**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.

**What I will do:**

* 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.

**What I will do:**

* 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.

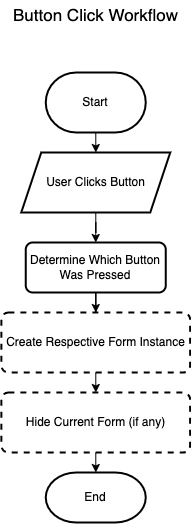
**Final Workflow Summary**

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.

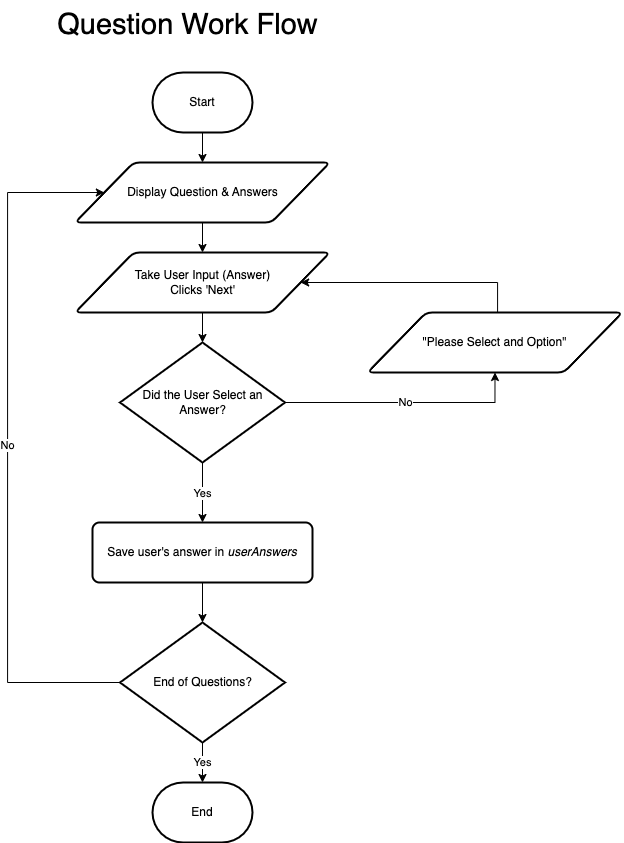
**Flowchart Representation**

The following flowchart summarizes the Button Click Workflow:

****

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.

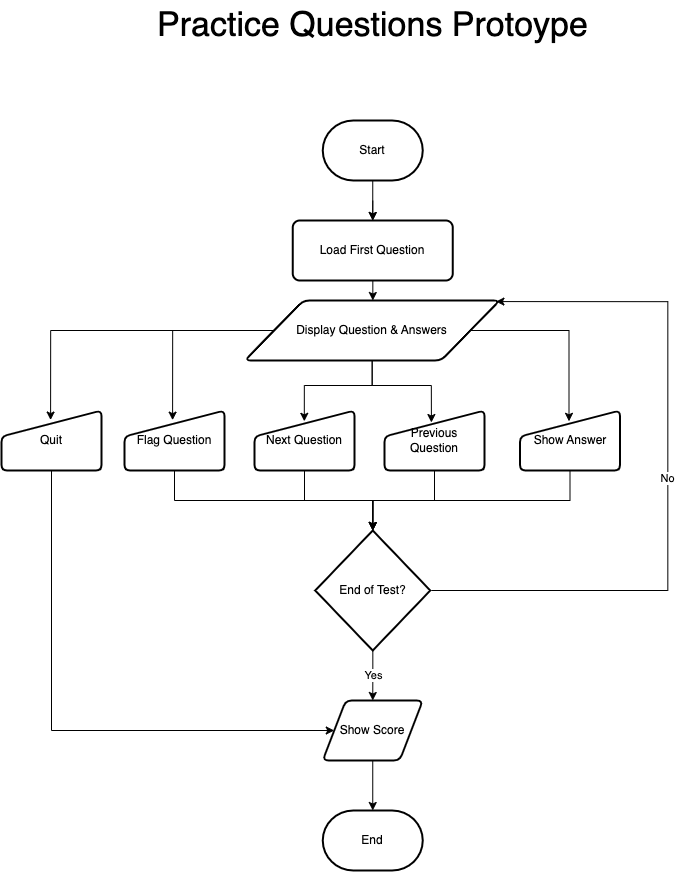
**How Questions and Answers will be processed?**

