**Peer Evaluation for Lab 6 – Chapter 13**

|  |  |
| --- | --- |
| Your name: (Your lab is the one being evaluated) | Amanda Akins |
| Name(s) of peer evaluator(s) | Me |
| Date: | 06/03/16 |

Instructions  
You should have already completed Lab 6. 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.

|  |  |
| --- | --- |
| ***In Class Exercises – CustomeList Class – Exercise 13.1 (NOT THE UI EXERCISES)*** | |
| Completed CustomerList Class?   * Created a class diagram in visual studio? Screen shot included? * Instance variables are camelCase and private? * Property/Method names are TitleCase? Properties/Methods are public? * Implements all properties/methods in the specification in the chart on page 419? * Implements + and - operators? * Implements delegate named ChangeHandler and Changed event for extra credit? * Implements any other methods/operators for extra credit? What are they? * Completed Customer Tests? Tests all properties and methods in the class? Tests + and – operators? DOESN’T NEED TO TEST the delegate and event. Screen shot is included? * Source code includes CustomerList class as well as test program? |  |
| ***In Class Exercises – Deck and Hand classes*** | |
| Completed Deck Class?   * Created a class diagram in visual studio? Screen shot included? * Instance variables are camelCase and private? * Property/Method names are TitleCase? Properties/Methods are public? * Implements all properties/methods in the specification on moodle? * Implements any other methods/operators for extra credit? What are they? * Completed Deck Tests? Tests all properties and methods in the class? Screen shot is included? * Source code includes Deck class as well as test program?   Completed Hand Class?   * Created a class diagram in visual studio? Screen shot included? * Instance variables are camelCase and private? * Property/Method names are TitleCase? Properties/Methods are public? * Implements all properties/methods in the specification on moodle? * Implements any other methods/operators for extra credit? What are they? * Completed Hand Tests? Tests all properties and methods in the class? Screen shot is included? * Source code includes Hand class as well as test program? |  |

|  |  |
| --- | --- |
| ***Programming style for all programs*** | |
| Is proper indentation used? Is each property/method indented properly? Is each control structure 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?) |  |

General comments and notes: For Customer Maintenance I commented out what I thought I needed to do for the set accessor argument exception thingers… They aren’t working right, I don’t think. It adds something over 30 characters, but if you try and run it again, then it throws the exception and won’t run. I also wasn’t sure how to test the indexers or save :/. Screen Shots and Source Code

**CUSTOMER MAINTENANCE –**

**CustomerTests:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using CustomerMaintenanceClasses;

namespace CustomerTests

{

class Program

{

static void Main(string[] args)

{

//TestCustomerConstructors();

//TestCustomerPropertyGetters();

//TestCustomerPropertySetters();

//TestCustomerListConstructor();

//TestCustomerListFill();

//TestCustomerAdd();

//TestCustomerRemove();

//TestCustomerListIndexers();

//TestCustomerListCount();

//TestCustomerListSave();

Console.WriteLine("Press the enter key"); // writes to screen

Console.ReadLine(); // waits for user to hit the "enter" key, allows program to stay open long enough to see what happened

}

public static void TestCustomerListConstructor()

{

CustomerList list = new CustomerList();

Console.WriteLine("Testing the constructor");

Console.WriteLine("Count. Expect 0 " + list.Count);

Console.WriteLine("ToString. Expect nothing " + list);

}

public static void TestCustomerAdd()

{

CustomerList list = new CustomerList();

Customer c1 = new Customer("Mickey", "Mouse", "mmouse@disney.com");

Customer c2 = new Customer("Donald", "Trump", "makeamericagreatagain@rep.com");

list.Add(c1);

list.Add(c2);

Console.WriteLine("Testing Add");

Console.WriteLine("Count. Expect 2 " + list.Count);

Console.WriteLine("ToString. Expect mickey and donald " + list);

}

public static void TestCustomerRemove()

{

CustomerList list = new CustomerList();

Customer c1 = new Customer("Mickey", "Mouse", "mmouse@disney.com");

Customer c2 = new Customer("Donald", "Trump", "makeamericagreatagain@rep.com");

list.Add(c1);

list.Add(c2);

list.Remove(c2);

Console.WriteLine("Testing Remove");

Console.WriteLine("Count. Expect 1 " + list.Count);

Console.WriteLine("ToString. Expect mickey " + list);

}

static void TestCustomerConstructors()

{

Customer c1 = new Customer(); // creating new product, () = default parameter

Customer c2 = new Customer("Manda", "Akins", "AAkins83@gmail.com"); // creating another product using overloaded constructor via 3 parameters

