**Peer Evaluation for Blackjack Application**

|  |  |
| --- | --- |
| Your name: (Your lab is the one being evaluated) | **Steven Wilson** |
| Name(s) of peer evaluator(s) | **Patrick Behrens-Jungfer** |
| Date: | **04/28/2019** |

Instructions  
You should have already completed as much of BlackJack as you can given your time constraints. After you and a peer have evaluated your work, you will submit this evaluation along with screen shots and source code indicated in moodle. You may make corrections to your work as a result of the evaluation.

|  |  |
| --- | --- |
| ***Blackjack GUI Problem*** | |
| Create a solution that included 3 or 4 projects? (Windows form (blackjack and concentration), class library, console app for testing?) Blackjack GUI is the startup project? Added a using using statement in the blackjack form to access the class library? Added a reference in the gui project to access the class library? | **Yes.**  **Yes.**  **Yes.**  **Yes.**  **Yes.** |
| Instance variables on the form are objects rather than simple variables (ints, strings)? What are those? | **Yes.**  **The deck, player hand, and dealer hand objects.** |
| Methods are added to the form to do work that is logically complicated that is not handled by a methods in the classes? Those methods pass parameters and return values appropriately? What are those methods? | **Yes, but I didn’t have enough time to complete the project, so I’m uploading what I have.** |
| Additional classes are created to deal with the interaction between classes like the BJHand or Player or Dealer and the UI? What are those? Those classes are added to the class diagram? Screen shot is included? | **No, but methods are.** |
| Does the application function appropriately? Can you interact with the application in a way that causes it to “degrade” and not gracefully? Screen shots are included? | **Not really.**  **Yes.**  **Yes.** |
| Source code for the UI is included? Instance variables, methods, classes and event handlers are illustrated? | **Yes. Not sure what you mean by the last part though.** |

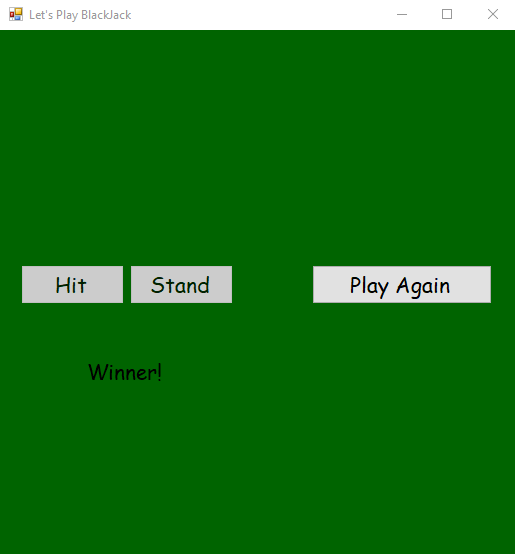
|  |  |
| --- | --- |
| ***Mexican Train Domino GUI Problem*** | |
| Added the UI project provided to the multi project solution? GUI is the startup project? Added a using using statement in the form to access the class library? Added a reference in the gui project to access the class library? | **Yes.**  **Yes.**  **Yes.**  **Yes.** |
| Comments have been added to the instance variables for the form? | **I didn’t change the form code.** |
| Comments have been added to methods are added to the form to do work that is logically complicated that is not handled by a methods in the classes? | **I didn’t change the code for the form.** |
| Comments have been added to the event handlers? | **See above.** |
| Does the application function appropriately? Screen shots are included? | **Almost. There’s a few problems I don’t understand but I don’t have enough time right now to fix them.** |

