**Design Section**

**Prototype 1 - Main Page:**

**Overview:**Overall, I intend to design the button-based navigation workflow necessary for users to navigate in between the different forms (pages). This section will therefore have a flow of what the user can do in the program I’m creating, the creation of the different form instances, and how transition to the next page will work. Where listing is needed, I will try to ensure proper navigation and data management.

**Decomposition to computable sections:**

|  |  |
| --- | --- |
| **Section** | **Justification (suitable for computation because…)** |
| Mapping Buttons to Open New Forms | This will allow my end user to navigate without errors by utilising different event handlers in the program to create and display new form instances. As a result I hope this will make sure user actions (button clicks) are mapped to their respective pages while also maintaining smooth transitions throughout the program as necessary per my analysis for my stakeholders listed above in requirements. |
| Implementing Navigation for Specific Pages | By assigning buttons to target pages (e.g., "Mock Test" opens InstructionsForm), the program can perform the way I want it to do for ease of user navigation in a predictable manner for testing and development throughout. This will also further help me eliminate errors, making sure the correct form opens in response to clicks on the buttons in the program by the end user. |
| Passing Data to Progress Page | Allows the Progress\_Page to show relevant user data - for example test scores. This will allows me to pass a dictionary from a global state in this program, making the content dynamic and real-time, as needed for my program relating to the analysis carried out which states that the user experience is tied directly to the ability to check their progress out. |
| Finalized Navigation Workflow | Allows for a logical workflow in relation to user interaction, navigation and data accessibility life cycle within my program. This will hopefully allow me to add smooth, button-based transitions while keeping the codebase fully functional and prevent any errors that hinder user experience. |

***Step 1: Mapping Buttons to Open New Forms***

To navigate between different pages, I will create event handlers for every button. When the user clicks a button, the program will create an instance of the corresponding form, show it and optionally hide the current form in view.

**Pseudocode:**

WHEN button is clicked:

CREATE a new instance of the target form (e.g., PracticePage)

OUTPUT the target form

HIDE the current form (optional)

**Reason:**  
Mapping buttons to specific actions will allow my end users to navigate through the program seamlessly which is, as emphasised above and in my overall documentation, key to making this program useful – the ability for users to interact with the program and transition between the different forms with no errors or mishaps. By creating a new form instance on a button click, I can therefore make sure that the program responds immediately to the user’s input, improving interactivity and therefore the user experience. Hiding the current form (optional) will also prevent the clutter on the screen for my goal of user experience optimisation.

**Approach:**

|  |
| --- |
| * **For all navigation buttons, I will define click event handlers.** |
| * **Event handlers will be created for each form to instantiate and display it.** |
| * **This overall should guarantee that the actions taken by users are mapped to the respective targets they were intended for within the program.** |

***Step 2: Implementing Navigation for Specific Pages***

In this next step, I need to add functionality to each button to allow users to open the appropriate page. These will consist of buttons such as "Mock Test," "Practice Page," and "Progress Page."

**Pseudocode for Mock Test Page:**

FUNCTION Mock\_test\_Click

CREATE new instance of InstructionsForm called nextForm

OUTPUT nextForm (Show it on the screen)

HIDE the current form

END FUNCTION

**Reason:**  
By clearly linking each button to its associated page (for example, the Mock Test button leads to the InstructionsForm), I can make sure that the program operates as the user expects it to. This is important for my intended user experience since the appropriate pages will load when users click on these buttons in the UI.

**Approach:**

|  |
| --- |
| * **When Mock Test Button is pressed, I will try to make sure InstructionsForm page shows instructions before the user takes the test.** |
| * **Completely check code for button references and their respective matches to prevent any bugs.** |

***Step 3: Passing Data to Progress Page***

To create the Progress\_Page, I will need to provide a dictionary that monitors the user’s test scores. This dictionary will then keep track of the tests that have been taken along with their respective scores for example Matthew scored 50% on Test 1. When I create the Progress\_Page, I will then be able to obtain the test scores from either a global state data class.  
  
**Pseudocode:**

FUNCTION Progress\_Click

RETURN testScores dictionary from GlobalData

CREATE new instance of Progress\_Page called nextForm, passing testScores as an argument

OUTPUT nextForm

CLOSE 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.  
  
The Progress\_Page should have access to the data of user progress such as attempted tests and scores. I call the global state for testScores dictionary and pass it to the Progress\_Page, therefore the page returns useful and current info. It also encourages separation of concerns, with data management being independent from UI navigation. I will set up a logic to get test scores from a global data store.