****

**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:

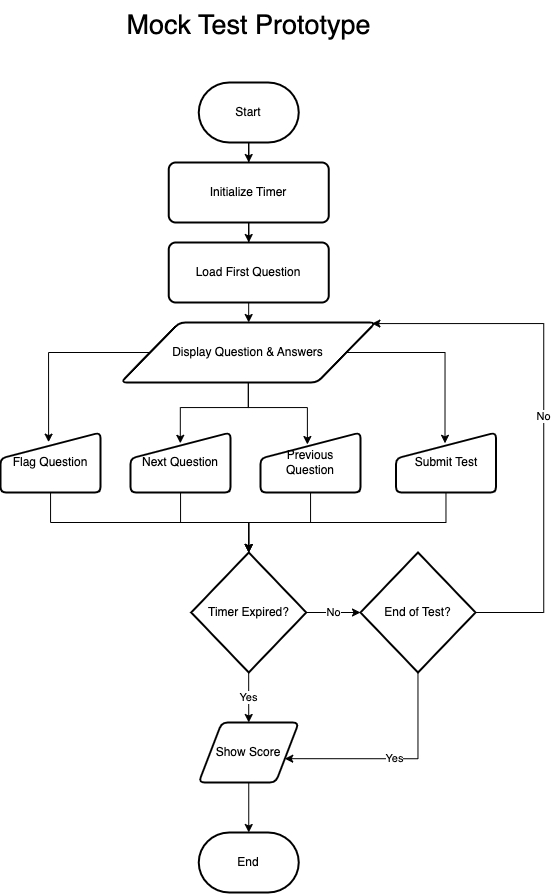
1. Navigation: The user navigates through the questions using the Next and Previous buttons, with the current question being displayed dynamically.
2. Score Calculation: The program updates the score as the user answers each question, providing real-time feedback.
3. Answer Display: Once the test is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

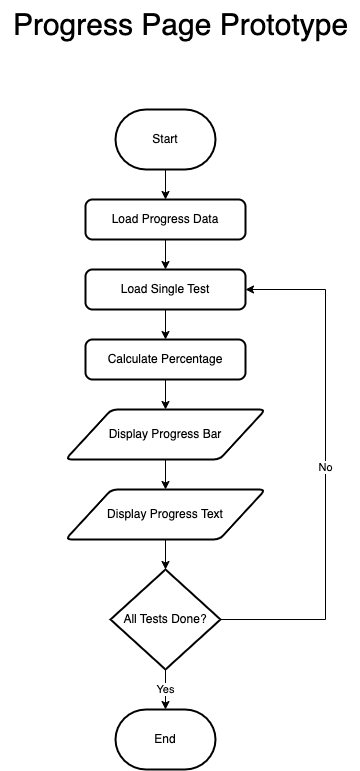
Highlight the user's answer as correct

**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:**

****

**Iteration 1:**

**Problem:** How can I track the progress of the user, including how many tests they have taken and their scores on those tests (graphically)? How will the score percentage be calculated?

**Solution:** To implement a graphical progress bar for the progress page, I can use the ProgressBar control in Windows Forms and a Label to display the score percentage. I can create a dynamic progress bar for each test, where the value of the progress bar reflects the user's score as a percentage. I will calculate the score percentage based on the number of correct answers (e.g., 10 questions per test, or adjust this to the actual number of questions in the test).

Next to each progress bar, a Score Display label will show the score percentage or indicate if the test has not been attempted. When navigating to the Progress\_Page, I will pass a dictionary containing the scores for each test to ensure all progress is displayed correctly. **Psuedocode:**

