Class Assignment – Card Games

Specification Part C – **Game Logic and Functionality**

Submission Due: September 28th (all parts together)

Part C Overview

In this part of the assignment, you will implement the logic class **CrazyEights** in the **Games** project and use it to complete the functionality of the **Crazy_Eights_Form** in the **GUI Project**.

The **CrazyEights** logic class is designed to completely handle the *game logic* and store the *game data* (this is called the business layer). The **GUI** allows the user to view this data and to make requests to the **CrazyEights** class when behaviour or data about the game is needed (this is called the presentation layer). **For this reason, the GUI should NOT contain any private or public class variables. Instead, the logic class should contain all the data necessary.**

Contents

Crazy Eights Rules

This section describes the **rules of Crazy Eights**. It is important that you understand these rules to implement the logic class correctly.

The aim of the game is to get rid of all the cards in your hand. Here are the rules for two players (human against computer) that your game will follow.

1. Each player is dealt eight cards at the start. The remaining cards are placed face down on the table, forming a draw pile. The top card of the draw pile is turned face up to start the discard pile.
2. The human player always goes first.
3. You can play any card of the *same suit* as the one at the top of the discard pile. E.g. there's a Club at the top of the pile, you can play any Club.
4. You can play any card of the *same face-value* as the one at the top of the discard pile. E.g. there's a Six of Clubs at the top of the pile, you can play any other Six.
5. You can play an Eight (of any suit) on top of any card, and you must then nominate the suit which must be played next (by the Computer). The game will prompt you for the suit after you play the Eight. E.g. there's a Six of Clubs at the top of the discard pile, you play an Eight of Diamonds and change the suit to Hearts. The next player then must play a Heart (or another Eight).
6. If the first card on the table is an Eight, then you can put down any card from your hand.
7. You may play only one card at a time.
8. If you have no card that you can play, you must draw a card from the draw pile. But you can't draw if a card in your hand is playable. If you've drawn a card and now have 13 cards in your hand, and you still can't play, then you must pass on your turn to the other player.
9. If the other player has a full hand and cannot play, and you also reach a full hand and cannot play (i.e. nobody can play), then your turn ends, and the discard and draw piles are combined, shuffled, and laid out again until the other player can play.
10. If all the cards in the draw pile have been taken, and a player needs to draw another card, then the pile of discarded cards is flipped to become the draw pile. The top card of the draw pile is turned face up to start the discard pile. Rule 6 applies (again).
11. The winner is the first player who gets rid of all the cards in their hand.

When the Computer is playing, it follows the same rules as above, with the following extra rules:

12. Eights are only played when no other cards can be played (if more than one Eight is available, then any one of those is played, without further consideration)
13. Once an Eight is played, the suit of that Eight now applies (e.g. when the Computer plays the Eight of Diamonds, the suit is now Diamonds – the Computer always chooses the suit the card already has).
14. When the Computer has a choice between playing a card of the *same suit* (rule 3) and playing one of the *same face-value* (rule 4), it always plays one of the *same face-value* (if more than one of the *same face-value* is available, then any one of those is played, without further consideration).

CrazyEights Logic Class Specification and UML

ActionResult Enum

Declare the following enum ***inside*** the **CrazyEights** class:

```
public enum ActionResult {
    /// <summary>
    /// A card was played that won the game
    /// </summary>
    WinningPlay,
    /// <summary>
    /// A valid card was played
    /// </summary>
    ValidPlay,
    /// <summary>
    /// A suit is required to continue play
    /// </summary>
    SuitRequired,
    /// <summary>
    /// Attempted to play an invalid card
    /// </summary>
    InvalidPlay,
    /// <summary>
    /// Attempted to play an invalid card when no cards can be played
    /// </summary>
    InvalidPlayAndMustDraw,
    /// <summary>
    /// A valid card was played, and the other player cannot play
    /// </summary>
    ValidPlayAndExtraTurn,
    /// <summary>
    /// Drew a playable card
    /// </summary>
    DrewPlayableCard,
    /// <summary>
    /// Drew an unplayable card
    /// </summary>
    DrewUnplayableCard,
    /// <summary>
    /// Drew an unplayable card and filled the hand
    /// </summary>
    DrewAndNoMovePossible,
    /// <summary>
    /// Drew an unplayable card and filled the hand, leaving both
    /// players unable to play, so piles were reset so that the
    /// the other player can continue play (rule 9)
    /// </summary>
    DrewAndResetPiles,
    /// <summary>
    /// Attempted to draw a card while moves were still possible
    /// </summary>
    CannotDraw,
    /// <summary>
    /// Flipped the discard pile to use as the new draw pile (rule 10)
    /// </summary>
    FlippedDeck
}
```

