Design Section

Prototype 1 - Main Page:

Overview:

In this section, I will plan the button-based navigation workflow to transition between different forms (pages) efficiently. The workflow will include determining user actions, creating form instances, and managing smooth transitions between pages. I will ensure proper navigation and data management where needed.

Decomposition to computable sections:

Section	Justification (suitable for computation because…)
Mapping Buttons to Open New Forms	Allows efficient navigation by using event handlers to create and display new form instances. This ensures user actions (button clicks) are mapped to appropriate pages while maintaining smooth transitions.
Implementing Navigation for Specific Pages	By explicitly assigning buttons to target pages (e.g., "Mock Test" opens InstructionsForm), the program behaves predictably. This eliminates errors, ensuring correct forms open in response to clicks.
Passing Data to Progress Page	Enables the Progress_Page to display relevant user data (e.g., test scores). Passing a dictionary from a global state ensures dynamic, real-time content is displayed, improving user experience.
Finalized Navigation Workflow	Provides a structured workflow for user actions, navigation logic, and data management. This ensures seamless button-based transitions while maintaining clean, functional code.

Step 1: Mapping Buttons to Open New Forms

To handle navigation between different pages, I will implement event handlers for each button. When a user clicks a button, the program will create an instance of the respective form, display it, and optionally hide the current form.

Pseudocode:

```
WHEN button is clicked:
    CREATE a new instance of the target form (e.g., PracticePage)
    DISPLAY the target form
    HIDE the current form (optional)
```

Reason:

Mapping buttons to specific actions allows the user to navigate through the program seamlessly. By creating a new form instance on a button click, I ensure that the program responds immediately to user input, improving interactivity and user experience. Hiding the current form (optional) prevents clutter on the screen and makes transitions smooth.

Approach:

- I will define click event handlers for all navigation buttons.
- Each event handler will create and show the respective form.
- This ensures that user actions are mapped to their intended targets.

Step 2: Implementing Navigation for Specific Pages

In this step, I will assign specific behavior to each button so that it opens the correct page. This will include buttons like "Mock Test," "Practice Page," and "Progress Page."

Pseudocode for Mock Test Page:

```
FUNCTION Mock_test_Click(event sender, event args)
    CREATE new instance of InstructionsForm called nextForm
    DISPLAY nextForm (Show it on the screen)
    HIDE the current form
END FUNCTION
```

Reason:
By explicitly mapping each button to its respective page (e.g., *Mock Test* button opens *InstructionsForm*), I ensure that the program behaves predictably. This is important for user experience as the correct pages open in response to user clicks. Using clear logic in event handlers eliminates errors, such as the wrong form opening due to incorrect references.

Approach:

- For the "Mock Test" button, I will ensure that it opens the *InstructionsForm* page to provide instructions before proceeding to the test.
- I will confirm that all button references are correct to avoid mismatches during navigation.

Step 3: Passing Data to Progress Page

To design the *Progress_Page*, I need to pass a dictionary that tracks test scores. This dictionary will store which tests have been attempted and their corresponding scores. I will retrieve the test scores from a global state or a centralized data class when creating the *Progress_Page* instance.

Pseudocode:

```
FUNCTION Progress_Click(event sender, event args)
    RETRIEVE testScores dictionary from GlobalData
    CREATE new instance of Progress_Page called nextForm, passing testScores as an
argument
    DISPLAY nextForm (Show it on the screen)
    HIDE the current form
END FUNCTION
```

Reason:
The *Progress_Page* needs access to user progress data to display attempted tests and scores. By retrieving the testScores dictionary from a global state and passing it to the *Progress_Page*, I ensure that the page reflects up-to-date and relevant information. This design also promotes separation of concerns, as data management is handled separately from UI navigation.

Approach:

- I will add logic to retrieve test scores from a global data store.
- I will pass this dictionary to the *Progress_Page* constructor when navigating to that form.
- This allows the *Progress_Page* to display scores dynamically and provide real-time feedback to the user.

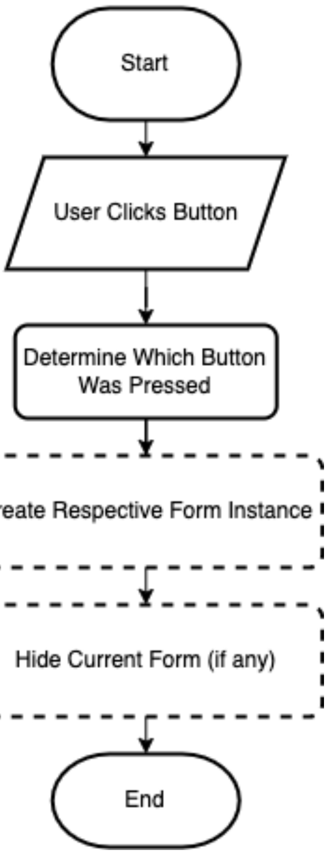
Final Workflow Summary

The finalized button navigation workflow will be implemented in three clear steps:

1. User Action: The program listens for button clicks.
2. Form Navigation: Based on the button clicked, I will create an instance of the corresponding form and display it.
3. Data Management: For specific pages (e.g., *Progress_Page*), I will pass necessary data to support dynamic content.

Flowchart Representation

The following flowchart summarizes the Button Click Workflow:


Start
User Clicks Button
Determine Which Button Was Pressed
Create Respective Form Instance
Hide Current Form (if any)
End

By carefully planning these steps, I will ensure smooth navigation between pages and efficient handling of user actions. Each form will serve its intended purpose, and any necessary data will be passed seamlessly. This design ensures that the workflow is clean, functional, and easy to maintain.

Processing Questions and User Answers

To process questions and user answers, I will implement a loop that displays each question along with its answers, takes user input, and saves the selected answer into a dictionary.

