event. That custom event stops the timer, compares the two cards, and updates the state of the buttons.

# 8. Event Handler: “Deal” Button

Now we’re ready to attach event handlers to our buttons. The first one is the big one: the “Deal” button

* Go to the **Design** view of your *MainWindow.xaml* file
* Click the “Deal” button to select it
* In the **Properties** panel, click the **Event Handlers** (lightning bolt) button to bring up the events the button can subscribe to.
* Double-click inside the Click event. Visual Studio creates an empty dealButton\_Click event handler for you
* In the body of the dealButton\_Click method, add the following code:

// If there are fewer than 3 cards left in the deck, shuffle it

// and reset the cards in the GUI

if (cardsDealt >= 50)

{

// Remove all dealt cards

playArea.Children.RemoveRange(0, playArea.Children.Count);

// Use a new shuffled deck

myDeck = new Deck();

myDeck.Shuffle();

// Reset the card counters

cardsDealt = 0;

cardNumber = 0;

}

else if (cardsDealt > 0 && midCard != null)

{

// Remove the middle card dealt from the previous hand

playArea.Children.RemoveAt(playArea.Children.Count - 1);

}

// Reset Card fields

card1 = null;

card2 = null;

midCard = null;

// Timer delay: wait 1.2 seconds before comparing the two dealt cards

DispatcherTimer compTimer = new DispatcherTimer { Interval = TimeSpan.FromSeconds(1.2) };

compTimer.Start();

compTimer.Tick += (send, args) =>

{

compTimer.Stop();

CompareCards(card1, card2);

};

// Timer delay: wait 0.6 seconds before dealing the second card

DispatcherTimer timer = new DispatcherTimer { Interval = TimeSpan.FromSeconds(0.5) };

timer.Start();

timer.Tick += (send, args) =>

{

timer.Stop();

card2 = DealCard(2);

};

// Deal the first card

card1 = DealCard(1);

// Update the number of cards dealt by 2

this.cardsDealt += 2;

Some discussion about all the code in this one:

* We start off by checking the status of the Deck.
  + If we’ve dealt 50 or more cards, it’s time to shuffle the Deck. We remove all of the child elements (cards) from the playArea, and reset the card counts to 0.
  + Otherwise, if we’ve just completed a hand (cardsDealt > 0 AND the third “midCard” is occupied), we remove the middle card from the GUI
* After resetting the Card fields, we set up two DispatcherTimer objects to delay the triggering of the second (right) card animation and the comparison of the two cards
* We deal the first Card, and increment the number of cards dealt by 2. We’re always guaranteed to deal at least 2 cards in a hand.

# 9. Implementing the other event handlers

There’s very little left to do. This last section has you set up the remaining event handlers for the YES and NO buttons, as well as the Reset Scores menu item in the Games menu:

* Go back to *MainWindow.xaml.* In **Design** view, select the YES button.
* In the **Properties** panel, go to the **Event Handlers** (lightning bolt button) and double-click inside the Click event handler box. Visual Studio creates a yesButton\_Click event handler for you.
* **YOUR TURN**: Inside the body of the yesButton\_Click event handler, add a statement that calls the DealThirdCard() method with an argument of “yes”.
* Repeat these steps for the NO button. The call to the DealThirdCard() method should use an argument of “no”.
* To set up the Reset Scores menu item’s event handler, go back to *MainWindow.xaml* and its **Design** view. Go to the **Document Outline** and locate/select the resetMenuItem menu item.
* In the **Properties** panel, go to the **Event Handlers** (lightning bolt button) and double-click inside the Click event handler box. Visual Studio take you to the resetMenutItem\_Click event handler in the code-behind file.
* **YOUR TURN**: Inside the body of that handler, add a statement that calls the Reset() method of the scores field.

# 10. Binding elements to the data model

You’ve set up all of the major pieces and event handlers; now it’s time to set up the data bindings. The three GUI elements that we’ll bind to the ScoresModel are the winsCountLabel, lossesCountLabel, and drawsCountLabel.

* Go back to *MainWindow.xaml* and its **Design** view, then right-click on the winsCountLabel (the “0” next to “Wins”)
* Choose **Create Data Binding for Content…**
* Make sure the selected **Binding Type** is “Data Context”
* In the **Path** box, select **Wins**
* Hit **OK**

**YOUR TURN**: Follow the same steps to bind the lossesCountLabel and drawsCountLabel to the Losses and Draws properties of ScoresModel, respectively.

FINALLY, one last cosmetic issue we need to take care of:

* In the **Document Outline**, select the cardContentControl item (i.e., the generic card sitting in the middle of the GUI)
* In the **Properties** panel, change its Visibility property to “Hidden”.

**Congratulations!** You’re finished with the game. **Build** the Solution, then run it.