***DECLARE a Dictionary called userAnswers***

***KEY: question index (int)***

***VALUE: selected answer (int)***

***FUNCTION Next\_Click(event sender, event args)***

***// Check if the user has selected an answer***

***IF user has selected an option for the current question***

***// Store the user's answer in the userAnswers dictionary***

***userAnswers[currentQuestionIndex] = selectedOption***

***END IF***

***// Proceed to the next question logic (as usual)***

***INCREMENT currentQuestionIndex***

***CALL LoadQuestion() to load the next question***

***END FUNCTION***

***FUNCTION ShowScore()***

***// Create a label for reviewing incorrect answers***

***CREATE a Label called reviewLabel***

***SET text to "Review of Incorrect Questions:"***

***SET font to Arial, size 10, bold***

***ADD reviewLabel to the scrollable panel***

***// Set the starting position for displaying questions***

***SET yPosition to the bottom of reviewLabel + 10***

***// Loop through all the questions***

***FOR each question in selectedTest***

***DECLARE userAnswer as -1 // Default value for no answer***

***DECLARE isCorrect as false***

***// Check if the user has answered the question***

***IF userAnswers contains the current question index***

***SET userAnswer to userAnswers[currentQuestionIndex]***

***SET isCorrect to (userAnswer == question.CorrectOptionIndex)***

***END IF***

***// Skip displaying the correct answers***

***IF isCorrect***

***CONTINUE to the next question***

***END IF***

***// Display the question text***

***CREATE a Label for the question***

***SET text to "Q{index + 1}: {question.Text}"***

***ADD question label to the scrollable panel***

***// Display the options***

***FOR each option in question.Options***

***CREATE a Label for the option***

***SET text to the option text***

***// Highlight user's incorrect answer in red***

***IF option index matches userAnswer***

***SET option label text color to Red***

***END IF***

***// Highlight the correct answer in green***

***IF option index matches question.CorrectOptionIndex***

***SET option label text color to Green***

***END IF***

***ADD option label to the scrollable panel***

***UPDATE yPosition for next option***

***END FOR***

***// Add spacing between questions***

***UPDATE yPosition for spacing between questions***

***END FOR***

***END FUNCTION***

**Problem:** How to Integrate it with the Mock test page?  
**Solution:** To integrate with my MockTest\_Page, updating the ShowScore method so it can save the score in a global object. Then, pass the scores to the Progress\_Page.  
  
**Psuedocode:**  
***DEFINE a global dictionary: testScores***

***testScores = NEW Dictionary<int, int>()***  
***FUNCTION SaveScore(testNumber, score)***

***IF testScores does NOT contain key testNumber***

***testScores[testNumber] = score***

***END IF***

***END FUNCTION***

***FUNCTION ShowScore(testNumber, score)***

***SaveScore(testNumber, score)***

***DISPLAY score to the user***

***END FUNCTION***

***FUNCTION NavigateToProgressPage()***

***Progress\_Page nextForm = NEW Progress\_Page(testScores)***

***nextForm.Show()***

***CURRENT\_PAGE.Hide()***

***END FUNCTION***

**Iteration 2:**

**Update**: To Dynamically calculate the progress bar based on the number of questions in the test  
  
**Psuedocode:**

***DEFINE a global dictionary: testQuestions***

***testQuestions = NEW Dictionary<int, int>()***

***testQuestions[1] = 10 // Test 1 has 10 questions***

***testQuestions[2] = 8 // Test 2 has 8 questions***

***testQuestions[3] = 12 // Test 3 has 12 questions***

***DEFINE a global dictionary: testScores***

***testScores = NEW Dictionary<int, int>()***

***FUNCTION DisplayProgressBar(testNumber, yPosition)***

***IF testScores contains key testNumber AND testQuestions contains key testNumber***

***scorePercentage = (testScores[testNumber] \* 100) / testQuestions[testNumber]***

***ELSE***

***scorePercentage = 0***

***END IF***

***ProgressBar progressBar = NEW ProgressBar***

***progressBar.Minimum = 0***

***progressBar.Maximum = 100***

***progressBar.Value = scorePercentage***

***progressBar.Size = NEW Size(200, 20)***

***progressBar.Location = NEW Point(100, yPosition)***

***Controls.Add(progressBar)***

***END FUNCTION***  
  
**Problem:** The text next to the progress bar is 10%/20% it is being calculated by Multiplying the score by 10,Assuming that the number of question is 10  
the percentage should be calculated with the dynamic question logic now  
 ***Text = testScores.ContainsKey(testNumber)***

***? $"{testScores[testNumber] \* 10}%"***

***: "Not Attempted",***

**Solution:** I can use a Label that dynamically calculates and displays the percentage based on the testScores and testQuestions.   
Calculate Percentage percentage is dynamically calculated now using the formula

**FUNCTION DisplayProgressBarWithPercentage(testNumber, yPosition)**

**IF testScores contains key testNumber AND testQuestions contains key testNumber**

**percentage = (testScores[testNumber] \* 100) / testQuestions[testNumber]**

**ELSE**

**percentage = 0**

**END IF**

**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 = testScores.ContainsKey(testNumber)**

**? $"{percentage}%"**

**: "Not Attempted"**

**percentageLabel.AutoSize = TRUE**

**percentageLabel.Location = NEW Point(310, yPosition) // Adjust X coordinate to position the label next to the progress bar**

**Controls.Add(progressBar)**

**Controls.Add(percentageLabel)**

**END FUNCTION**

**Iteration 3:**

**Update:** It only shows the progress of the mock test right now I want to change so that so it also shows the progress of practice page  
I have to add A variable to score practice score globally