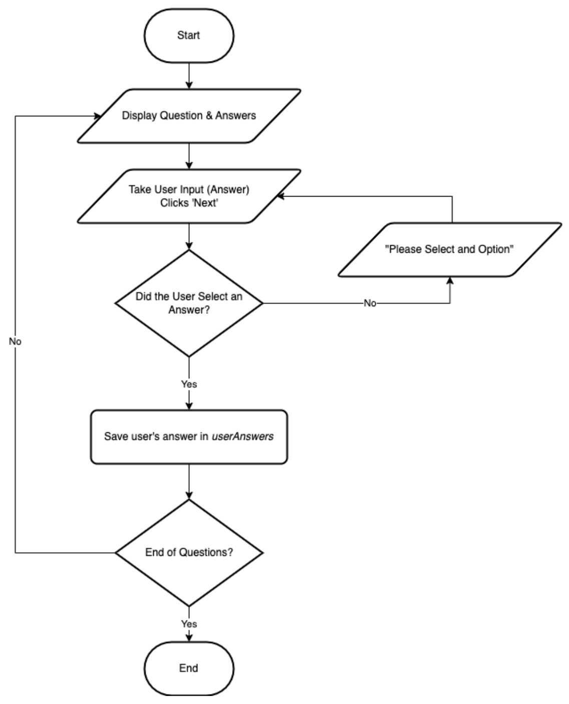
Pseudocode:

```
// Define a dictionary to store user answers
DEFINE Dictionary<int, string> userAnswers = NEW Dictionary<int, string>()

FUNCTION ProcessQuestions(questionsList)
    currentQuestionIndex = 0

    WHILE currentQuestionIndex < questionsList.Count
        DISPLAY questionsList[currentQuestionIndex].QuestionText
        DISPLAY questionsList[currentQuestionIndex].Answers AS Options

        userInput = GET UserInput()

        IF userInput IS NULL OR EMPTY
            CONTINUE // Go back to the same question
        ELSE
            userAnswers[currentQuestionIndex] = userInput
            currentQuestionIndex += 1 // Move to next question
        END IF
    END WHILE

    DISPLAY "All Questions Completed. Thank you!"
END FUNCTION
```

Explanation:

1. Dictionary Storage: Answers are stored in a dictionary where the key is the question index, and the value is the user's selected answer.
2. Display and Input: Questions and answers are displayed, and user input is taken dynamically.
3. Validation: If the user doesn't select an answer, the program remains on the current question.
4. End Condition: The loop terminates when all questions are processed, and a completion message is shown.

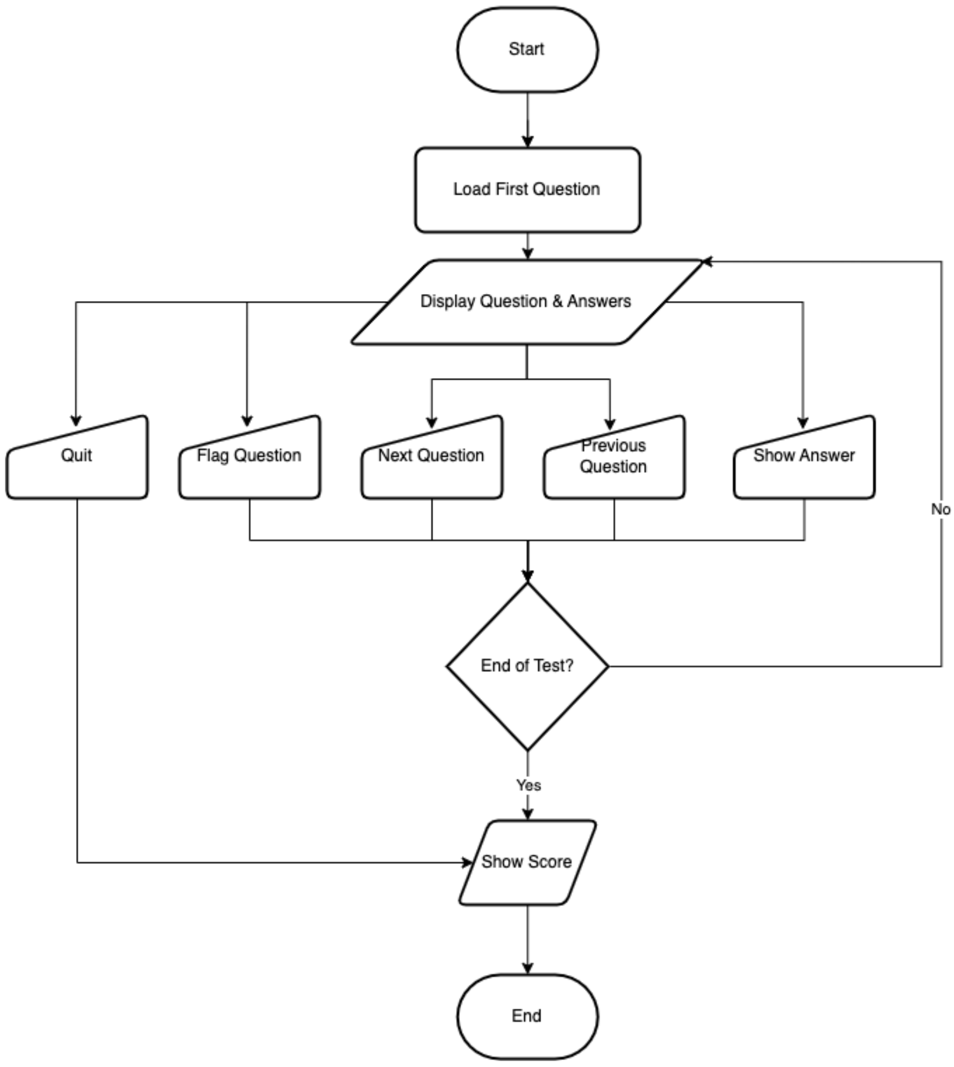
Flowchart Alignment:


Start
Display Question & Answers
Take User Input (Answer) Clicks 'Next'
"Please Select and Option"
Did the User Select an Answer?
No
No
Yes
Save user's answer in *userAnswers*
End of Questions?
Yes
End

Prototype 2 - Practice Page

Overview:
In this section, I will plan the workflow for the Practice Page, focusing on managing navigation, handling questions, and tracking user answers. The goal is to implement efficient transitions between questions, calculate the score, and display feedback based on user selections. The page will handle navigation through questions, display correct answers, and track test performance.