Console.WriteLine("Testing both constructors"); // to get something to show on screen use "WriteLine"

Console.WriteLine("Default constructor. Expecting ??? ???, ???. " + c1); // testing default contructor, expecting default values which are nothing, turns p1 into string and prints on screen (nothing)

Console.WriteLine("Overloaded constructor. Expecting Manda Akins, AAkins83@gmail.com. " + c2); // testing overloaded constructor,

Console.WriteLine(); // () = will print a blank line

}

static void TestCustomerPropertyGetters()

{

Customer c1 = new Customer("Manda", "Akins", "AAkins83@gmail.com");

Console.WriteLine("Testing getters"); //not changing value

Console.WriteLine("First Name. Expecting Manda. " + c1.FirstName);

Console.WriteLine("Last Name. Expecting Akins. " + c1.LastName);

Console.WriteLine("Email Address. Expecting AAkins83@gmail.com. " + c1.Email);

Console.WriteLine();

}

static void TestCustomerPropertySetters()

{

Customer c1 = new Customer("Manda", "Akins", "AAkins83@gmail.com");

Console.WriteLine("Testing setters");

c1.FirstName = "Manda"; //changing property

c1.LastName = "Akins";

c1.Email = "AAkins83@gmail.com";

Console.WriteLine("Expecting Manda, Akins, AAkins83@gmail.com. " + c1);

Console.WriteLine();

}

static void TestCustomerListFill()

{

CustomerList cList = new CustomerList();

cList.Fill();

Console.WriteLine("Testing Fill");

Console.WriteLine("Expecting list of customers:\n" + cList);

Console.WriteLine();

}

static void TestCustomerListCount()

{

CustomerList cList = new CustomerList();

cList.Fill();

Console.WriteLine("Expecting 1 " + cList.Count);

}

}

}

**Customer:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CustomerMaintenanceClasses

{

public class Customer

{

// Instance variables - "private" for data hiding

private string fName;

private string lName;

private string eMail;

public Customer()

{

fName = "???";

lName = "???";

eMail = "???";

}

public Customer(string firstName, string lastName, string eMailAddress)

{

fName = firstName;

lName = lastName;

eMail = eMailAddress;

}

public string FirstName

{

get

{

return fName;

}

set

{

//if (value.Length > 30)

//throw new ArgumentOutOfRangeException("Must be less than 30 characters.");

fName = value;

}

}

public string LastName

{

get

{

return lName;

}

set

{

;// if (value.Length > 30)

//throw new ArgumentOutOfRangeException("Must be less than 30 characters.");

lName = value;

}

}

public string Email

{

get

{

return eMail;

}

set

{

//if (value.Length > 30)

//throw new ArgumentOutOfRangeException("Must be less than 30 characters.");

eMail = value;

}

}

public string GetDisplayText()

{

return fName + " " + lName + ", " + eMail;

}

public override string ToString()

{

return GetDisplayText();

}

}

}

**CustomerList:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace CustomerMaintenanceClasses

{

public class CustomerList

{

private List<Customer> customers = new List<Customer>();

public CustomerList() { }

public int Count

{

get

{

return customers.Count;

}

}

public void Add(Customer bob)

{

customers.Add(bob);

}

public void Remove(Customer bob)

{

customers.Remove(bob);

}

public void Fill()

{

customers = CustomerDB.GetCustomers();

}

public void Save()

{

CustomerDB.SaveCustomers(customers);

}

public override string ToString()

{

string output = "";

foreach (Customer c in customers)

{

output += c.ToString() + "\n";

}

return output;

}

public static CustomerList operator +(CustomerList cl, Customer c)

{

cl.Add(c);

return cl;

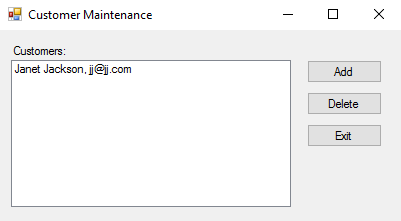
}

public static CustomerList operator -(CustomerList cl, Customer c)

{

cl.Remove(c);

return cl;



**CONCENTRATION:**

**boardForm:**

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 CardClassLibrary;

namespace Concentration

{

public partial class boardForm : Form

{

public boardForm()

{

InitializeComponent();

}

#region Instance Variables

//\* You'll need an array of strings named cards, 2 indexes and the number of matches

//\* My array is set up to hold 21 strings and I ignore the 0 element because I named the picture boxes