**Psuedocode:**  
***CLASS GlobalData***

***STATIC Dictionary<int, int> TestScores = NEW Dictionary<int, int>()***

***// Key: Test number (integer)***

***// Value: Test score (integer)***

***STATIC Dictionary<int, int> PracticeScores = NEW Dictionary<int, int>()***

***// Key: Practice test number (integer)***

***// Value: Practice test score (integer)***

***END CLASS***I then have to store the scores aswell **Psuedocode:**

***FUNCTION ShowScore()***

***CALL Controls.Clear()***

***IF selectedTest IS NOT NULL THEN***

***SET testIndex = INDEX OF selectedTest IN tests + 1***

***GlobalData.PracticeScores[testIndex] = score***

***END IF***

***END FUNCTION***

I also changed the label to better identify the tests and show progress under  
**Psuedocode:**

***FUNCTION DisplayTestProgress()***

***CREATE Label mockProgressLabel***

***SET mockProgressLabel.Text TO "Mock Test Progress"***

***ADD mockProgressLabel TO form controls***

***FOR EACH testNumber IN testQuestions.Keys***

***CALL AddProgressRow("Test", testNumber, mockTestScores, REF yPosition)***

***END FOR***

***CREATE Label practiceProgressLabel***

***SET practiceProgressLabel.Text TO "Practice Test Progress"***

***ADD practiceProgressLabel TO form controls***

***FOR EACH testNumber IN testQuestions.Keys***

***CALL AddProgressRow("Test", testNumber, practiceTestScores, REF yPosition)***

***END FOR***

***END FUNCTION***

**Iteration 5:**

**Problem:** After adding the checkbox for completed topics, how can I track what the user has completed? And if they go back and complete it later, how will that be updated?  
**Solution**: I will introduce a timer that checks what the user has selected to be completed. This timer will run every second. If the user hasn't completed any topics, it will display "None."  
add a timer and the topics completed  
  
**Psuedocode:**

***FUNCTION InitializeTopicsCompleted()***

***CREATE Label topicsCompletedLabel***

***SET topicsCompletedLabel.Text TO "Topics Completed: None"***

***SET topicsCompletedLabel.Font TO Arial, 12, Bold***

***SET topicsCompletedLabel.AutoSize TO True***

***SET topicsCompletedLabel.Location TO Point(20, yPosition) // Position based on yPosition***

***ADD topicsCompletedLabel TO form controls***

***CREATE Timer updateTimer***

***SET updateTimer.Interval TO 1000 // Update every 1 second***

***updateTimer.Tick += (s, e) => CALL UpdateTopicsCompleted()***

***updateTimer.Start()***

***END FUNCTION***

**Prototype 5 - Flagged Questions Page:  
Iteration 1:**  
For getting all the flagged questions and showing them on one page

**Psuedocode: *FUNCTION DisplayFlaggedQuestions()***

***FOR each questionIndex IN test.Value***

***SET question TO GlobalData.AllTests[test.Key - 1][questionIndex]***

***CREATE Label questionLabel***

***SET questionLabel.Text TO "Q: " + question.Text + "\nA: " + question.Options[question.CorrectOptionIndex]***

***SET questionLabel.Font TO Arial, 10, Regular***

***SET questionLabel.AutoSize TO True***

***SET questionLabel.Location TO Point(40, yPosition)***

***ADD questionLabel TO scrollablePanel.Controls***

***yPosition = yPosition + 50***

***END FOR***

***END FUNCTION***

**Prototype 6 - Traffic Signs Page:**

**Iteration 1:**  
**Problem:** How to know what topics are completed by the user?  
**Solution:** I will add a checkbox under every topic so that when the user completes a topic, they can check it. This allows the user to track their progress and mark topics as completed.