CrazyEights UML and Behaviour Details

Implement the UML diagram below in the **CrazyEights** class within the **Games** project. You must implement all the characteristics of the UML except for the optional private methods. The optional methods are highly recommended, but you may alter or change them if you wish. You may also add additional private variables or methods, but **you may not add additional public methods, variables, or properties.**

You can test the correctness of your UML using the unit tests in **4_GamesTests** in the *Test Explorer*.

Do not begin working on this UML until all unit tests pass for part A (including *1_CardTests*, *2_HandTests*, and *3_CardPileTests*).

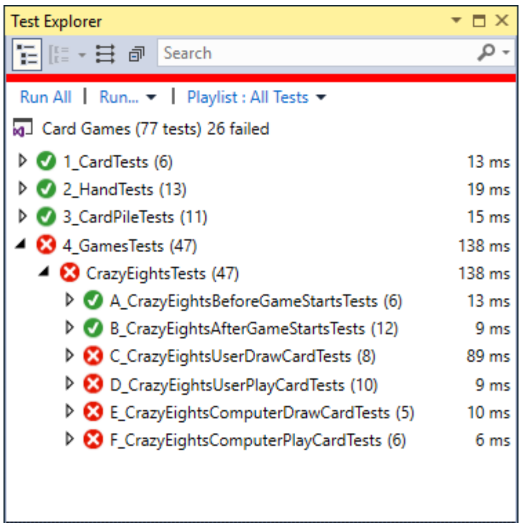
+ class CrazyEights
- _drawPile : CardPile - _discardPile : CardPile «+get» +TopDiscard : Card «+get» +IsDrawPileEmpty : bool «+get» «-set» +ComputerHand : Hand «+get» «-set» +UserHand : Hand «+get» «-set» +IsUserTurn : bool «+get» «-set» +IsPlaying : bool
+StartGame() +StartGame(Hand userHand, Hand computerHand, CardPile discardPile, CardPile drawPile) +SortUserHand() +UserDrawCard() : ActionResult +UserPlayCard(int cardNum, Suit? chosenSuit = null) : ActionResult +ComputerAction() : ActionResult *Optional:* *-PlayCard(Hand hand, int cardNum, Suit newSuit) : ActionResult* *-DrawCard(Hand hand) : ActionResult* *-IsHandPlayable(Hand hand) : bool* *-IsCardPlayable(Card card) : bool*

Private Attributes	_drawPile	Represents the pile of draw cards.
	_discardPile	Represents the pile of discarded cards.
Properties	TopDiscard	Accesses the card at the top of the discard pile.
	IsDrawPileEmpty	Returns true if the draw pile is empty.
	ComputerHand	Represents the computer's hand.
	UserHand	Represents user's hand.
	IsUserTurn	Used to track which player's turn it is. True if it is the user's turn.
	IsPlaying	Used to track if a game is in progress. True if a game is in progress.

Methods	StartGame()	Sets up a game of Crazy Eights according to the normal rules.
	StartGame(…)	Sets up a game of Crazy Eights using the given hands and card piles. The user still plays first.
	SortUserHand()	Sorts the user's hand.
	UserDrawCard()	1. Throw an exception if a game has not started 2. Throw an exception if it is not the user's turn 3. Attempt to draw a card for the user, performing actions according to the rules. This method may return any of the following: • **ActionResult.CannotDraw** (the user has a playable card already and cannot draw or flip the deck) • **ActionResult.DrewPlayableCard** (the user drew a playable card) • **ActionResult.DrewUnplayableCard** (the user drew an unplayable card) • **ActionResult.DrewAndNoMovePossible** (the user drew an unplayable card and filled their hand with no possible moves, forcing them to end their turn) • **ActionResult.DrewAndResetPiles** (the user drew an unplayable card and filled their hand with no possible moves, but the computer also has a full hand with no possible moves, so the two piles were combined, shuffled, and reset so that the computer can continue play) • **ActionResult.FlippedDeck** (the draw pile was empty, so the discard pile was flipped instead of drawing)
	UserPlayCard(…)	1. Throw an exception if a game has not started 2. Throw an exception if it is not the user's turn 3. If a Suit is given and the FaceValue of the chosen card is Eight, then the Suit of the chosen card will change to the given Suit. 4. Attempt to play the chosen card for the user according to the rules of the game. This method may return any of the following: • **ActionResult.SuitRequired** (an Eight was selected but no Suit was given) • **ActionResult.WinningPlay** (the chosen card was played successfully, winning the game for the user) • **ActionResult.ValidPlayAndExtraTurn** (the chosen card was played successfully, and the Computer cannot play, so an extra turn is given to the user) • **ActionResult.ValidPlay** (the chosen card was played successfully, ending the turn for the user) • **ActionResult.InvalidPlayAndMustDraw** (the chosen card is not allowed to be played, and in fact none of the user's cards can be played) • **ActionResult.InvalidPlay** (the chosen card is not allowed to be played, and a different card should be chosen instead)