Flowchart Representation:



Decomposition into Computable Sections:

Section	Justification (suitable for computation because...)
Mapping Buttons to Navigate Between Questions	Allows users to navigate through questions using **Next** and **Previous** buttons. This ensures smooth navigation through the questions while tracking progress.
Tracking and Displaying Current Question	Keeps track of the current question index to ensure users always know which question they are on. This ensures a clean user interface and proper question management.
Updating the Score Dynamically	Dynamically calculates and updates the score based on the user's responses. This provides real-time feedback and improves the user experience.

Display Correct Answers After Completion	Ensures that users see the correct answers after completing the test. This gives valuable feedback to the user and helps reinforce learning.

Step 1: Mapping Buttons to Navigate Between Questions

To handle navigation through the questions (next and previous), I will implement Next and Previous buttons. When clicked, these buttons will update the currentQuestionIndex and navigate to the respective question.

Pseudocode:

```
WHEN NextButton_Click:
    IF currentQuestionIndex < selectedTest.Questions.Count - 1:
        Increment currentQuestionIndex
        LoadCurrentQuestion()

WHEN PreviousButton_Click:
    IF currentQuestionIndex > 0:
        Decrement currentQuestionIndex
        LoadCurrentQuestion()
```

Reason: Mapping buttons to navigation actions allows users to go through the test sequentially, ensuring proper flow. The buttons will help in iterating through the questions, preventing errors related to question ordering or skipping.

Approach:
- Implement click event handlers for the Next and Previous buttons.
- Each handler will update the question index and refresh the displayed question accordingly.

Step 2: Tracking and Displaying Current Question

The program needs to track the index of the current question to show the right question at all times. I will display the current question number and total questions (e.g., "Question 2 of 5").

Pseudocode:

```
FUNCTION UpdateQuestionDisplay:
    trackerLabel.Text = "Question [currentQuestionIndex + 1] of
[selectedTest.Questions.Count]"
```

Reason: Displaying the question number helps users understand their progress in the test, improving navigation and user awareness.

Approach:
- Update the question index whenever the user moves forward or backward in the test.
- Ensure the trackerLabel always reflects the current position in the test.

Step 3: Updating the Score Dynamically

The score will be updated after each question is answered. The user will receive immediate feedback about whether their answer was correct or incorrect.

Pseudocode:

```
FUNCTION UpdateScore(selectedOptionIndex):
    IF selectedOptionIndex == currentQuestion.CorrectOptionIndex:
        Increment score
    ELSE:
        No change
```

Reason: Updating the score dynamically provides real-time feedback to the user, helping them track their progress and understand which questions they answered correctly.

Approach:

- Compare the selected answer with the correct answer and adjust the score accordingly.
- Display the score update dynamically, either after each question or at the end of the test.

Step 4: Displaying Correct Answers After Completion

Once the test is completed, the program will display the correct answers for each question. This feedback will help users learn from their mistakes.

Pseudocode:

```
FUNCTION ShowCorrectAnswers:
    FOR EACH question IN selectedTest.Questions:
        IF selectedOptionIndex != question.CorrectOptionIndex:
            Highlight incorrect answers
        ELSE:
            Highlight correct answer
```

Reason: Displaying the correct answers after the test is completed helps users learn from their mistakes and reinforces the correct information.

Approach:

- After the test is completed, iterate over each question and compare the selected answer to the correct one.
- Highlight answers accordingly (e.g., correct answers in green, incorrect ones in red).

Final Workflow Summary

The finalized workflow for the Practice Page will consist of the following:

1. Navigation: The user navigates through the questions using the Next and Previous buttons, with the current question being displayed dynamically.
2. Score Calculation: The program updates the score as the user answers each question, providing real-time feedback.
3. Answer Display: Once the test is completed, the correct answers will be displayed to provide feedback to the user.