**Approach:**

|  |
| --- |
| * **I will add logic which will aim to retrieve the test scores from a global data store.** |
| * **When I navigate to that form, I will pass this dictionary to Progress\_Page constructor.** |
| * **This should allow *Progress\_Page* to display scores dynamically and provide real-time feedback to the end user improving the user experience.** |

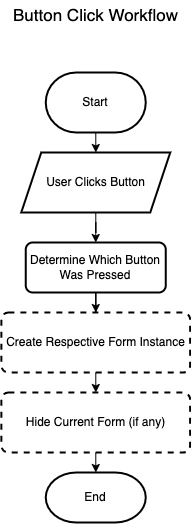
**Final Workflow Summary**

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

1. Since the state changes as described above, the program listens for individual button clicks.
2. Form Navigation Based on the button click, I will instantiate the relevant form and then show 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 can 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 should 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:**

DECLARE userAnswers AS DICTIONARY

FUNCTION ProcessQuestions(questionsList)

currentQuestionIndex = 0

WHILE currentQuestionIndex < questionsList.Count

OUTPUT questionsList[currentQuestionIndex.QuestionText]

OUTPUT questionsList[currentQuestionIndex.Answers]

OUTPUT “PLEASE GIVE AN INPUT”

INPUT 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

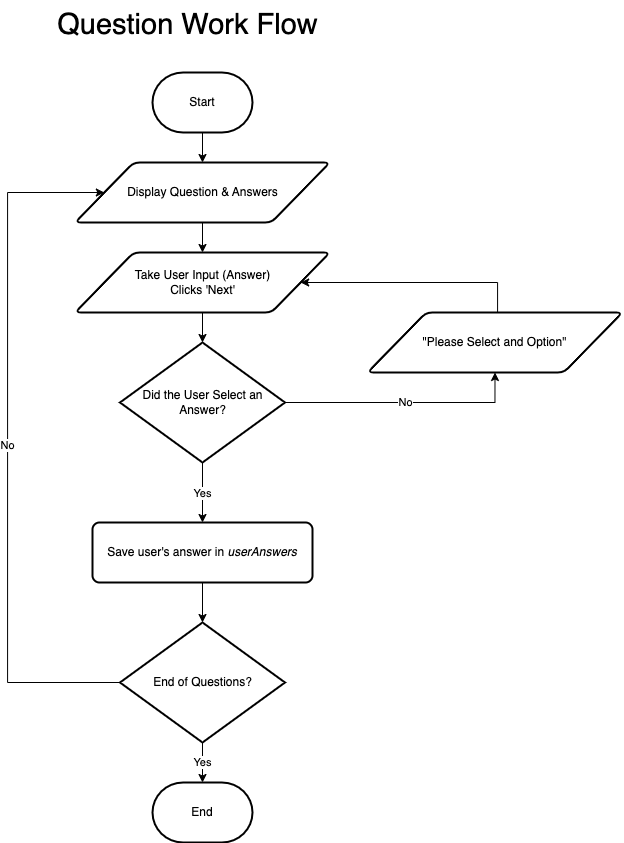
OUTPUT "All Questions Completed. Thank you!"

END FUNCTION

**Explanation:**

1. Dictionary Storage: Each answer is stored in a dictionary with question index as the key and user selected answer as the value.
2. Display and Input: Questions and answers are displayed, and user input is taken dynamically.
3. Validation : If the user does not select an answer, the program stays at the current question.
4. End Condition: The loop terminates when all questions are processed, and a completion message is shown. This is met when the user has gone through all the questions, and it displays a completion message.

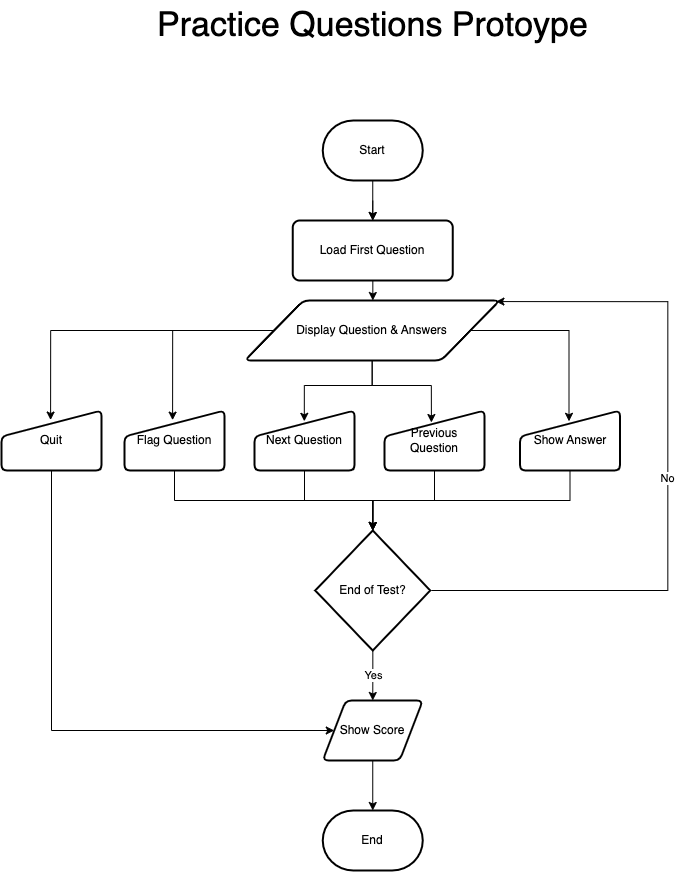
**Flowchart Alignment:**

****

**Prototype 2 - Practice Page**

**Overview:**In this section, I will design the workflow for the Practice Page, which encompasses navigation management, question management, and answer tracking from user inputs. The goal is to implement simple transitions between questions, calculate the user’s score, and display feedback based on user’s selections.