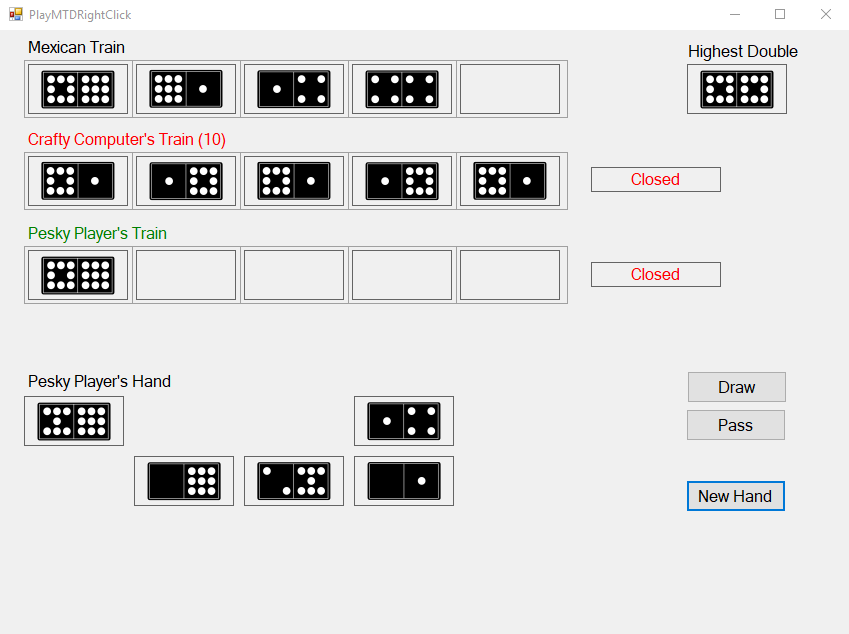
|  |  |
| --- | --- |
| ***Programming style for all programs*** | |
| Is proper indentation used? Is each event handler indented properly? Is each method indented properly? |  |
| Are comments used appropriately? |  |
| Do variable names use camel case? (camelCase for example) |  |
| Do property/method names use Title Case (or Pascal Case?) |  |

**GitHub repo:** https://github.com/drazhok/CS234N

General comments and notes:

Screen Shots and Source Code





**BlackJack**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

using CardClasses;

namespace BlackJack

{

public partial class boardFormSimple : Form

{

#region Instance Variables

Deck cards;

BJHand dealerHand;

BJHand playerHand;

#endregion

public boardFormSimple()

{

InitializeComponent();

// Initializes the instance variables

cards = new Deck();

dealerHand = new BJHand();

playerHand = new BJHand();

// Shuffles the new deck 10 times

for(int i = 0; i < 10; i++)

cards.Shuffle();

}

#region Methods

private void Reset()

{

hitButton.Enabled = true;

standButton.Enabled = true;

playerWinLabel.Visible = false;

dealerWinLabel.Visible = false;

playAgainButton.Enabled = false;

HideAllCards();

cards.Shuffle();

Card c1 = cards.Deal();

Card c2 = cards.Deal();

Card c3 = cards.Deal();

Card c4 = cards.Deal();

playerHand.AddCard(c1);

Show(card6, c1);

dealerHand.AddCard(c2);

Show(card1, c2);

playerHand.AddCard(c3);

Show(card7, c3);

dealerHand.AddCard(c4);

Show(card2, c4);

}

private bool CheckWinner()

{

if (playerHand.IsBusted)

return false;

else

{

if (!dealerHand.IsBusted)

{

if (playerHand.Score > dealerHand.Score)

return true;

else

return false;

}

else

return true;

}

}

private void HideAllCards()

{

for (int i = 1; i <= 10; i++)

this.Controls["card" + i].Visible = false;

}

private void Show(PictureBox p, Card c)

{

p.Image = Image.FromFile(System.Environment.CurrentDirectory + "\\Cards\\" + c.FileName);

}

private void ShowBack(PictureBox p, Card c)

{

p.Image = Image.FromFile(System.Environment.CurrentDirectory + "\\Cards\\black\_back.jpg");

}

#endregion

private void frmBoard\_Load(object sender, EventArgs e)

{

Reset();

hitButton.Enabled = true;

standButton.Enabled = true;

playerWinLabel.Visible = false;

dealerWinLabel.Visible = false;

playAgainButton.Enabled = false;

}

private void hitButton\_Click(object sender, EventArgs e)

{

playerHand.AddCard(cards.Deal());

if (CheckWinner())

{

playerWinLabel.Visible = true;

standButton.Enabled = false;

hitButton.Enabled = false;

playAgainButton.Enabled = true;

}

else

{

dealerWinLabel.Visible = true;

standButton.Enabled = false;

hitButton.Enabled = false;

playAgainButton.Enabled = true;

}

}

private void standButton\_Click(object sender, EventArgs e)

{

}

private void playAgainButton\_Click(object sender, EventArgs e)

{

Reset();

}

}

}

**MTD**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace MTDClasses

{

[Serializable()]

public class Domino

{

private int side1;

private int side2;

public Domino()

{

Side1 = 0;

Side2 = 0;

}

public Domino(int p1, int p2)

{

side1 = p1;

side2 = p2;