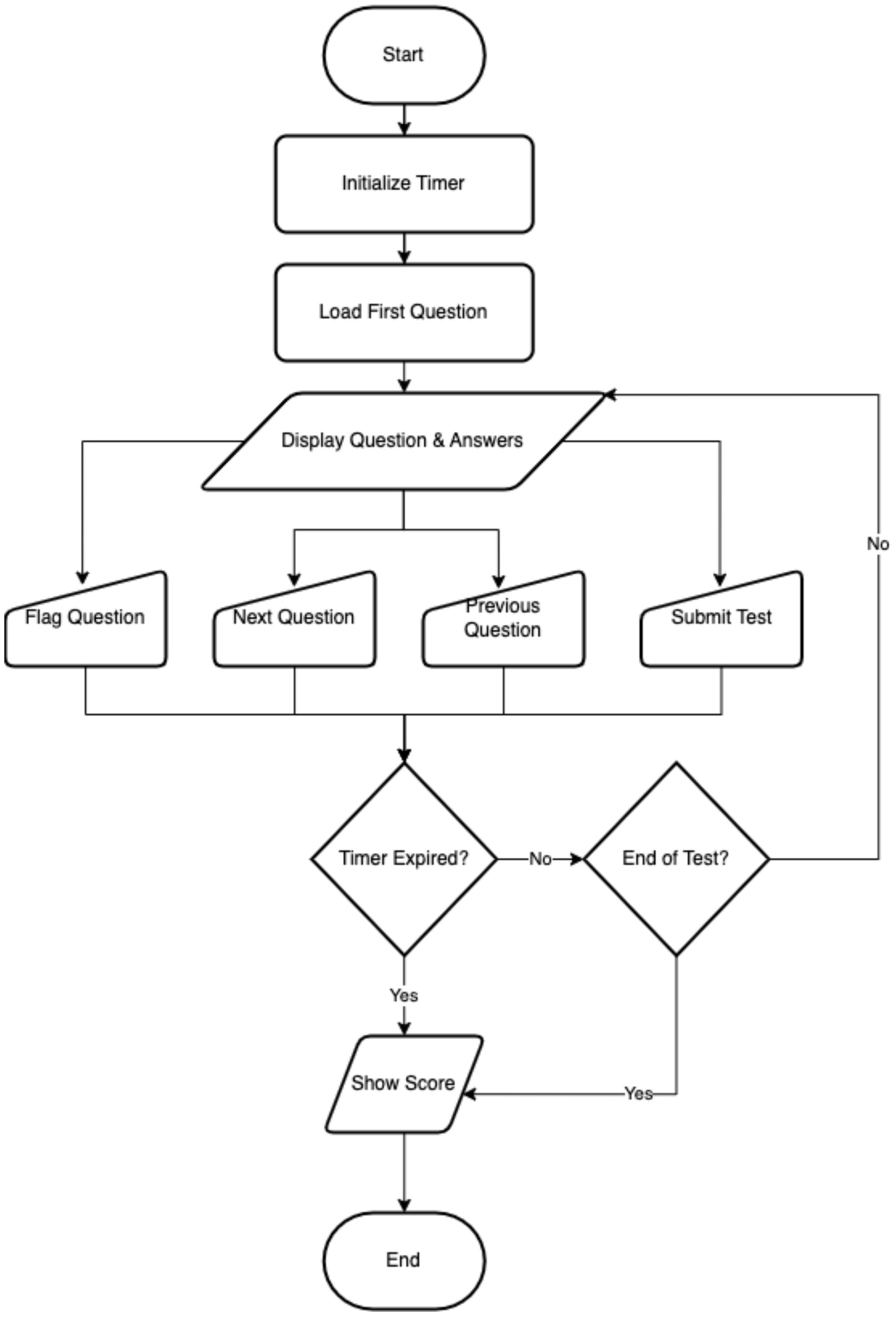
By carefully planning these steps, I will ensure that the Practice Page functions efficiently, providing a smooth user experience. The navigation, score tracking, and answer display mechanisms will help improve the test-taking experience and make the page dynamic and interactive.

Prototype 3 - Mock Test Page

Overview:

The Mock Test Page will have functionality similar to the practice page but with the added feature of a countdown timer. The timer will help manage the test duration, and users will be able to navigate through questions, flag important ones, track their answers, and review their performance at the end.

Flowchart Representation:



Decomposition into Computable Sections:

Section	Justification (suitable for computation because...)

Implementing Timer	The countdown timer ensures the test duration is tracked and test stops when time runs out.
Updating Timer	The timer needs to be updated every second to display the time remaining and stop the test when it hits zero.
Flagging Questions	Allows users to flag questions they want to review later and navigate through flagged questions.
Displaying User’s Choices and Results	After completing the test, it needs to display the user’s selections and correct answers.
Displaying Review of Incorrect Answers	Providing feedback by highlighting incorrect answers in red and correct answers in green.

Step 1: Adding a Timer Feature

Feature:
Implement a countdown timer to display the remaining time for the quiz.

Pseudocode:

```
WHEN InitializeTimer:
    Set timeRemaining = 30 * 60  // 30 minutes
    Start quizTimer every 1000 milliseconds (1 second)
    Display timeRemaining on timerLabel

WHEN quizTimer_Tick:
    IF timeRemaining > 0:
        Decrement timeRemaining by 1
        Update timerLabel with remaining time
    ELSE:
        Stop quizTimer
        Show final score
```

Reason:
A countdown timer helps keep the user aware of the time left, ensuring the quiz is completed within the time limit.

Approach:

- Create and initialize the timer on page load.
- Update the time every second and display it in a label.
- Stop the timer when the time is up and show the final score.

Step 2: Stopping the Quiz When Time Runs Out

Feature:
Stop the quiz automatically when the timer reaches zero and show the score.

Pseudocode:

```
WHEN quizTimer_Tick:
    IF timeRemaining == 0:
        Stop quizTimer
        Call ShowScore() to display the final score
```

Reason:
This ensures the quiz ends immediately when the timer expires, preventing any further answers or changes.

Approach:

- Monitor the timer during each tick and check for when time runs out.
- Call the ShowScore method to display the results once time is up.

Step 3: Fixing Timer Ambiguity Error

Feature:
Resolve the ambiguity error between System.Windows.Forms.Timer and System.Threading.Timer.

Pseudocode:

```
// Specify the correct Timer class
quizTimer = new System.Windows.Forms.Timer();
```

Reason:
The ambiguity error arises because both System.Windows.Forms.Timer and System.Threading.Timer are used in the project. Explicitly specifying the correct one resolves this conflict.

Approach:

- Ensure the timer class used is System.Windows.Forms.Timer.
- Modify the code wherever necessary to ensure there's no confusion between the two timer classes.

Step 4: Increasing Timer Duration

Feature:
Increase the quiz duration from 30 seconds to 57 minutes.

Pseudocode:

```
WHEN InitializeTimer:
    Set timeRemaining = 57 * 60  // 57 minutes
    Start quizTimer every 1000 milliseconds (1 second)
```

Reason:
A longer timer duration allows users more time to complete the quiz, especially for more complex questions.