**Flowchart Representation:**

****

**Decomposition into Computable Sections:**

|  |  |
| --- | --- |
| **Section** | **Justification (suitable for computation because…)** |
| Mapping Buttons to Navigate Between Questions | Enables users to navigate between questions with Next and Previous buttons. This will make sure the end user can navigate through the questions whilst keeping track of your 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 for an effective UI in my program. |
| Updating the Score Dynamically | I need to try to dynamically calculate and updates the score based on the user's responses. This will provide real-time feedback and improves the user experience. |
| Display Correct Answers After Completion | The program needs to makes sure that after the user completes the test, they can see the right answers. This provides useful feedback for the user and aids retention as highlighted in my analysis. |

***Step 1: Mapping Buttons to Navigate Between Questions***

I am going to implement and allow users to use Next and Previous buttons to navigate through the questions. These buttons also update the currentQuestionIndex when clicked and route to the appropriate question.

**Pseudocode:**

WHEN NextButton\_Click:

IF currentQuestionIndex < selectedTest.Questions.Count - 1:

currentQuestionIndex =+1

LoadCurrentQuestion()

WHEN PreviousButton\_Click:

IF currentQuestionIndex > 0:

currentQuestionIndex =-1

LoadCurrentQuestion()

**Reason**: By mapping buttons to navigate actions, users can progress through the test in order and maintain flow. Also, the buttons will make sure that you iterate through the questions instead of going in an unwanted manner (for example, going from first to third question while skipping a second one which would be undesirable for the end user).

**Approach:**

* Attach click event handlers for 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 know the index of the current question for it to show the correct question at any time. I am going to show the number of current question (e.g. Question 2 of 5).

**Pseudocode:**

FUNCTION UpdateQuestionDisplay

trackerLabel = Question [currentQuestionIndex + 1]   
END FUNCTION

**Reason:** Showing the current number of question allows users to see where they are in the test, which helps in navigation and in turn users know how far along we are.

**Approach:**

|  |
| --- |
| * **Change the question index every time the user clicks on next or previous in test.** |
| * **Ensure the trackerLabel always reflects the current position in the test.** |

***Step 3: Updating the Score Dynamically***

Here’s how the user will be scored as they answer each question. The user is given instant feedback on whether their answer was right or wrong.

**Pseudocode:**

FUNCTION UpdateScore(selectedOptionIndex)

IF selectedOptionIndex == CorrectOptionIndex(currentQuestion)

Score = +1  
END FUNCTION

**Reason:** As a result of this, the user will receive live feedback on their score; this information allows them to see how many questions they answered correctly thus providing a sense of real-time performance metrics.

**Approach:**

|  |
| --- |
| * **Check the selected answer against the correct answer and update the score.** |
| * **Display the score update in a dynamic way, either after each question or at the end of the test.** |

***Step 4: Displaying Correct Answers After Completion***

Once the user completes the test, the program should display the correct answers for each one of the questions. This feedback will enable users to understand and learn from their mistakes.

**Pseudocode:**

FUNCTION ShowCorrectAnswers

FOR LOOP question IN Questions(selectedTest)

IF selectedOptionIndex != CorrectOptionIndex(question)

Highlight incorrect answers

ELSE

Highlight correct answer  
END FUNCTION

**Reason**: Displaying the correct answers after the test is completed will allow users to learn from their mistakes and reinforce the correct answers for retention and understanding of the material as required in reference to my aims for the app.

