**ShopEase app test analysis**

**Introduction**

The testing process is an essential part of the development of any application, especially when that application is complex like the cloud-based ShopEase app. The primary purpose of the testing process is to ensure that all functional requirements are met, performance verification, and ensure seamless integration between the different modules of the application. This analysis focuses on the different stages of the testing process that were conducted during the development of the ShopEase app, from individual unit testing to user acceptance testing. The analysis reflects how unit testing, functional testing, integration testing, and user acceptance testing affect system quality and performance.

**Test Objectives**

The test objectives for ShopEase were to ensure:

1. Meet the functional requirements of the application as defined by the customer.
2. All units work independently, ensuring they interact smoothly after integration.
3. Ensure system performance in a cloud environment, and efficiently process data without degradation in performance.
4. Ensure that the application is easy to use and compatible with business needs as expected by the customer.

In a ShopEase app development scenario, real-time data is required, whether it's product management, notifications, or customer analytics. So, the primary goal of testing was to ensure that all these elements work harmoniously and efficiently in the cloud environment.

**Types of tests performed**

**A. Unit Testing**

Unit tests centered around verifying the functionality of each unit individually within the system. For example, unit tests focused on modules such as product management, customer management, and notification system. The primary goal of unit testing was to ensure that each module performs its assigned tasks correctly without errors.

1. **Product Management Unit**
   * **Tested status**: Add, update, and delete products.
   * **Result**: These operations have been tested to verify that data is entered correctly and detect errors such as the absence of some required fields such as price or type.
   * **Bottom line**: All data validation errors have been corrected, ensuring that product additions or modifications are done properly before they are entered into the database.
2. **Customer Management Unit**
   * **Tested case**: Add new customers and edit or delete their data.
   * **Result**: Validation of the entered data was tested to ensure that important data such as name or contact was not lost.
   * **Bottom Result**: Some errors related to the non-verification of some fields were detected, and fixed to ensure that the unit worked properly.
3. **Notification Unit**
   * **Tested status**: Send notifications to all subscribers when a new product is added or an existing product is updated.
   * **Result**: Tested how correctly messages were sent to subscribers when new products were added.
   * **Bottom line**: Notifications were ensured to work efficiently, and some adjustments were made to the message format.

**B. Integration Testing**

The goal of the integration test was to verify the interaction between different modules within the system. In the ShopEase app, the focus was on making sure that different modules, such as product management and a notification system, interact seamlessly.

1. **Integrate product management with notification system**
   * **Tested status**: Ensure that notifications are sent when a new product is added.
   * **Result**: The test detected delays in sending notifications at times, and the problem was resolved by improving the mechanism of running notifications.
   * **Bottom line**: The notification sending mechanism has been adjusted to ensure it works in real time without delay.
2. **Integration of the analytics unit with product management**
   * **Tested case**: Verify the accuracy of reports extracted from product-specific data.
   * **Result**: Some errors were detected in fetching data which affected the accuracy of reports.
   * **Bottom line**: SQL queries have been corrected to improve reporting accuracy and ensure that the analytics module extracts data correctly.

**c. Functional Testing (Functional Testing)**

Functional testing focused on verifying that the application meets all the practical requirements set by the client. Functional tests included:

1. **Tested status**: Add a product and make sure the confirmation message appears.
   * **Result**: The model's behavior was tested while adding products and verified that the application successfully handles this process.
   * **Bottom line**: It has been ensured that the process is running smoothly and appropriate messages appear when the product is successfully added.
2. **Tested case**: Register new customers and manage their data.
   * **Result**: New customers have been tested and properly validated and stored.
   * **The end result**: it was ensured that the system worked correctly and the reliability of the processes related to customer registration was ensured.

**d. User Acceptance Test (UAT)**

User acceptance testing was the final step to ensure that the application met the business needs. This test was conducted in collaboration with the customer to ensure that the app is easy to use and performs the required functions.

1. **Tested case**: Product management, notification dispatch, and report generation.
   * **Result**: All processes from adding products to sending notifications and generating reports have been successfully tested.
   * **The end result**: the customer was satisfied with the performance of the system, and some minor modifications were proposed to improve the user interface.