//\* starting at 1 rather than 0

Card[] cards = new Card[21];

int matches = 0;

int firstPick = NOT\_PICKED\_YET, secondPick = NOT\_PICKED\_YET; // stores first and second indexes

const int NOT\_PICKED\_YET = -1;

// this is for the timer that displays the elapsed time. I included it for illustration purposed only.

private int seconds = 0;

#endregion

#region Methods

//\* Write some methods

// you may need more methods but here's the list that I used

// this should fill the cards (array of strings) with card file names

private void FillCards()

{

int index = 1;

for (int suit = 1; suit <= 4; suit++)

{

for (int value = 1; value <= 5; value++)

{

// \*Add a string for the filename of the card

// to the array. The first card, for example, should be

// cards[1] = "cardac.jpg";

cards[index] = new Card(value, suit);

index++;

}

}

}

//\* determines if the 2 cards are a match. The "value" of the card is in the filename at the 5th char.

private bool IsMatch(int index1, int index2)

{

if (cards[index1].HasMatchingValue(cards[index2]))

return true;

else

return false;

}

//\* Shuffles the strings in the cards array. I'll give you an algorithm in class for this.

private void ShuffleCards()

{

Random randomNumberGenerator = new Random();

for (int index = 1; index <= 20; index++)

{

int rndIndex = randomNumberGenerator.Next(1, 21);

Card temp = cards[index];

cards[index] = cards[rndIndex];

cards[rndIndex] = temp;

}

}

// loads the specified card picture box with the filename from the array

private void LoadCard(int i)

{

PictureBox card = (PictureBox)this.Controls["card" + i];

card.Image = Image.FromFile(System.Environment.CurrentDirectory + "\\Cards\\" + cards[i].Filename);

}

//\* loads an image for the back of a card in the specified card

private void LoadCardBack(int i)

{

PictureBox card = (PictureBox)this.Controls["card" + i];

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

}

//\* disables and hides the specified card - set visible property to false

private void HideCard(int i)

{

PictureBox card = (PictureBox)this.Controls["card" + i];

card.Visible = false;

}

//\* disables and hides all cards. Call HideCard in a loop.

private void HideAllCards()

{

for (int i = 1; i <= 20; i++)

{

HideCard(i);

}

}

//\* disables the specified card

private void DisableCard(int i)

{

PictureBox card = (PictureBox)this.Controls["card" + i];

card.Enabled = false;

}

//\* disables all cards. Call DisableCard in a loop.

private void DisableAllCards()

{

for (int i = 1; i <= 20; i++)

{

DisableCard(i);

}

}

private void ShowCard(int i)

{

PictureBox card = (PictureBox)this.Controls["card" + i];

card.Visible = true;

}

//\* enables and shows all of the cards, make picture box

private void ShowAllCards()

{

for (int i = 1; i <= 20; i++)

{

ShowCard(i);

}

}

//\* enables the cards that are still visible on the board

private void EnableAllVisibleCards()

{

for (int i = 1; i <= 20; i++)

{

PictureBox card = (PictureBox)this.Controls["card" + i];

if (card.Visible == true)

{

card.Enabled = true;

}

else

{

card.Enabled = false;

}

}

}

#endregion

//\* finish this

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

{

gameTimer.Enabled = true;

lblElapsedTime.Text = seconds.ToString();

// fill the cards array - you have a method

FillCards();

// shuffle the cards.

ShuffleCards();

// show the back of all cards - you have a method

for (int i = 1; i <= 20; i++)

{

LoadCardBack(i);

}

}

// This is only for illustration purposes

private void gameTimer\_Tick(object sender, EventArgs e)

{

seconds++;

lblElapsedTime.Text = seconds.ToString();

}

//\* Finish this. See pseudocode below.

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

{

PictureBox card = (PictureBox)sender;

int cardIndex = int.Parse(card.Name.Substring(4));

/\* if the first card is not picked yet

\* save the index

\* load the card image

\* disable the card so the user can't click the same card twice

\* else (the first card has been picked so this is the second card)

\* save the index of the second card

\* load the card image

\* disable all the cards

\* start the flip timer - flipTimer.Enabled = true;

\* end if

\*/

if (firstPick == NOT\_PICKED\_YET)

{

firstPick = cardIndex;

LoadCard(firstPick);

DisableCard(firstPick);

}

else

{

secondPick = cardIndex;

LoadCard(secondPick);

DisableAllCards();

flipTimer.Enabled = true;

}

}

//\* ToDo: Finish this. See pseudocode below.

private void flipTimer\_Tick(object sender, EventArgs e)

{

/\* stop the timer

\* if there's a match

\* hide the first card clicked

\* hide the second card clicked

\* reset both indexes

\* increment the number of matches

\* if the game is over

\* if the user wants to play again

\* shuffle the cards - ShuffleCards();

\* show the card backs and enable all of the cards - LoadCardBack();

\* else

\* exit the application?