**Approach:**

|  |
| --- |
| * **Once the test is over, the aim is to loop through each question and check the answer against the correct one in the database.** |
| * Highlight answers appropriately (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:

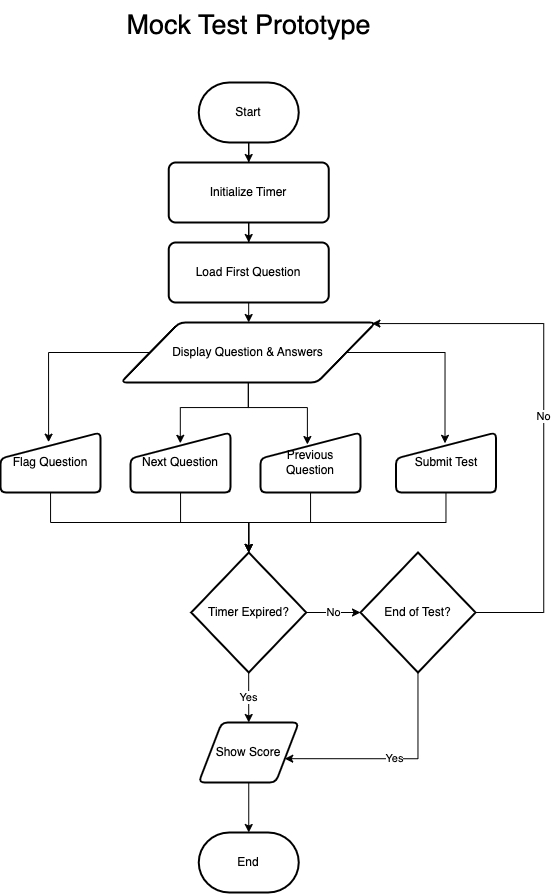
1. Navigation: The user navigates through the questions using the Next and Previous buttons, with the current question being displayed dynamically.
2. Score Calculation: The program updates the score as the user answers each question, providing real-time feedback.
3. Answer Display: Once the test is 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 as needed for this program to be successful in it’s approach of teaching users.

**Prototype 3 - Mock Test Page**

**Overview:**   
  
The functionality of Mock Test Page will be similar to that of the practice page as laid out above but it will have the added countdown timer feature. The timer is necessary in order to emulate the examination that users are preparing for. The users will be able to navigate through each question where the user can flag important questions, and review their performance at the end in the form of a ‘Mock Examination’ as highlighted in the analysis section of my project. This is a key feature for my stakeholders.

**Flowchart Representation:**

****

Decomposition into Computable Sections:

|  |  |
| --- | --- |
| **Section** | **Justification (suitable for computation because…)** |
| Implementing Timer | The countdown timer keeps track of the test duration and automatically ends the test once the allotted time has run out mirroring the conditions of a real exam. |
| Updating Timer | The timer needs to be updated every second to display the time the user has remaining and end the test when this time hits zero regardless of questions left. Reflecting the examination once again. |
| Flagging Questions | Allows users to flag questions they want to review later and navigate through these flagged questions. |
| Displaying User’s Choices and Results | After completing the test, the app needs to display the user’s selections and correct answers. |
| Displaying Review of Incorrect Answers | Providing feedback to the user’s by highlighting all of the incorrect answers in red and the correct answers in green. |

### ***Step 1: Adding a Timer Feature***

**Feature:**  
Implement a countdown timer to display the remaining time for the Mock Test.

**Pseudocode:**

WHEN InitializeTimer:

Set timeRemaining = 30 \* 60 // 30 minutes

Start quizTimer every 1 second

Display timeRemaining on timerLabel

WHEN quizTimer\_Tick:

IF timeRemaining > 0:

timeRemaining = -1

Update timerLabel with remaining time

ELSE:

Stop quizTimer

Show final score

ENDWHILE

**Reason:**  
The test needs to utilise a countdown timer to keep the user aware of the time remaining in order to complete the quiz within the time limit.

**Approach:**

|  |
| --- |
| * Upon page loading, create and start the timer. |
| * Update the time every second and display it in a label. |
| * Stop the timer when the time runs out and show the final score. |

### ***Step 2: Stopping the Quiz When Time Runs Out***

**Feature:**  
Finish the test automatically when the timer reaches 0 and show the score.

**Pseudocode:**

WHEN quizTimer\_Tick:

IF timeRemaining == 0:

Stop quizTimer

Call ShowScore() to display the final score

ENDWHILE

**Reason:**  
This will ensure the test ends immediately when the timer expires, preventing any further answers or changes to any answers given by the user.

**Approach:**

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

### ***Step 3: Flagging Questions for Review***

**Feature:**  
Implement the ability for the user to flag certain questions that they can review later as needed for a Mock Test.

**Pseudocode:**

WHEN FlagButton\_Click

IF currentQuestionIndex is not in flaggedQuestions:

Add currentQuestionIndex to flaggedQuestions

ELSE

Remove currentQuestionIndex from flaggedQuestions

**Reason:**  
Flagging questions will allow the user to mark the questions they want to revisit later, improving their ability to understand and revisit difficult questions.

**Approach:**

|  |
| --- |
| * **Initialise 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 the user to skip flagged questions and answer the next one - without selecting an answer just like the real examination.

**Pseudocode:**

WHEN NextButton\_Click

IF currentQuestionIndex is flagged

currentQuestionIndex = +1

LOAD Next Question

ELSE IF selectedAnswer is valid

Save the answer and proceed

**Reason:**  
This feature will make it so that users aren’t forced to answer flagged questions immediately, giving them the ability to skip and return later.