Approach:

- Adjust the initial timeRemaining to 57 minutes.
- Update the timer accordingly and display the new countdown.

Step 5: Flagging Questions for Review

Feature:
Add the ability for users to flag questions to review later.

Pseudocode:

```
WHEN FlagButton_Click:
    IF currentQuestionIndex is not in flaggedQuestions:
        Add currentQuestionIndex to flaggedQuestions
    ELSE:
        Remove currentQuestionIndex from flaggedQuestions
```

Reason:
Flagging questions lets users mark ones they want to revisit later, improving their ability to manage difficult questions.

Approach:

- Create a list flaggedQuestions to store flagged question indices.
- Update the flag status whenever the "Flag" button is clicked.

Step 6: Skipping Flagged Questions

Feature:
Allow users to skip flagged questions without selecting an answer.

Pseudocode:

```
WHEN NextButton_Click:
    IF currentQuestionIndex is flagged:
        Increment currentQuestionIndex
        LoadNextQuestion
    ELSE IF selectedAnswer is valid:
        Save the answer and proceed
```

Reason:
This feature ensures users aren't forced to answer flagged questions immediately, giving them the freedom to skip and return later.

Approach:

- Check if the current question is flagged before navigating.
- Skip flagged questions if the user chooses to move forward without answering.

Step 7: Tracking User Answers

Feature:
Track and store user answers for review after completing the quiz.

Pseudocode:

```
WHEN NextButton_Click:
    Save selectedAnswer in userAnswers[currentQuestionIndex]
```

Reason:
Tracking answers allows users to review their responses at the end of the quiz, highlighting correct and incorrect choices.

Approach:

- Store each selected answer in a dictionary (userAnswers).
- Update the dictionary each time the user answers a question.

Step 8: Displaying Correct Answers After Completion

Feature:
At the end of the quiz, show the correct answers alongside the user's selections.

Pseudocode:

```
FUNCTION ShowCorrectAnswers:
    FOR EACH question IN selectedTest.Questions:
        IF userAnswers[questionIndex] != question.CorrectOptionIndex:
            Highlight the user's answer as incorrect
            Highlight the correct answer as correct
        ELSE:
            Highlight the user's answer as correct
```

Reason:
Displaying the correct answers after the quiz helps users learn from their mistakes and reinforces the correct answers.

Approach:

- Iterate over the answers and highlight them.
- Display feedback after the quiz is completed to show the correct answers.

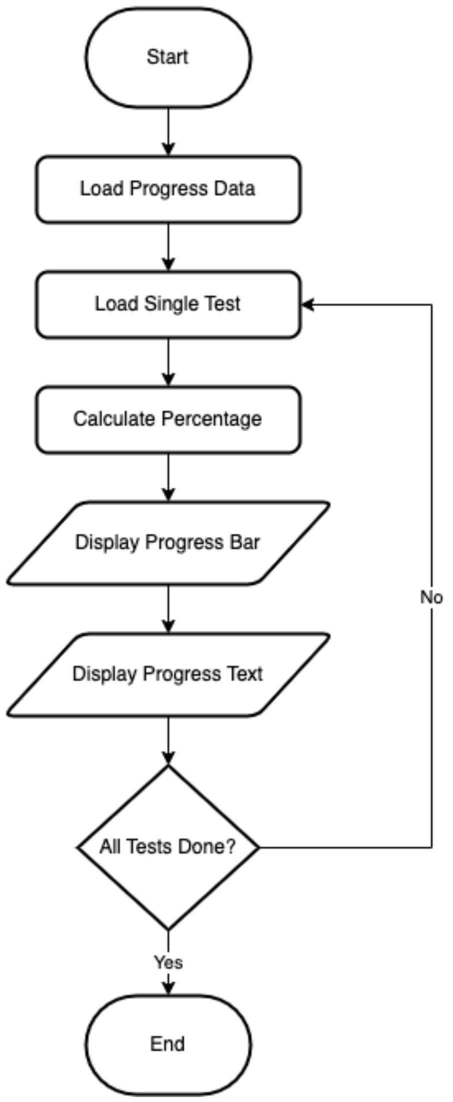
Prototype 4 - Progress Page:

Overview:

The Progress Page visually tracks a user's test performance using progress bars and percentages. It dynamically displays progress for both mock tests and practice tests. Key features include:

- Progress Bars: Represent scores as a percentage for each test.
- Dynamic Percentage Calculation: Adjusts for varying numbers of questions per test.
- Global Score Tracking: Integrates scores from both the Mock Test Page and Practice Page.
- Completed Topics Tracking: Shows completed topics with real-time updates using a timer.

Flowchart Representation:



Decomposition into Computable Sections:

Section	Justification (suitable for computation because...)
Displaying Progress for Practice Tests	Adds practice test scores to the global score dictionary and displays them alongside mock test progress.
Global Score Tracking	Stores and retrieves scores from a global dictionary for both mock tests and practice tests.
Label Identification for Test Types	Adds labels like "Mock Test Progress" and "Practice Test Progress" for clarity when displaying progress.

Completed Topics Tracking	Uses a **timer** to monitor and update the list of topics completed by the user in real-time.
Dynamic UI Updates	Dynamically creates progress bars, labels, and updates content based on the latest scores and topics data.

Step 1: Tracking User Progress and Scores

To track user progress (tests taken and scores), I will use a graphical progress bar and labels. The progress percentage will be calculated dynamically based on the number of questions in each test.