}

public int Side1

{

get => side1;

set

{

if (value < 0 || value > 12)

throw new ArgumentOutOfRangeException("The domino side value is invalid.");

side1 = value;

}

}

public int Side2

{

get => side2;

set

{

if (value < 0 || value > 12)

throw new ArgumentOutOfRangeException("The domino side value is invalid.");

side2 = value;

}

}

public void Flip()

{

int tempSide = side1;

side1 = side2;

side2 = tempSide;

}

public int Score => side1 + side2;

public bool IsDouble() => (side1 == side2);

public string Filename => String.Format("d{0}{1}.png", side1, side2);

//public override string ToString() => String.Format("Side 1: {0} Side 2: {1}", Side1, Side2);

public override string ToString() => String.Format("[{0}, {1}]", Side1, Side2);

public override bool Equals(object obj)

{

if (obj == null)

return false;

Domino dominoObj = (Domino)obj;

if (dominoObj.Side1 == this.side1 && dominoObj.Side2 == this.side2)

return true;

else

return false;

}

public override int GetHashCode() => ToString().GetHashCode();

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace MTDClasses

{

public class BoneYard

{

private List<Domino> boneyard;

public delegate void EmptyHandler(BoneYard by);

public event EmptyHandler Empty;

public BoneYard() => boneyard = new List<Domino>();

public BoneYard(int maxDots)

{

boneyard = new List<Domino>();

for (int i = 0; i <= maxDots; i++)

for (int o = i; o <= maxDots; o++)

boneyard.Add(new Domino(i, o));

}

public void Shuffle()

{

Random RNG = new Random();

for (int current = 0; current < boneyard.Count; current++)

{

int random = RNG.Next(0, boneyard.Count);

Domino tempDomino = boneyard[current];

boneyard[current] = boneyard[random];

boneyard[random] = tempDomino;

}

}

public bool IsEmpty()

{

if (boneyard.Count == 0)

{

//Empty(this);

return true;

}

else

{

return false;

}

}

public int DominosRemaining => boneyard.Count;

public Domino Draw()

{

if (boneyard.Count == 0)

return null;

Random RNG = new Random();

int i = RNG.Next(0, boneyard.Count);

Domino domino = boneyard[i];

boneyard.RemoveAt(i);

return domino;

}

public Domino this[int index]

{

get => boneyard[index];

set => boneyard[index] = value;

}

public override string ToString()

{

string list = "";

foreach (Domino domino in boneyard)

list += domino.ToString();

return list;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace MTDClasses

{

/// <summary>

/// Represents a generic Train for MTD

/// </summary>

public abstract class Train

{

protected List<Domino> dominos;

protected int engineValue;

public Train()

{

dominos = new List<Domino>();

engineValue = 0;

}

public Train(int engineValue)

{

dominos = new List<Domino>();

EngineValue = engineValue;

}

public int Count => dominos.Count;

/// <summary>

/// The first domino value that must be played on a train

/// </summary>

public int EngineValue

{

get => engineValue;

set => engineValue = value;

}

public bool IsEmpty => (dominos.Count == 0);

public Domino LastDomino => (dominos.Count == 0) ? null : dominos[dominos.Count - 1];

/// <summary>

/// Side2 of the last domino in the train. It's the value of the next domino that can be played.

/// </summary>

public int PlayableValue

{

get

{

if (IsEmpty)

return engineValue;

return LastDomino.Side2;

}

}

public void Add(Domino d) => dominos.Add(d);

public Domino this[int index] => dominos[index];

/// <summary>

/// Determines whether a hand can play a specific domino on this train and if the domino must be flipped.

/// Because the rules for playing are different for Mexican and Player trains, this method is abstract.

/// </summary>

public abstract bool IsPlayable(Hand h, Domino d, out bool mustFlip);

/// <summary>

/// A helper method that determines whether a specific domino can be played on this train.