**Approach:**

|  |
| --- |
| * **Check if the current question is flagged before navigating further.** |
| * **Skip flagged questions if the user decides to move forward without answering.** |

### ***Step 7: Tracking User Answers***

**Feature:**  
Track and store answers given by the user for review after the test has been completed.

**Pseudocode:**

WHEN NextButton\_Click

Save selectedanswer in userAnswers[currentQuestionIndex]

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

**Approach:**

|  |
| --- |
| * **Store each selected answer in a dictionary (userAnswers).** |
| * **Update this 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 inputted selections during the mock examination.

**Pseudocode:**

FUNCTION ShowCorrectAnswers:

FOR LOOP 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 for each question that the user did not answer correctly after the quiz helps users learn from their mistakes.

**Approach:**

|  |
| --- |
| * **Iterate over the answers and highlight them (are they incorrect or correct?).** |
| * **Display feedback after the quiz is completed to show the correct answers to the questions.** |

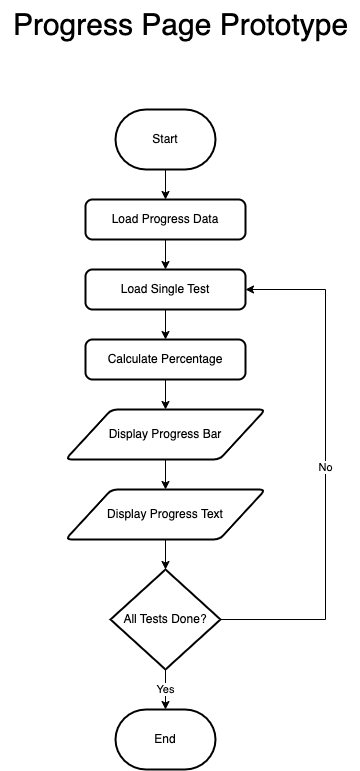
**Prototype 4 - Progress Page:**

**Overview:**

### The **Progress Page** visually tracks a user's test performance using progress bars and percentages. Progress tracking is necessary for my program to be successful, in order to help users improve their retention ability and tackle key areas where they are not succeeding. This page is therefore designed to dynamically display progress for both mock tests and practice tests. Key features will 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.

### **Flowchart Representation:**

****

### **Decomposition into Computable Sections:**

|  |  |
| --- | --- |
| **Section** | **Justification (suitable for computation because…)** |
| **Displaying Progress for Practice Tests** | Adds the practice test scores to a global score dictionary and displays them alongside the mock test progress for user accessibility. |
| **Global Score Tracking** | Allows the program to store and retrieves these scores from a global dictionary for both mock tests and practice tests – essential for progress tracking which is a key feature necessary for my stakeholders. |
| **Label Identification for Test Types** | Adds labels for example "Mock Test Progress" and "Practice Test Progress" for clarity when displaying progress in examinations. |
| **Completed Topics Tracking** | Uses a **timer** to monitor and update the list of topics completed by the user for accurate progress tracking for the end user. |
| **Dynamic UI Updates** | Creates progress bars and labels dynamically, updating content from the latest data for scores and topics. |

### ***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 testScores as Dictionary

SET testScores = { " KEY " : " test number ",   
" VALUE " : “test score” }

Define testQuestions as Dictionary

SET testScores = { " KEY " : " test number ",   
" VALUE " : “total number of questions” }

FUNCTION SaveScore(testNumber, score)

IF testScores does not contain key testNumber  
 THEN

testScores[testNumber] = score

END IF

END FUNCTION

FUNCTION CalculateScorePercentage(testNumber)

IF testScores contains key testNumber AND testQuestions contains key testNumber  
 THEN

RETURN (testScores[testNumber] \* 100) / testQuestions[testNumber]

ELSE

RETURN 0

END IF

**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)

percentage = CalculateScorePercentage(testNumber)

Minimum(progressBar) = 0

Maximum(progressBar) = 100

Value(progressBar) = percentage

Define percentageLabel as Label

IF percentage > 0 THEN   
 Text(percentageLabel) = (percentage)   
 ELSE   
 OUTPUT "Not Attempted"   
 ENDIF

END FUNCTION

**Reason:**  
Progress bars will allow my user to see a visual representation of their progress and labels help provide clarity with exact percentages, which is also needed for improvement and boosting ‘morale’ in their performance as indicated by my stakeholders for this project.

**Approach:**

|  |
| --- |
| * **Insert the Progress Bar to display all scores visually for the user’s convenience.** |
| * **Calculate percentages of tests and update the according labels alongside progress bars.** |

### ***Step 3: Integrating Mock and Practice Test Scores***

I will try to make sure that the progress displays to include both Mock and Practice tests by creating individual dictionaries for both.