Pseudocode:

```
DEFINE a global dictionary: testScores
    KEY: test number (int)
    VALUE: test score (int)

DEFINE a global dictionary: testQuestions
    KEY: test number (int)
    VALUE: total number of questions (int)

FUNCTION SaveScore(testNumber, score)
    IF testScores does NOT contain key testNumber
        testScores[testNumber] = score
    END IF
END FUNCTION

FUNCTION CalculateScorePercentage(testNumber)
    IF testScores contains key testNumber AND testQuestions contains key testNumber
        RETURN (testScores[testNumber] * 100) / testQuestions[testNumber]
    ELSE
        RETURN 0
END FUNCTION
```

Reason:
Tracking progress requires storing scores for each test in a global dictionary. Dynamic calculation ensures the correct percentage even if tests have varying question counts.

Approach:

- Save user scores globally using dictionaries.
- Calculate the percentage score dynamically using the formula:
  (score / total_questions) * 100.

Step 2: Displaying Progress Bars and Labels

I will display a progress bar for each test, showing the user's score as a percentage. A label next to the progress bar will display the exact percentage or indicate if the test has not been attempted.

Pseudocode:

```
FUNCTION DisplayProgressBarWithPercentage(testNumber, yPosition)
    percentage = CalculateScorePercentage(testNumber)

    ProgressBar progressBar = NEW ProgressBar
        progressBar.Minimum = 0
        progressBar.Maximum = 100
        progressBar.Value = percentage
        progressBar.Size = NEW Size(200, 20)
        progressBar.Location = NEW Point(100, yPosition)

    Label percentageLabel = NEW Label
        percentageLabel.Text = (percentage > 0) ? $"{percentage}%" : "Not Attempted"
        percentageLabel.AutoSize = TRUE
        percentageLabel.Location = NEW Point(310, yPosition)

    Controls.Add(progressBar)
    Controls.Add(percentageLabel)
END FUNCTION
```

Reason:
Progress bars give a visual representation of progress, while labels provide clarity with exact percentages.

Approach:

- Use the ProgressBar control to display scores visually.
- Dynamically calculate percentages and update labels alongside the bars.

Step 3: Integrating Mock and Practice Test Scores

I will extend the progress display to include both Mock and Practice tests by introducing separate dictionaries.

Pseudocode:

```
CLASS GlobalData
    STATIC Dictionary<int, int> MockTestScores = NEW Dictionary<int, int>()
    STATIC Dictionary<int, int> PracticeTestScores = NEW Dictionary<int, int>()

FUNCTION ShowScore(testType, testNumber, score)
    IF testType == "Mock"
        GlobalData.MockTestScores[testNumber] = score
    ELSE IF testType == "Practice"
        GlobalData.PracticeTestScores[testNumber] = score
    END IF
END FUNCTION

FUNCTION DisplayTestProgress()
    yPosition = 50

    // Mock Test Progress
    CREATE Label mockProgressLabel
    mockProgressLabel.Text = "Mock Test Progress"
    Controls.Add(mockProgressLabel)

    FOR EACH testNumber IN testQuestions.Keys
        CALL DisplayProgressBarWithPercentage(testNumber, yPosition,
GlobalData.MockTestScores)
        yPosition += 30
    END FOR

    // Practice Test Progress
    CREATE Label practiceProgressLabel
    practiceProgressLabel.Text = "Practice Test Progress"
    Controls.Add(practiceProgressLabel)
    FOR EACH testNumber IN testQuestions.Keys
        CALL DisplayProgressBarWithPercentage(testNumber, yPosition,
GlobalData.PracticeTestScores)
        yPosition += 30
    END FOR
END FUNCTION
```

Reason:
Tracking both Mock and Practice tests separately ensures clarity and flexibility for progress evaluation.

Approach:

- Maintain separate dictionaries for Mock and Practice test scores.
- Display progress under distinct sections for better organization.

Step 4: Handling Topics Completed with a Timer

To track completed topics, I will add a timer that checks for user updates every second and dynamically displays the completed topics.

Pseudocode:

```
FUNCTION InitializeTopicsCompleted()
    CREATE Label topicsCompletedLabel
        topicsCompletedLabel.Text = "Topics Completed: None"
        topicsCompletedLabel.Font = Arial, 12, Bold
        topicsCompletedLabel.AutoSize = True
        topicsCompletedLabel.Location = Point(20, yPosition)
    Controls.Add(topicsCompletedLabel)

    CREATE Timer updateTimer
        updateTimer.Interval = 1000
        updateTimer.Tick += (s, e) => CALL UpdateTopicsCompleted()
        updateTimer.Start()
END FUNCTION

FUNCTION UpdateTopicsCompleted()
    completedTopics = GET Completed Topics Count()
    IF completedTopics > 0
        topicsCompletedLabel.Text = $"Topics Completed: {completedTopics}"
    ELSE
        topicsCompletedLabel.Text = "Topics Completed: None"
    END IF
END FUNCTION
```

Reason:
Using a timer ensures that updates are fetched and displayed dynamically without requiring user intervention.

Approach:

- Create a Timer that checks for updates every second.
- Update the topics label dynamically based on the completed topics count.

Step 5: Final Workflow Summary

The finalized workflow for the Progress Page will consist of the following:

1. Tracking Progress: Save scores and dynamically calculate percentages for each test.
2. Displaying Progress Bars: Show progress bars and labels for Mock and Practice tests separately.
3. Integrating Practice Scores: Use dictionaries to track Mock and Practice test scores globally.
4. Dynamic Updates: Add a timer to track completed topics and update the display in real-time.

Key Features Implemented:

- Graphical progress bars for visual progress tracking.
- Dynamic percentage calculation based on total test questions.
- Separate tracking for Mock and Practice tests.
- Real-time topic completion updates using a timer.

By implementing these steps, the Progress Page will provide a clear, dynamic, and user-friendly experience for tracking test progress and completed topics.