**Psuedocode:**  
***DEFINE Global List CompletedTopics AS EMPTY LIST***

***FUNCTION Traffic\_Signs\_page()***

***CALL InitializeComponent()***

***END FUNCTION***

***FUNCTION Giving\_Order\_Complete\_CheckedChanged(sender, e)***

***CALL UpdateCompletedTopics("Giving Orders", Giving\_Order\_Complete.Checked)***

***END FUNCTION***

***FUNCTION Warning\_Signs\_Complete\_CheckedChanged(sender, e)***

***CALL UpdateCompletedTopics("Warning Signs", Warning\_Signs\_Complete.Checked)***

***END FUNCTION***

***FUNCTION Direction\_Signs\_Complete\_CheckedChanged(sender, e)***

***CALL UpdateCompletedTopics("Direction Signs", Direction\_Signs\_Complete.Checked)***

***END FUNCTION***

***FUNCTION Information\_Signs\_Complete\_CheckedChanged(sender, e)***

***CALL UpdateCompletedTopics("Information Signs", Information\_Signs\_Complete.Checked)***

***END FUNCTION***

***FUNCTION Road\_Work\_Complete\_CheckedChanged(sender, e)***

***CALL UpdateCompletedTopics("Road Work Signs", Road\_Work\_Complete.Checked)***

***END FUNCTION***

***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***

***END FUNCTION*Obstacle:** when the user goes back to the menu and open traffic page again the checkbox becomes empty  
  
**Solution:** The issue arises because the state of the checkboxes (whether they are checked or not) is not being preserved when navigating away from the Traffic\_Signs\_page and reopening it. This happens because each time the form is opened, a new instance of Traffic\_Signs\_page is created, and the checkboxes are reset to their default state (unchecked).

To fix this, I will store the state of the checkboxes (whether they are checked or unchecked) in a persistent location, such as a global variable, a file, or a database. When the Traffic\_Signs\_page is reopened, I can retrieve the saved state and update the checkboxes accordingly.  
 **Psuedocode:**  
***DEFINE Static Dictionary CheckboxStates AS DICTIONARY***

***{***

***"Giving Orders": FALSE,***

***"Warning Signs": FALSE,***

***"Direction Signs": FALSE,***

***"Information Signs": FALSE,***

***"Road Work Signs": FALSE***

***}***

***DEFINE Global List CompletedTopics AS EMPTY LIST***

***FUNCTION Traffic\_Signs\_page()***

***// Initialize page components***

***CALL InitializeComponent()***

***// Restore the checkbox states when the form loads***

***CALL RestoreCheckboxStates()***

***END FUNCTION***

***// Function to restore checkbox states***

***FUNCTION RestoreCheckboxStates()***

***// Restore each checkbox state from the static dictionary***

***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"]***

***// Debug logging for validation***

***PRINT "Checkbox States Restored:"***

***FOR EACH entry IN CheckboxStates***

***PRINT entry.Key + ": " + entry.Value***

***END FOR***

***END FUNCTION***

***// Checkbox event handler to update the completed topics***

***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***

***// Update the static dictionary with the checkbox state***

***SET CheckboxStates[topic] = isCompleted***

***// Debug logging for validation***

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

***END FUNCTION***

**Iteration 2:**

**Problem:** How to put pictures with text so the user can study signs from?  
  
**Solution:** I will add a new page under each topic so that the user can see what signs belong to that particular topic. When the user clicks on a topic, they will be directed to a page that displays all the relevant signs under that topic.

**Psuedocode**:  
***FUNCTION Orders\_Signs\_Click(sender, e)***

***CREATE nextForm AS new Orders\_Signs()***

***CALL nextForm.Show()***

***CALL this.Hide()***

***END FUNCTION***

**Problem:** How to put pictures with text?  
  
**Solution:** Implementation of a datagrid with scroll wheel

**Psuedocode**:  
***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 // Adjust row height to fit images***

***SET signsGridView.AllowUserToAddRows TO false***

***SET signsGridView.ReadOnly TO true***

***CREATE imageColumn AS new DataGridViewImageColumn()***

***SET imageColumn.HeaderText TO "Sign Image"***

***SET imageColumn.Name TO "ImageColumn"***

***SET imageColumn.ImageLayout TO DataGridViewImageCellLayout.Zoom***

***ADD imageColumn TO signsGridView.Columns***

***CREATE infoColumn AS new DataGridViewTextBoxColumn()***

***SET infoColumn.HeaderText TO "Information"***

***SET infoColumn.Name TO "InfoColumn"***

***ADD infoColumn TO signsGridView.Columns***

***CALL AddSignRow(signsGridView, "Signs with red circles are mostly prohibitive.\r\nPlates below signs qualify their message.", "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")***

***IF signsGridView.Rows.Count > 1 THEN***

***SET signsGridView.Rows[0].DefaultCellStyle.Font TO new Font("Arial", 10, FontStyle.Bold)***

***ADD signsGridView TO Controls***

***END FUNCTION***

***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)***

***CREATE signImage AS Image.FromFile(imageFullPath)***

***CALL grid.Rows.Add(signImage, info)***

***END FUNCTION***

**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***