**Pseudocode:**

Create Class GlobalData

MockTestScores = DICTIONARY

PracticeTestScores = DICTIONARY

FUNCTION ShowScore(testType, testNumber, score)

IF testType == "Mock"

MockTestScores[testNumber] = score

ELSE IF testType == "Practice"

PracticeTestScores[testNumber] = score

END IF

End FUNCTION

FUNCTION DisplayTestProgress()

Define mockProgressLabel AS Label  
 Display Progress Bars for Mock Scores

Define practiceProgressLabel AS Label  
 Display Progress Bars for Practice Scores

End FUNCTION

**Reason:**  
Tracking Mock and Practice tests separately will ensure even more clarity for the evaluation of progress on the Progress page for the user and indicate possible discrepancy between Timed test & normal practice questions – to help boost performance in the real exam.

**Approach:**

|  |
| --- |
| * **Host two separate dictionaries in my program to store Mock and Practice Test Scores** |
| * **Display both progress metrics individually in their own distinct sections for easier viewing.** |

### ***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.

To automatically track topics that have been completed in the program, I aim to add a timer that will check for user actions updates every second and therefore dynamically display the completed topics.

**Pseudocode:**

**Define updateTimer as a Timer**

**Define topicsCompletedLabel as a label**

FUNCTION InitializeTopicsCompleted()

Interval(updateTimer) = 1 second  
 Tick(updateTimer)  
 Call UpdateTopicsCompleted() function

FUNCTION UpdateTopicsCompleted()

completedTopics = CompletedTopicsCount()  
 IF completedTopics > 0   
 topicsCompletedLabel = ("Topics Completed: ", completedTopics)  
 ELSE   
 topicsCompletedLabel = "Topics Completed: None"

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 during execution of my program.** |
| * **Update the topics label based on the count of completed topics by user.** |

### ***Step 5: Final Workflow Summary***

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

1. **Tracking Progress:** Save user’s scores and calculate percentages for each test taken.
2. **Displaying Progress Bars:** Progress bars and labels for Mock and Practice tests will be shown separately for better UX.
3. **Integrating Practice Scores:** Implement and use dictionaries to track Mock and Practice test scores throughout the execution of the entire program.
4. **Dynamic Updates:** Use a timer to track topics that have been completed and update this in 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.

**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**
   * This will guarantee that the positions of the marked questions are kept in an appropriate format for example a dictionary with a list of the positions of the questions marked as the value and the test number as the key.
2. **Accessing Flagged Questions**
   * Use the test number to access flagged question position.
   * Retrieve the corresponding questions and their correct answers from Global Data.
3. **Dynamic UI Creation**
   * Iterate through the questions that have been flagged and create labels for each question and it’s corresponding correct answer.
   * Add these labels to a **panel**.
4. **Scrollable UI Integration**
   * To manage situations where there are several marked queries, arrange all generated content inside the panel.

### **Final Steps and Functionality:**

1. **Store Flagged Positions**: Ensure flagged questions are tracked during the test.
2. **Pass Flagged Questions**: Provide flagged positions when navigating to the Flagged Questions Page.
3. **Retrieve Data**: Fetch flagged questions from Global Data.
4. **Display Data**: Labels are dynamically created and positioned to show the text of the question and the appropriate response.
5. **Scrollable UI**: Make sure a scrollable interface is available for viewing all of the questions.

**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 checkboxstate as a static 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 will allow for persistence during navigation and make sure that the states remain consistent throughout execution.

**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()

Giving\_Order\_Complete.Checked = CheckboxStates["Giving Orders"]

Warning\_Signs\_Complete.Checked = CheckboxStates["Warning Signs"]

Direction\_Signs\_Complete.Checked = CheckboxStates["Direction Signs"]

Information\_Signs\_Complete.Checked = CheckboxStates["Information Signs"]

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 = TRUE

IF topic = NOT IN CompletedTopics

Add topic to CompletedTopics

ELSE

Delete topic from CompletedTopics

END IF

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 OrdersSignClick

CREATE nextForm AS new Orders\_Signs()

CALL nextForm.Show()

Hide current form

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()

Define signsGridView

Define imageColumn

Add(imageColumn)(Columns(signsGridView))

Define infoColumn

Add(infoColumn)(Columns(signsGridView))

AddSignRow(signsGridView, "Signs with red circles.", PIC.png)

AddSignRow(signsGridView, "Entry to 20 mph zone", PIC.png)

AddSignRow(signsGridView, "End of 20 mph zone", PIC.png)

END FUNCTION

***Adding Rows:***

FUNCTION AddSignRow(grid, info, imagePath)

imageFullPath = ("Signs\_Giving\_Order", ImagePath)

Define SignImg AS Image   
 SignImg = FromFile(Image)(imageFullPath)  
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.