Prototype 5 - Flagged Questions Page:

Overview:
The Flagged Questions Page is designed to display all the questions that a user has flagged during a test. It extracts flagged question indices, retrieves the corresponding question details, and displays the question text and the correct answer in a clean, scrollable interface.

Decomposition into Computable Sections:

Section	Justification (suitable for computation because...)
Retrieving Flagged Questions	Loops through all flagged question indices to fetch the respective question details.
Accessing Test Data	Retrieves the flagged questions using the GlobalData.AllTests dictionary to ensure all tests are accessible.
Displaying Flagged Questions	Dynamically creates labels for each question, displaying both the question text and the correct answer.
Scrollable UI Creation	Adds the dynamically generated question labels to a scrollable panel for an organized display.
Dynamic Positioning of Labels	Adjusts the yPosition to space out the displayed questions properly within the scrollable panel.

Steps to Implement the Flagged Questions Page:

1. Setup Data Structure
    - Ensure flagged question indices are stored in a suitable structure, such as a dictionary where the key is the test number and the value is a list of flagged question indices.

```
Dictionary<int, List<int>> flaggedQuestions = new Dictionary<int,
List<int>>();
```

2. Accessing Flagged Questions
    - Use the test number to access flagged question indices.
    - Retrieve the corresponding questions and their correct answers from GlobalData.AllTests.

```
question = GlobalData.AllTests[testNumber - 1][flaggedIndex];
```

3. Dynamic UI Creation
    - Loop through the flagged questions and dynamically create labels for each question and its correct answer.
    - Add these labels to a scrollable panel.
4. Positioning and Styling
    - Use the yPosition variable to space out labels properly.
    - Apply styling like font, size, and location for better readability.

```
Label questionLabel = new Label();
questionLabel.Text = "Q: " + question.Text + "\nA: " +
question.Options[question.CorrectOptionIndex];
questionLabel.Font = new Font("Arial", 10, FontStyle.Regular);
questionLabel.AutoSize = true;
questionLabel.Location = new Point(40, yPosition);
scrollablePanel.Controls.Add(questionLabel);
```

```
        yPosition += 50;
```

5. Scrollable UI Integration
   - Place all dynamically generated content within a scrollable panel to handle cases where there are multiple flagged questions.

Final Steps and Functionality:

1. Store Flagged Indices: Ensure flagged questions are tracked during the test.
2. Pass Flagged Questions: Provide flagged indices when navigating to the Flagged Questions Page.
3. Retrieve Data: Fetch flagged questions from GlobalData.AllTests.
4. Display Data: Dynamically create and position labels to display question text and correct answers.
5. Scrollable UI: Ensure all questions can be viewed within a scrollable interface.

Prototype 6 - Traffic Signs Page:

Overview:
The Traffic Signs Page allows users to mark topics as completed, track their progress, and study traffic signs with a combination of text and images. This page implements checkboxes for progress tracking, persistent state storage, navigation to topic-specific pages, and a grid to display images with descriptions.

Decomposition into Computable Section

Section	Justification (suitable for computation because...)
Restoring Checkbox States	Ensures that previously checked topics remain checked when reopening the Traffic Signs Page.
Updating Checkbox States	Tracks changes to checkboxes, updates the state in a persistent storage, and maintains completed topics.
Navigating to Topic Pages	Allows users to click on a topic and view all related traffic signs with images and text.
Implementing DataGrid for Images/Text	Displays traffic signs with their descriptions in a structured format using a scrollable DataGridView.
Dynamically Adding Rows	Ensures that new signs with images and text can be dynamically added to the grid

---

Step 1: Persistent Checkbox States

To persist checkbox states across page reloads, I will use a static dictionary. This dictionary will hold the state (checked/unchecked) for each traffic sign category.

Pseudocode:

```
DEFINE Static Dictionary CheckboxStates AS DICTIONARY
{
    "Giving Orders": FALSE,
    "Warning Signs": FALSE,
    "Direction Signs": FALSE,
    "Information Signs": FALSE,
    "Road Work Signs": FALSE
}
```

Reason:
Storing checkbox states in a static dictionary allows for persistence during navigation and ensures the states remain consistent.

Approach:

- Use a static dictionary to hold the state of each checkbox.
- The dictionary allows easy updates and retrieval of checkbox states.

Step 2: Restoring Checkbox States

On page load, checkbox states will be restored using the values from the dictionary.

Pseudocode:

```
FUNCTION RestoreCheckboxStates()
    SET Giving_Order_Complete.Checked = CheckboxStates["Giving Orders"]
    SET Warning_Signs_Complete.Checked = CheckboxStates["Warning Signs"]
    SET Direction_Signs_Complete.Checked = CheckboxStates["Direction Signs"]
    SET Information_Signs_Complete.Checked = CheckboxStates["Information Signs"]
    SET Road_Work_Complete.Checked = CheckboxStates["Road Work Signs"]
END FUNCTION
```

Reason:
Restoring the state of checkboxes ensures a seamless user experience, so users don't lose progress after navigating or reloading the page.

Approach:

- Retrieve values from the static dictionary.
- Set the corresponding checkbox state dynamically.

Step 3: Updating Checkbox States and Completed Topics

Checkbox interactions will dynamically update both the dictionary and a list of completed topics.

Pseudocode:

```
FUNCTION UpdateCompletedTopics(topic, isCompleted)
    IF isCompleted IS TRUE
        IF topic IS NOT IN CompletedTopics
            ADD topic TO CompletedTopics
        END IF
    ELSE
        REMOVE topic FROM CompletedTopics
    END IF

    SET CheckboxStates[topic] = isCompleted
END FUNCTION
```

Reason:
Tracking completed topics alongside updating checkbox states ensures data consistency and enables progress tracking.

Approach:

- Update the CheckboxStates dictionary when a checkbox is clicked.
- Maintain a list of completed topics for progress tracking.