**Environment and tools used in the test**

The tests were carried out in an environment similar to the production environment using:

* **MySQL database**: MySQL was used to store and retrieve data during tests.
* **Visual Studio debugging tool**: Used during unit testing and integration testing to identify and fix problems.
* **Breakpoints and monitors**: Breakpoints are set in sensitive places of code such as AddProduct and NotifySubscribers to track data flow.

**Summary of results**

* **Unit** testing: All tests passed successfully after minor repairs, and it was ensured that the core processes were working properly.
* **Integration test**: Some problems were discovered in the interaction between the units, and they were solved to improve the integration between the different components.
* **Functional** testing: It has been ensured that all the features that the user sees are working properly under normal and extreme circumstances.
* **User acceptance test**: The test was completed successfully and customer satisfaction, with some suggestions to improve the interface.

**The impact of testing and debugging processes on the quality of the final application**

Testing and debugging processes are an essential part of the software development lifecycle and have a direct impact on the final application quality. Through these processes, the development team can detect and correct bugs before the final application is released, reducing the problems that users may face later. To illustrate the impact of these processes on the final application quality, we discuss the following elements:

**Improve system stability:**

Careful testing and continuous correction help detect problems that may affect system stability. For example, bugs that cause an app to crash or freeze can be detected and corrected before launch. Without proper testing processes, users may experience serious issues that cause the user experience to decline.

In the process of developing this application, each software module was independently reviewed and tests were carried out to ensure that the modules were working properly either in an isolated environment or when combined with other modules.

**Improved performance:**

By correcting performance-related errors such as system slowness or unjustified resource consumption, the development team can ensure that the application runs quickly and efficiently. During the tests, emphasis was placed on measuring system response when adding products, sending notifications, and extracting reports from the analytics unit. Some issues related to response speed have been identified and corrected to ensure faster and more responsive performance.

**Enhance user experience:**

Comprehensive tests help improve the user experience by ensuring that every function in the system works as expected. For example, extensive testing of customer management functions and products was performed to ensure that the UI gives clear error messages when the input is incorrect or when some fields are empty. This reduces the frustration that the user may experience and improves their overall experience of using the application.

**Early error detection and analysis:**

One of the primary benefits of ongoing testing and debugging processes is the ability to detect errors at early stages of the development lifecycle. The earlier the error is detected, the less expensive it is to fix. By running tests at each stage of development, the development team was able to identify several bugs before merging the modules together, helping to minimize the impact of those bugs on the system as a whole.

**Ensure compliance with requirements:**

The test aims to verify that the final system meets the requirements set by the customer. For example, in this app, tests were performed to verify that all products, notifications, and reports appear and function correctly according to the customer's needs. Without proper testing, there may be gaps between what the customer asks for and what has actually been implemented.

**Evaluation of integration between software units:**

During the integration of software modules (such as product management, customer management, notification system, and analytics module), it was ensured that these modules worked together seamlessly without any adversely affecting the other. How data moves between different modules has been tested, such as sending customer notifications when a new product is added or an existing product is updated. This testing process helped identify some integration issues, which were addressed directly.

**Final quality assurance of the application:**

The quality of the final application depends heavily on the effectiveness of the testing and debugging processes. The more structured and detailed these processes, the better the result. By implementing thorough testing and debugging processes at every stage of development, the team can ensure that the final product does not have major problems and that it meets user expectations.

**The bottom line**

The overall testing process has proven that the ShopEase app is ready to launch. Through the use of a combination of unit testing, integration testing, functional testing, and user acceptance testing, it is ensured that the system works efficiently and meets all customer requirements.

**Test Report: Product and customer management application**

This report examines the results of different test cases performed on the Product Management, Customer Management, and Notification System modules. Each test is designed to ensure that the application works as expected in normal scenarios and edge situations.

**Product management test cases:**

**Test case 1: Add a new product**





* **Inputs**: Enter a valid product type, size, color, price, and inventory.
* **Expected result**: The product is added successfully and the DataGridView is updated to display the new product.
* **Result**: **Success** The product was successfully added and displayed in the DataGridView.

A screenshot of a product management

Description automatically generated

* **Edge status**: Leave the Price field blank.
* **Expected result**: The system should block the extension and display an error message.
* **Result**: **Successful** system display error message indicating that price is required and the product has not been added.