\* end if

\* else

\* enable all of the cards left on the board - EnableAllCards();

\* end if

\* else

\* show the back of the first card clicked

\* show the back of the second card clicked

\* reset both indexes

\* enable all of the cards left on the board

\* end if

\*/

flipTimer.Enabled = false;

if (IsMatch(firstPick, secondPick))

{

HideCard(firstPick);

HideCard(secondPick);

firstPick = NOT\_PICKED\_YET;

secondPick = NOT\_PICKED\_YET;

matches++;

if (matches == 10)

{

MessageBox.Show("WOOHOO!");

}

}

else

{

LoadCardBack(firstPick);

LoadCardBack(secondPick);

firstPick = NOT\_PICKED\_YET;

secondPick = NOT\_PICKED\_YET;

}

EnableAllVisibleCards();

}

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

{

EnableAllVisibleCards();

}

}

}

**Deck:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CardClassLibrary

{

public class Deck

{

private List<Card> cards = new List<Card>();

public Deck()

{

int index = 0;

for (int suit = 1; suit <= 4; suit++)

{

for (int value = 1; value <= 13; value++)

{

cards.Add(new Card(value, suit));

index++;

}

}

}

public int NumCards

{

get

{

return cards.Count;

}

}

public int Count()

{

int i = 0;

foreach (Card count in cards)

{

i++;

}

return i;

}

public void Shuffle()

{

Random randomNumberGenerator = new Random();

for (int index = 0; index < cards.Count ; index++)

{

int rndIndex = randomNumberGenerator.Next(0, cards.Count);

Card temp = cards[index];

cards[index] = cards[rndIndex];

cards[rndIndex] = temp;

}

}

public Card Deal()

{

Card c1 = cards[0];

cards.RemoveAt(0);

return c1;

}

**Hand:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CardClassLibrary

{

public class Hand

{

private List<Card> cards = new List<Card>();

public Hand() { }

public int Count()

{

int i = 0;

foreach (Card count in cards)

{

i++;

}

return i;

}

public int NumCards

{

get

{

return cards.Count;

}

}

public Card GetCard(int index)

{

return cards[index];

}

public int IndexOf(Card newCard)

{

return cards.IndexOf(newCard);

}

public int IndexOf(int value)

{

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

{

if (cards[i].Value == value) // i is index of card in the cards list. Value is the property of the card.

return i;

}

return -1; // return if out of scope

}

public int IndexOf(int value, int suit)

{

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

{

if (cards[i].Value == value && cards[i].Suit == suit) // i is index of a card in cards list. Value is the property of

{

return i;

}

}

return -1; // return if out of scope \*/

}

/\*public int IndexOf(int value, int suit)

{

Card c = new Card(value, suit);

return IndexOf(c);

}\*/

public bool HasCard(Card searchCard)

{

foreach (Card cardInHand in cards)

{

if (searchCard.Value == cardInHand.Value && searchCard.Suit == cardInHand.Suit)

return true;

}

return false;

}

public bool HasCard(int searchValue)

{

foreach (Card cardInHand in cards)

{

if (searchValue == cardInHand.Value)

return true;

}

return false;

}

public bool HasCard(int searchValue, int searchSuit)

{

foreach (Card c in cards)

{

if (searchValue == c.Value && searchSuit == c.Suit)

return true;

}

return false;

}

public void Add(Card card)

{

cards.Add(card);

}

public Card Discard(int index)

{

Card c1 = cards[index];

cards.RemoveAt(index);

return c1;

}

public override string ToString()

{

string output = "";

foreach (Card c in cards)

{

output += c.ToString() + "\n";

}

return output;

}

}

}

**CardClassTests:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using CardClassLibrary;

namespace CardClassTests

{

class Program

{

static void Main(string[] args)

{

TestCardConstructors();

TestDeckMethods();

TestHandMethods();

Console.WriteLine();

Console.ReadLine();

}

static void TestHandMethods()

{

Hand h = new Hand();

Card c1 = new Card(1, 4);

Card c2 = new Card(12, 3);

Card c3 = new Card(2, 2);

Console.WriteLine("Testing Hand Methods");

Console.WriteLine("Testing NumCards. Expecting 0. " + h.Count());

h.Add(c1);

h.Add(c2);

Console.WriteLine("Adding 2 cards. Expecting Ace of Spades and Queen of Hearts. " + c1 + " " + c2);