Step 4: Navigation to Topic-Specific Pages

Clicking on a checkbox or button will navigate users to a detailed page for the corresponding topic.

Pseudocode:

```
FUNCTION Orders_Signs_Click(sender, e)
    CREATE nextForm AS new Orders_Signs()
    CALL nextForm.Show()
    CALL this.Hide()
END FUNCTION
```

Reason:
Navigating to specific pages allows users to dive deeper into detailed information about each traffic sign type.

Approach:

- Attach click events to each topic.
- Create and show the corresponding form while hiding the current page.

Step 5: Displaying Traffic Signs in a DataGridView

Traffic signs and descriptions will be displayed dynamically using a DataGridView.

Pseudocode:
Initialize Grid:

```
FUNCTION InitializeGrid()
    CREATE signsGridView AS new DataGridView()
    SET signsGridView.Dock TO DockStyle.Fill
    SET signsGridView.AutoSizeColumnsMode TO DataGridViewAutoSizeColumnsMode.Fill
    SET signsGridView.RowTemplate.Height TO 100
    SET signsGridView.ReadOnly TO true

    CREATE imageColumn AS new DataGridViewImageColumn()
    SET imageColumn.HeaderText TO "Sign Image"
    SET imageColumn.ImageLayout TO DataGridViewImageCellLayout.Zoom
    ADD imageColumn TO signsGridView.Columns

    CREATE infoColumn AS new DataGridViewTextBoxColumn()
    SET infoColumn.HeaderText TO "Information"
    ADD infoColumn TO signsGridView.Columns

    CALL AddSignRow(signsGridView, "Signs with red circles are mostly prohibitive.",
"Blank.png")
    CALL AddSignRow(signsGridView, "Entry to 20 mph zone", "Entry_to_20_mph_zone.png")
    CALL AddSignRow(signsGridView, "End of 20 mph zone", "End_of_20_mph_zone.png")

    ADD signsGridView TO Controls
END FUNCTION
```

Adding Rows:

```
FUNCTION AddSignRow(grid, info, imagePath)
    CREATE appDirectory AS AppDomain.CurrentDomain.BaseDirectory
    CREATE imageFullPath AS System.IO.Path.Combine(appDirectory, "Signs_Giving_Order",
imagePath)

    IF NOT System.IO.File.Exists(imageFullPath) THEN
        THROW new System.IO.FileNotFoundException("Image file not found: " +
imageFullPath)
    END IF

    CREATE signImage AS Image.FromFile(imageFullPath)
    CALL grid.Rows.Add(signImage, info)
END FUNCTION
```

Reason:
A DataGridView provides a structured and visually appealing way to display traffic signs alongside their descriptions.

Approach:

- Use DataGridView with image and text columns.
- Dynamically add rows for each traffic sign using images stored locally.

Step 6: Debugging for Validation

Print statements will validate checkbox states while restoring and updating the dictionary.

Pseudocode:

```
PRINT "Checkbox States Restored:"
FOR EACH entry IN CheckboxStates
    PRINT entry.Key + ": " + entry.Value
END FOR

PRINT "Updated Checkbox State: " + topic + " = " + isCompleted
```

Reason:
Debugging with print statements ensures that the checkbox states and completed topics are being updated correctly, which is crucial for tracking user progress.

Approach:

- Print the current state of the dictionary during restoration and updates.
- Validate the consistency of checkbox states and the completed topics list.

Step 7: Final Workflow Summary

The finalized workflow for the Traffic Signs Page will include:

1. Tracking Checkbox States: Use a static dictionary to persist checkbox states across page reloads.
2. Restoring States: Retrieve and set checkbox states dynamically on page load.
3. Dynamic Updates: Update the dictionary and completed topics list as checkboxes are interacted with.
4. Navigation: Enable navigation to detailed pages for each traffic sign topic.
5. Displaying Signs: Use a DataGridView to show traffic signs and descriptions.
6. Validation: Add debugging statements to ensure consistency and correctness during state restoration and updates.

END OF DOC

Prototype 7 - Settings Page:

Iteration 1:

Update: Setting button to change the background colour of each form for user visibility

Create a new Form Page of Settings

Psuedocode:

```
FUNCTION Change_Color_Click(sender, e)
   CREATE colorDialog AS new ColorDialog()
   IF colorDialog.ShowDialog() == DialogResult.OK THEN
      SET GlobalBackgroundColor TO colorDialog.Color
      CALL MessageBox.Show("Background color updated. It will apply to all forms when they are
reopened.",
                     "Settings",
                     MessageBoxButtons.OK,
                     MessageBoxIcon.Information)
   END IF
END FUNCTION
```

Iteration 2:

Problem: How can the user change the size of the font?
Solution: Introduce a slider on the Settings Page So That The User can change the size of the font as well as choose if they want to make it bold/italic

Psuedocode:

```
FUNCTION FontSizeSlider_Scroll(sender, e)
   SET GlobalFontSize TO fontSizes[fontSizeSlider.Value]
   SET fontPreviewLabel.Font TO new Font("Arial", GlobalFontSize, GlobalFontStyle)
END FUNCTION
FUNCTION FontStyleCheckBox_CheckedChanged(sender, e)
   SET fontStyle TO FontStyle.Regular
   IF boldCheckBox.Checked THEN
      fontStyle |= FontStyle.Bold
   END IF
   IF italicCheckBox.Checked THEN
      fontStyle |= FontStyle.Italic
   END IF
   SET GlobalFontStyle TO fontStyle
```

```
    SET fontPreviewLabel.Font TO new Font("Arial", GlobalFontSize, GlobalFontStyle)
    CALL MessageBox.Show("Font style updated to " + GlobalFontStyle, "Settings",
MessageBoxButtons.OK, MessageBoxIcon.Information)
END FUNCTION
```