/// It can be called in the Mexican and Player train class implementations of the abstract method

/// </summary>

protected bool IsPlayable(Domino d, out bool mustFlip)

{

if (IsEmpty)

{

if(engineValue == d.Side1)

{

mustFlip = false;

return true;

}

else if(engineValue == d.Side2)

{

mustFlip = true;

return true;

}

else

{

mustFlip = false;

return false;

}

}

else

{

if(d.Side1 == PlayableValue)

{

mustFlip = false;

return true;

}

else if(d.Side2 == PlayableValue)

{

mustFlip = true;

return true;

}

else

{

mustFlip = false;

return false;

}

}

}

virtual public void Play(Hand h, Domino d)

{

bool mustFlip = false;

if (IsPlayable(d, out mustFlip))

{

if (mustFlip)

d.Flip();

dominos.Add(d);

}

else

throw new Exception("That domino isn't playable!");

}

public override string ToString()

{

string list = "";

foreach (Domino domino in dominos)

list += domino.ToString();

return list;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace MTDClasses

{

public class MexicanTrain : Train

{

public MexicanTrain() : base()

{

engineValue = 0;

}

public MexicanTrain(int engineValue) : base(engineValue){}

public override bool IsPlayable(Hand h, Domino d, out bool mustFlip) => base.IsPlayable(d, out mustFlip);

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace MTDClasses

{

public class PlayerTrain : Train

{

private bool open = false;

private Hand player;

public PlayerTrain() : base(){}

public PlayerTrain(Hand h) : base()

{

player = h;

}

/// <summary>

/// This is the most appropriate constructor for the class.

/// </summary>

/// <param name="h">Represents the Hand object to which the train belongs</param>

/// <param name="engineValue">Represents the first playable value on the train</param>

public PlayerTrain(Hand h, int engineValue) : base (engineValue)

{

player = h;

}

/// <summary>

/// Returns whether or not the train is open. An open train

/// can be played upon by any player.

/// </summary>

public bool IsOpen => open;

/// <summary>

/// Open the train

/// </summary>

public void Open() => open = true;

/// <summary>

/// Close the train

/// </summary>

public void Close() => open = false;

/// <summary>

/// Can the domino d be played by the hand h on this train?

/// If it can be played, must it be flipped to do so?

/// </summary>

/// <param name="d"></param>

/// <param name="mustFlip"></param>

/// <param name="h"></param>

/// <returns></returns>

public override bool IsPlayable(Hand h, Domino d, out bool mustFlip)

{

if(IsOpen || player == h)

{

if(base.IsPlayable(d, out mustFlip))

return true;

else

return false;

}

else

{

mustFlip = false;

return false;

}

}

public override void Play(Hand h, Domino d)

{

if (h != player && !IsOpen)

throw new ArgumentException("The train is closed!");

else

{

base.Play(h, d);

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace MTDClasses

{

/// <summary>

/// Represents a hand of dominos

/// </summary>

public class Hand

{

/// <summary>

/// The list of dominos in the hand

/// </summary>

private List<Domino> dominos;

public delegate void EmptyHandler(Hand hand);

public event EmptyHandler Empty;

/// <summary>

/// Creates an empty hand

/// </summary>

public Hand() => dominos = new List<Domino>();

/// <summary>

/// Creates a hand of dominos from the boneyard.

/// The number of dominos is based on the number of players

/// 2–4 players: 10 dominoes each

/// 5–6 players: 9 dominoes each

/// 7–8 players: 7 dominoes each

/// </summary>

/// <param name="by"></param>

/// <param name="numPlayers"></param>

public Hand(BoneYard by, int numPlayers)

{

dominos = new List<Domino>();

int quantity = 0;

if (numPlayers >= 2 && numPlayers <= 4)

quantity = 10;

else if (numPlayers >= 5 && numPlayers <= 6)

quantity = 9;

else if (numPlayers >= 7 && numPlayers <= 8)

quantity = 7;

else

quantity = 5;

for (int i = 0; i < quantity; i++)

dominos.Add(by.Draw());

}

public void Add(Domino d) => dominos.Add(d);

public int Count => dominos.Count;

public bool IsEmpty

{

get

{

if(dominos.Count == 0)

{

//Empty(this);

return true;

}

else

{

return false;

}

}

}

/// <summary>

/// Sum of the score of each domino in the hand

/// </summary>

public int Score

{

get

{

int score = 0;

foreach (Domino domino in dominos)

score += domino.Score;

return score;

}

}

/// <summary>

/// Does the hand contain a domino with value in side 1 or side 2?

/// </summary>

/// <param name="value">The number of dots on one side of the domino that you're looking for</param>

public bool HasDomino(int value)

{

foreach (Domino domino in dominos)

if (domino.Side1 == value || domino.Side2 == value)

return true;

return false;

}

/// <summary>

/// DOes the hand contain a double of a certain value?