Console.WriteLine("Testing NumCards. Expecting 2 " + h.Count());

Console.WriteLine("Testing IndexOf(Card). Expecting 0 1. " + h.IndexOf(c1) + " " + h.IndexOf(c2));

Console.WriteLine("Testing IndexOf(value). Expecting 0 1. " + h.IndexOf(1) + " " + h.IndexOf(12));

Console.WriteLine("Testing IndexOf(value, suit). Expecting 0 1. " + h.IndexOf(1, 4) + " " + h.IndexOf(12, 3));

Console.WriteLine("Testing HasCard(Card). Expecting True False. " + h.HasCard(c1) + " " + h.HasCard(c3));

Console.WriteLine("Testing HasCard(value). Expecting True False. " + h.HasCard(1) + " " + h.HasCard(2));

Console.WriteLine("Testing HasCard(value, suit). Expecting True False. " + h.HasCard(1, 4) + " " + h.HasCard(2, 2));

h.Discard(0);

Console.WriteLine("Testing Discard. Expect 1. " + h.NumCards);

}

static void TestDeckMethods()

{

Deck d1 = new Deck();

Console.WriteLine("Testing Deck methods");

Console.WriteLine("Testing NumCards. Expecting 52 " + d1.Count());

Console.WriteLine("Testing IsEmpty. Expecting false " + d1.IsEmpty());

Console.WriteLine("Testing Deal. Expect Ace of Clubs. " + d1.Deal());

d1.Shuffle();

Console.WriteLine("Testing Shuffle. Should display a new card. " + d1.Deal());

Console.WriteLine("Here is my deck." + d1);

}

static void TestCardConstructors()

{

Card c1 = new Card();

Card c2 = new Card(13, 4);

Console.WriteLine("Testing the constructors");

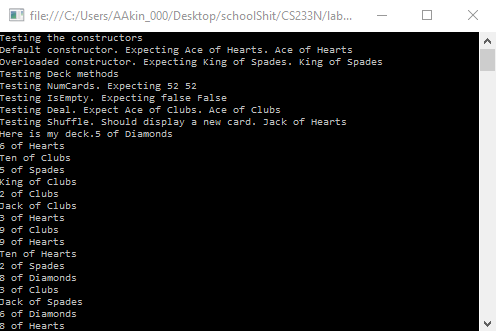
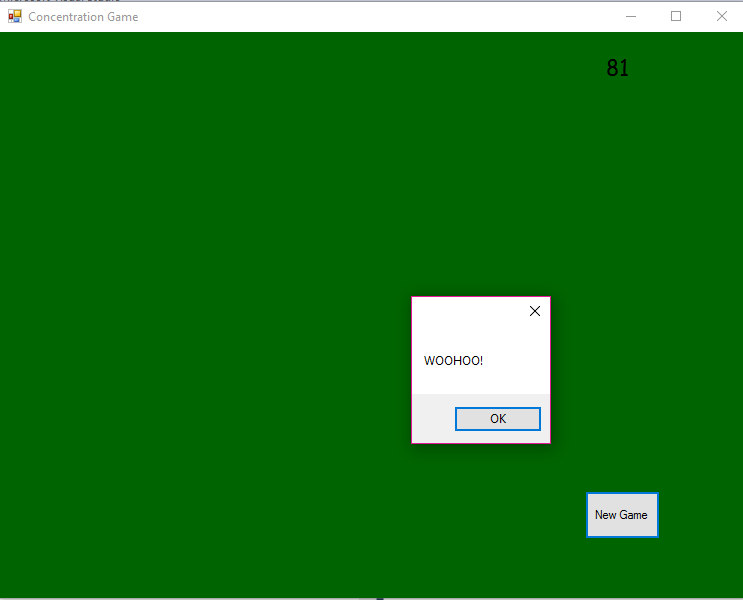
Console.WriteLine("Default constructor. Expecting Ace of Hearts. " + c1);

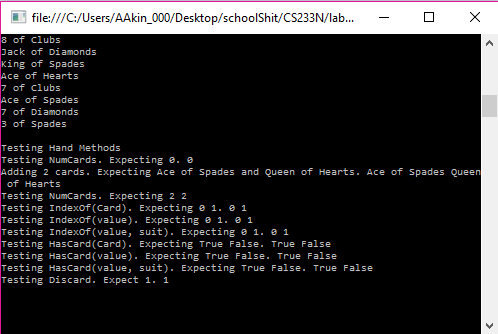
Console.WriteLine("Overloaded constructor. Expecting King of Spades. " + c2);

}

}

}

****

****