### ***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.

**Prototype 7 - Settings Page:**  
**Overview:**

In this section, I will plan the Settings Page workflow, allowing users to adjust the background color, font size, and font style. The settings page will include controls such as a ColorDialog for changing the background color, a slider for adjusting the font size, and checkboxes for selecting font styles (bold/italic). I will also ensure smooth transitions between forms, with updates being applied globally across all forms.

**Decomposition to computable sections:**

|  |  |
| --- | --- |
| **Section** | **Justification (suitable for computation because…)** |
| Mapping ColorDialog to Change Background Color | The background color will be changed based on user selection. A ColorDialog is suitable for allowing users to select a color dynamically. |
| Implementing Font Size and Style Adjustments | Tracks changes to checkboxes, updates the state in a persistent storage, and maintains completed topics. |
| Centering Forms on Screen | The font size will be adjusted with a slider, and the font style (bold/italic) will be toggled with checkboxes. This ensures a dynamic user interface. |
| Resetting Data | The reset functionality needs to delete specific files to clear mock test scores and completed topics, ensuring data management is handled efficiently. |
| Finalized Settings Workflow | The finalized settings workflow will ensure smooth application of user preferences like background color, font size, and style. The reset button will handle data deletion and inform the user of the results. |

***Step 1: Mapping ColorDialog to Change Background Color***

To allow users to change the background color, I will implement a button that opens a ColorDialog. When the user selects a color, the program will update a global variable for the background color and notify the user about the update.

**Pseudocode:**

Procedure Change\_Color\_Click

Open a new ColorDialog

If the user selects a color in the ColorDialog

Set the global variable related to bg colour to the user’s selected color

Display a notification box (message box) to let the user know the bg color has been changed

End Procedure

**Reason:** This step allows users to make visual changes to an application by setting a new background color. Since this will be implemented as a global change, all the forms will then reflect that updated background on reopening - hopefully providing consistency within the application.

**Approach:**

* Add a button labeled "Change Background Color" to my Settings Page.
* When this button will be clicked, the button will then open the ColorDialog.
* User will select a color and this will be stored in a variable.
* A prompt will appear that informs the user that this change has taken effect.

***Step 2: Implementing Font Size and Style Adjustments***

Users will be able to adjust the font size using a slider, and choose bold or italic styles using checkboxes. The global font settings will be updated dynamically as the user interacts with the controls.

**Pseudocode:**

*Pseudocode for Font Size:*

Procedure FontSizeSlider\_Scroll:

Update the global font size based on the slider value

FontSize = Value(fontSizeSlider)

Update the font size of the preview label

End Procedure

*Pseudocode for Font Style (Bold/Italic):*

PROCEDURE FontStyle\_Change

fontStyle = Regular(FontStyle)

IF Checked(boldCheckBox)

fontStyle = Bold(FontStyle)

END IF

IF Checked(italicCheckBox)

fontStyle = Italic(FontStyle)

END IF

Update the font style globally for all pages

Update the font of the preview label in settings view

Display a message about the font style update (applies to all pages now for UI experience)

END PROCEDURE

**Reason:** The slider allows the user to easily change font size, while the checkboxes give them control over the font style. This ensures a personalized user interface. The preview label immediately updates so the user can see the changes in real-time.

Approach:

* A slider will allow users to adjust the font size.
* Two checkboxes will control whether the font is bold or italic.
* The font size and style will be updated dynamically, with a preview label showing the updated font.
* A message box will confirm the changes when the style is updated.

***Step 3: Centering Forms on Screen***

When any form is opened, I know that Windows Forms won’t automatically center this text. Therefore as I need it to be centered on the screen to enhance the user experience I will manually implement this code into every page.

**Code:**

this.StartPosition = FormStartPosition.CenterScreen;

**Reason:** Centering forms ensures a consistent and professional look across different screen sizes and resolutions. It also ensures that the form will be easily visible to the user when it is opened.

**Approach:**

* I will ensure that every form has this line of code to center it on the screen when it is loaded.

***Step 4: Resetting Data***

The reset functionality will allow users to delete test scores and completed topics by removing the relevant files. If any of the specified files exist, they will be deleted. The user will be informed of the outcome.

**Pseudocode:**

PROCEDURE reset\_Click

filesToDelete = []

SET filesToDelete = [

"PracticeScores.txt",

"Mock\_Score.txt",

"Flagged\_Questions.txt",

"data.txt"

]

filesDeleted = FALSE

filesExist = FALSE

FOR loop file IN filesToDelete

IF FileExists(file)

filesExist = TRUE

DeleteFile(file)

filesDeleted = TRUE

END IF

END FOR

IF filesDeleted = TRUE

OUTPUT "Data has been deleted."

ELSE

OUTPUT "No data to reset."

END IF

END PROCEDURE

**Reason:** This feature allows users to reset their data, which can be useful if they want to start fresh or clear unnecessary data. The user will receive feedback on whether the operation was successful.