/// </summary>

/// <param name="value">The number of (double) dots that you're looking for</param>

public bool HasDoubleDomino(int value)

{

foreach (Domino domino in dominos)

if (domino.Side1 == value && domino.Side2 == value)

return true;

return false;

}

/// <summary>

/// The index of a domino with a value in the hand

/// </summary>

/// <param name="value">The number of dots on one side of the domino that you're looking for</param>

/// <returns>-1 if the domino doesn't exist in the hand</returns>

public int IndexOfDomino(int value)

{

for(int i = 0; i < dominos.Count; i++)

if (dominos[i].Side1 == value || dominos[i].Side2 == value)

return i;

return -1;

}

/// <summary>

/// The index of the do

/// </summary>

/// <param name="value">The number of (double) dots that you're looking for</param>

/// <returns>-1 if the domino doesn't exist in the hand</returns>

public int IndexOfDoubleDomino(int value)

{

for(int i = 0; i < dominos.Count; i++)

if (dominos[i].Side1 == value && dominos[i].Side2 == value)

return i;

return -1;

}

/// <summary>

/// The index of the highest double domino in the hand

/// </summary>

/// <returns>-1 if there isn't a double in the hand</returns>

public int IndexOfHighDouble()

{

int highest = -1;

for(int i = 0; i < dominos.Count; i++)

{

if(dominos[i].IsDouble() && dominos[i].Side1 > highest)

{

highest = i;

}

}

return highest;

}

public Domino this[int index]

{

get

{

if(index < 0 || index >= dominos.Count)

throw new ArgumentOutOfRangeException("Invalid index!");

return dominos[index];

}

}

public void RemoveAt(int index) => dominos.RemoveAt(index);

/// <summary>

/// Finds a domino with a certain number of dots in the hand.

/// If it can find the domino, it removes it from the hand and returns it.

/// Otherwise it returns null

/// </summary>

/// <param name="value"></param>

/// <returns></returns>

public Domino GetDomino(int value)

{

int index = IndexOfDomino(value);

if (index == -1)

return null;

Domino found = dominos[index];

dominos.RemoveAt(index);

if (found.Side1 == value)

return found;

else

{

found.Flip();

return found;

}

}

/// <summary>

/// Finds a domino with a certain number of double dots in the hand.

/// If it can find the domino, it removes it from the hand and returns it.

/// Otherwise it returns null

/// </summary>

/// <param name="value"></param>

/// <returns></returns>

public Domino GetDoubleDomino(int value)

{

int index = IndexOfDoubleDomino(value);

if (index == -1)

return null;

else

{

Domino domino = dominos[index];

dominos.RemoveAt(index);

return domino;

}

}

/// <summary>

/// Draws a domino from the boneyard and adds it to the hand

/// </summary>

/// <param name="by"></param>

public void Draw(BoneYard by) => dominos.Add(by.Draw());

/// <summary>

/// Plays the domino at the index on the train.

/// Flips the domino if necessary before playing.

/// Removes the domino from the hand.

/// Throws an exception if the domino at the index

/// is not playable.

/// </summary>

/// <param name="index"></param>

/// <param name="t"></param>

private void Play(int index, Train t)

{

bool mustFlip = false;

Domino domino = dominos[index];

if(t.IsPlayable(this, domino, out mustFlip))

{

dominos.RemoveAt(index);

if (mustFlip)

domino.Flip();

t.Play(this, domino);

}

else

throw new ArgumentException("That domino can't be played on this train!");

}

/// <summary>

/// Plays the domino from the hand on the train.

/// Flips the domino if necessary before playing.

/// Removes the domino from the hand.

/// Throws an exception if the domino is not in the hand

/// or is not playable.

/// </summary>

public void Play(Domino d, Train t)

{

int index = IndexOfDomino(t.PlayableValue);

if(index == -1)

throw new ArgumentException("That domino isn't in the hand!");

t.Play(this, d);

}

/// <summary>

/// Plays the first playable domino in the hand on the train

/// Removes the domino from the hand.

/// Returns the domino.

/// Throws an exception if no dominos in the hand are playable.

/// </summary>

/// <param name="t"></param>

/// <returns></returns>

public Domino Play(Train t)

{

int index = IndexOfDomino(t.PlayableValue);

if (index == -1)

throw new Exception("No domino in this hand can be played.");

else

{

Domino domino = dominos[index];

t.Play(this, domino);

return domino;

}

}

public override string ToString()

{

string list = "";

foreach (Domino domino in dominos)

list += domino.ToString();

return list;

}

}

}