	ComputerAction()	This method performs an action as part of the computer’s turn. It is not necessarily a ‘whole turn’. This may involve drawing a card or playing a card, depending on the state of the game. Remember, drawing a card is only possible if no cards can be played. Read the rules of the game for information on how the computer decides actions. 1. Throw an exception if a game has not started 2. Throw an exception if it is not the computer’s turn 3. Perform an action according to the computer’s rules. This method may return any of the following ActionResults (keep in mind that the computer should never attempt an invalid play, so the InvalidPlay and CannotDraw ActionResults are not seen here): • **ActionResult.DrewPlayableCard** (the computer drew a playable card) • **ActionResult.DrewUnplayableCard** (the computer drew an unplayable card) • **ActionResult.DrewAndNoMovePossible** (the computer drew an unplayable card and filled their hand with no possible moves, forcing them to end their turn) • **ActionResult.DrewAndResetPiles** (the computer drew an unplayable card and filled their hand with no possible moves, but the user also has a full hand with no possible moves, so the two piles were combined, shuffled, and reset so that the user can continue play) • **ActionResult.FlippedDeck** (the draw pile was empty, so the discard pile was flipped instead of drawing) • **ActionResult.WinningPlay** (the computer played a card successfully, winning the game) • **ActionResult.ValidPlayAndExtraTurn** (the computer played a card successfully, but the user cannot play, so the computer has an extra turn) • **ActionResult.ValidPlay** (the computer played a card successfully, ending their turn)
Optional Methods	The methods here are optional. You should have at least some private methods in your class. Otherwise, your UserDrawCard, UserPlayCard, and ComputerAction methods will be far too long. The methods here are enough to complete the functionality of the game. Feel free to make changes to these methods or add in more methods as you see fit.	
	PlayCard(…)	Plays the chosen Card (given by cardNum) from the given Hand (given by hand), with a particular Suit (given by newSuit). Returns an ActionResult based on the normal rules of the game. This method could be used for both the player *and* the computer.
	DrawCard(…)	Attempts to draw a Card into the given Hand. Returns an ActionResult based on the normal rules of the game. This method could be used for both the player *and* the computer.
	IsHandPlayable(…)	Returns true if the given Hand contains a playable Card.
	IsCardPlayable(…)	Returns true if the given Card is currently playable.

GUI Functionality



```
Test Explorer
Run All | Run... | Playlist : All Tests
Card Games (77 tests) 26 failed
1_CardTests (6)                                13 ms
2_HandTests (13)                               19 ms
3_CardPileTests (11)                           15 ms
4_GamesTests (47)                             138 ms
  CrazyEightsTests (47)                       138 ms
    A_CrazyEightsBeforeGameStartsTests (6)     13 ms
    B_CrazyEightsAfterGameStartsTests (12)      9 ms
    C_CrazyEightsUserDrawCardTests (8)         89 ms
    D_CrazyEightsUserPlayCardTests (10)         9 ms
    E_CrazyEightsComputerDrawCardTests (5)     10 ms
    F_CrazyEightsComputerPlayCardTests (6)      6 ms
```

The **CrazyEights** logic class is designed to completely control the *game logic* and store the *game data*. The **GUI** is simply a way to view this data and to make requests to the **CrazyEights** class when needed. **For this reason, the GUI should NOT contain any private or public class variables.**

Once all the you have some of the **CrazyEights** UML complete, and you can successfully pass some of the unit tests such as **A_CrazyEightsBeforeGameStartsTests** and **B_CrazyEightsAfterGameStartsTests**, you can begin to implement and test GUI functionality to make the game interactive. For example, after completing those two sets of unit tests, you should be able to display the initial game board after pressing the *Deal* card button, which should start a game via `CrazyEights.StartGame()`. You can retrieve the players' hands from the GUI via `CrazyEights.UserHand` and `CrazyEights.ComputerHand`.