**Approach:**

* The reset button will trigger the deletion of the specified files.
* If any files are deleted, a confirmation message will appear. If no files are found, a warning message will notify the user.

**Final Workflow Summary**

1. User Action: The program listens for button clicks or slider movements.
2. Settings Management: Upon interaction, the settings (background color, font size, font style) will be updated in real-time.
3. Form Behaviour: The forms in the program will open centered on the screen for a consistent UX as needed for a smooth consistent learning experience for my users without ‘clutter’.
4. Resetting Data: The reset button will clear all specified files.

**Prototype 7 - Hazard Perception Test Page:**  
**Overview:**

In this section, I will plan the Hazard Perception Test feature for the application, allowing users to interact with a video. These videos will have have hazards which will need to be flagged similar to the real theory exam. As a result the user will need to flag specific timestamps and receive points for these correct flags.

**Decomposition to computable sections:**

|  |  |
| --- | --- |
| **Section** | **Justification (suitable for computation because…)** |
| Map the Video Playback to User Interaction | The video will play and pause based on user interaction, providing a base for flagging timestamps that can be identified as the hazards in the video. |
| Flagging Timestamps on Video | The user will be able to click specific points in the video, which will need real-time detection of the timestamp. |
| Awarding Points for Correct Flags | Points will be awarded for correctly flagged timestamps within a specified time range. |
| Displaying Real-Time Timestamp | The current video timestamp will be displayed dynamically during playback. |

***Step 1: Mapping Video Playback to User Interaction To allow users to interact with a video.***

I will use a Windows Media Player control to load and play the video. The video will be played as soon as the form loads, and the timestamp will be updated every second.

**Pseudocode:**

PROCEDURE Hazard\_perception\_test\_Load

SET URL(axWindowsMediaPlayer1) TO "Path of Video"

SET Interval(timer) TO 500

Start()(timer)

END PROCEDURE

PROCEDURE timer\_Tick

currentMedia(axWindowsMediaPlayer1)  
   
 SET currentTime TO VideoTime  
   
 SET Text(label1) TO currentTime

END IF  
END PROCEDURE

**Reason:** This step will make sure that the video starts playing as soon as the user enters the test, and the current timestamp is displayed for reference when a flag is placed.

**Approach:**

* Implement a Windows Media Player control.
* Start the video when the form loads.
* Update the timestamp on every tick of a timer.

***Step 2: Flagging Timestamps on Video Users will be able to flag specific timestamps by clicking on the form during playback. When a user clicks, the current timestamp of the video will be stored.***

**Pseudocode:**

PROCEDURE Form\_Click

currentTime = currentPosition(Ctlcontrols(axWindowsMediaPlayer1))

timestamp = FromSeconds(currentTime)(TimeSpan)

OUTPUT “Flagged Timestamp:”, timestamp

END PROCEDURE

**Reason:** This functionality allows the user to mark specific moments in the video making it interactive. The timestamp will then therefore be flagged when the user clicks the form.

**Approach:**

* Put in a click event handler for the form.
* Retrieve the current timestamp from the video and display this in a label format to the user.

***Step 3: Awarding Points for Correct Flags Users will be awarded points if they flag a timestamp within a specified time range (e.g., within the first 5 seconds of a relevant event in the video).***

**Pseudocode:**

Procedure AwardPoints (timestamp):

If timestamp >= targetStartTime And timestamp <= targetEndTime Then

Points = points +1

OUTPUT ”Correct flag! Points awarded."

Else

OUTPUT "Incorrect flag. No points awarded”  
 ENDIF

End Procedure

**Reason:** This can allow me to reward users for correctly identifying the hazards in the video as required in the real test.

**Approach:**

* Create a specific time window for valid flags to be raised by user input ( for example 5 seconds after an event).
* Check if the flagged timestamp falls within the valid time window range created.
* Award points accordingly to the time the flag is placed and display a message confirming this result for progress tracking.

***Step 4: Displaying Real-Time Timestamp During playback, the current timestamp will be continuously displayed on the form to give the user a reference point for flagging timestamps.***

**Pseudocode:**

PROCEDURE timer1\_Tick

currentTime TO currentPosition((axWindowsMediaPlayer1))

OUTPUT currentTime

END IF

END PROCEDURE

**Reason:** This will make sure that the user can see the current timestamp helping them decide when to flag a hazard in the video being played.

**Approach:**

* Use a timer to update the timestamp label every second.
* Display the timestamp in a label so users can refer to this easily.

**Final Workflow Summary:**

1. User Action: The user can click on the Form when a video is being played in order to mark/flag a hazard to the program.
2. Timestamp Flagging: This timestamp will be captured and displayed to the user.
3. Points Awarding: Points will only be awarded if the user flags the timestamp within the designated time range.
4. Real-Time Timestamp Display: The timestamp is updated continuously as the video plays.