Preparation: Before understanding the codebase, read its requirements that describe its intended functionality. Be aware that this implementation may differ from your previous experiences or knowledge. Throughout the debugging process, if needed, you can refer back to the codebase functionality description to avoid incorrect assumptions about any section's purpose or behavior.

Procedure:

1. Get a quick overview of the codebase to develop a high-level understanding of the code structure
    - 1.1. Start from the codebase's entry point, which is in the `main` in Program.cs.
    - 1.2. Trace the general control flow through the codebase, you do not need to focus on details. Take stock of the codebase structure. You might want to pay attention to:
        - 1.2.1. Functions/components;:
            - Form classes (e.g., departUpdateForm, Form2, departmentReport, depart_student_view)
            - Database operations (using SqlConnection and SqlCommand)
            - UI components (buttons, text boxes, error providers)
        - 1.2.2. Their locations within the code structure:
            - departUpdateForm.cs: Main form for updating department information
            - departUpdateForm.Designer.cs: UI design for the update form
            - Other form files mentioned in the project structure
        - 1.2.3. How they interact with each other (i.e., method calls)
            - Form navigation (e.g., opening departmentReport from departUpdateForm)
            - Database interactions in button click events
            - Input validation using error providers

2. Identify and examine potential bug-containing code sections
    - 2.1. First, decide which code sections require more thorough examination: Based on your overview gained from the previous step, prioritize sections with a higher chance of containing the bug. In this case, prioritize the `btnUpdate_Click` method (lines 36-83) in `departUpdateForm.cs` as it handles core update functionality:
        - 2.1.1. Start with the section that you believe is the most potentially bug-relevant.
        - 2.1.2. Trace the data flow through the section to identify key operations and their purpose, pay attention to how variables are manipulated: such as input validation (lines 41-60) and database update operation (lines 61-83). Read the code block carefully to determine this section's expected output or behavior. If needed, refer back to the overall functionality description to ensure accurate understanding.
        - 2.1.3. Identify what this section's input(s) should be and propose inputs likely to trigger the bug. The valid inputs for this method are textBoxCourse.Text and textBoxDuration.Text. Since different inputs may lead to different code paths, you might try multiple input scenarios if necessary. If you have no idea of how to choose the inputs, you could start by using the inputs provided in the codebase (Course = "Computer Science", Duration = "4" ).
        - 2.1.4. Perform mental calculations with your proposed inputs: For the test case, first step through the validation logic (line 41-60). If validation passes, follow the database update process (lines 61-83). Go through this section and calculate its intermediate output/behavior. Take notes to track variable manipulations and intermediate output.
        - 2.1.5. Compare the calculated output (or observed behavior) with the expected output:

            2.1.5.1. If match: conclude this section is likely bug-free, move to the next section that has a higher chance to contain the bug, and repeat from Step 2.1.2.
            2.1.5.2. If they don't match: conclude this section likely contains the bug. Form a hypothesis about which statement(s) are problematic. Based on your previous calculations, compare each statement's intermediate output/behavior with the expected output to identify the mismatch. Once identified, propose a fix and move to Step 3 to validate your hypothesis.
    2.2. If the bug remains undetected, revisit other potentially bug-relevant sections in `departUpdateForm.cs`, such as `btnDelete_Click` or input validation methods.
    2.3. If still unresolved, expand your analysis to sections initially considered less likely to contain the bug, applying the same process (Steps 2.1.2 to 2.1.5) to each.

3. Validate your proposed bug fix
    3.1. Focus on the specific code section you believe contains the bug (e.g., within `btnUpdate_Click`). Assume you've implemented the fix and other sections work correctly.
    3.2. Redo the mental calculation from Step 2.1.4 with the assumed fix in place. Take notes on recalculated intermediate outputs:
        3.2.1. If you are confident about your identified bug, you may choose to recalculate only the fixed statement.
        3.2.2. Otherwise, if you are less certain, you have the option to recalculate the entire section for a more thorough check
    3.3. Compare the new output with the expected output:
        3.3.1. If they match: Your proposed fix likely solves the bug
        3.3.2. If they don't match: Your fix may be incorrect, or this section may not contain the bug. Consider:
            3.3.2.1. If you have another hypothesis for this section, return to Step 3.1 to validate it.
            3.3.2.2. Otherwise, return to Step 2 to analyze other code sections.
    3.4. Repeat Steps 2 and 3 until the bug is resolved or all possibilities are exhausted.