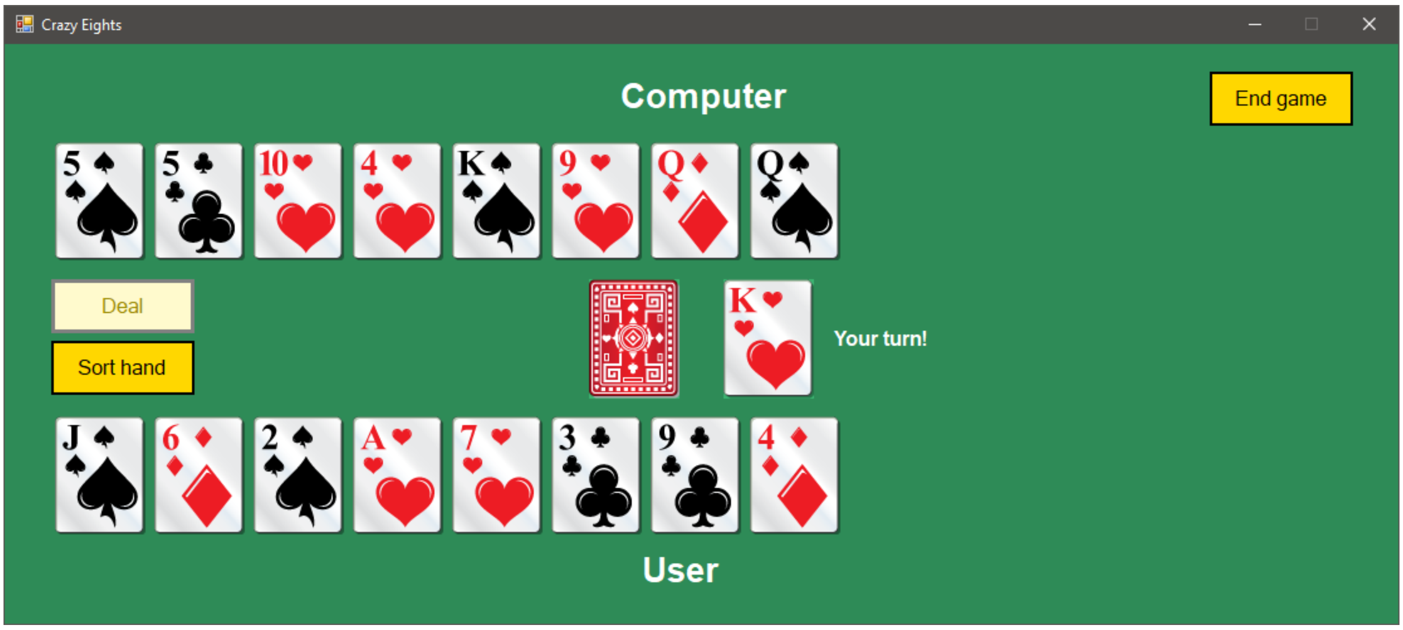


Figure 1: GUI after pressing 'Deal'

It is important that your GUI meets the following requirements:

- **Crazy_Eights_Form does not contain *any* class variables (private or public).**
- It is possible to play a complete game of Crazy Eights against the computer.
- It is not possible for the user to crash the program.
- It is not possible for the user to break the rules of the game.
- It is easy for the user to understand what is happening at all times. This means that the GUI updates slowly as events occur. For example, during the computer's turn, the computer may draw two cards and then play a card. The GUI should therefore update at least three times for this example, with a delay between each update so that the user can observe all three events. You can use the `UpdateInstructions` method from part B to achieve this, and to inform the user about the events that occur. It is up to you what the instruction label says between updates, but you may refer to the demonstration videos for ideas and examples. You will also need to use the `Refresh` method on `TableLayoutPanels` and `PictureBoxes` that you want to update during wait times.

Functionality Demonstration

See the Card Games assessment folder for demonstration videos.