**Test case 3: Delete a product**

A screenshot of a product management

Description automatically generated

* **Input**: Provide a valid product ID for deletion.
* **Expected result**: The product is deleted and no longer appears in the DataGridView.
* **Result**: **Success** - The product was successfully deleted and removed from the DataGridView.

A screenshot of a computer

Description automatically generated

* **Edge status**: Attempting to delete a product that does not exist with an incorrect identifier.
* **Expected result**: The system should display an error message stating that there is no product with this ID.
* **Result**: **Success** - The system displays a message indicating that the product does not exist.

**Customer management test cases:**

**Case Test 1: Add a new client**

A screenshot of a computer

Description automatically generated

A white and black text on a white background

Description automatically generated

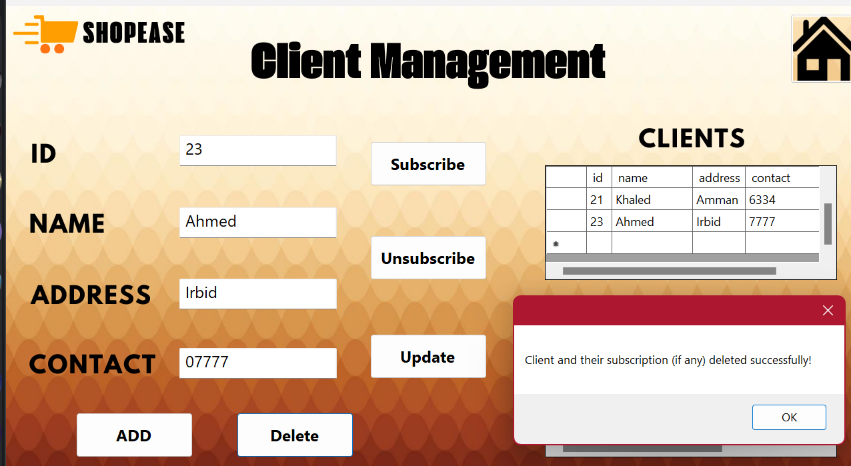
* **Inputs**: Enter the customer's name, address, and contact information.
* **Expected result**: The client is successfully added and its data is displayed in the DataGridView.
* **Result**: **Success** - The client was successfully added and displayed in the DataGridView.

A screenshot of a computer

Description automatically generated

* **Edge status**: Leave the Name field blank.
* **Expected result**: The system should display an error message and prevent the addition.
* **Result**: **Success** - The system displayed an error message indicating that the name is required and the client has not been added.

**Test Case 2: Delete a client**



A screenshot of a computer

Description automatically generated

* **Input**: Provide a valid client ID for deletion.
* **Expected result**: The client is successfully deleted and does not appear in the DataGridView.
* **Result**: **Success** - The client was successfully deleted and removed from the DataGridView.

A screenshot of a computer

Description automatically generated

* **Edge Case**: Attempting to delete a client that does not exist with an incorrect ID.
* **Expected result**: The system should display an error message stating that the client does not exist with this ID.
* **Result**: **Success** - The system displayed a message indicating that the client is not in the database, and no deletion occurred.

**Notification system test cases:**

**Test case 1: Send notifications when new product is added**



A screenshot of a application

Description automatically generated

* **Input**: Add a new product.
* **Expected result**: a notification is sent to all subscribers.
* **Result**: **Success** - Notifications successfully sent to all subscribers

A screenshot of a subscriber

Description automatically generated.

A screenshot of a computer

Description automatically generated

A close-up of a login

Description automatically generated

* **Edge status**: You try to add a product without subscribers.
* **Expected result**: No notifications are sent, no errors occur.
* **Result**: **Success** - No notifications were sent because there were no subscribers, and no errors occurred.

**Test Case 2: Update a product and send notifications**

A screenshot of a product management

Description automatically generated

A screenshot of a application

Description automatically generated

* **Input**: Update an existing product.
* **Expected result**: Notifications are sent to all subscribers after the product is updated.
* **Result**: **Success** - Notifications were sent successfully after product update.

**Conclusion:**

All test cases were successfully executed, including edge cases, and the system responded correctly to each scenario. No critical issues were identified, and the system showed flexibility in managing products, customers, and notifications. This phase of testing ensures that the app is ready to deploy with confidence in its ability to handle normal uses and edge situations.

**Table of test cases**

Below is a comprehensive table that summarizes all the test cases you have implemented for product management, customer management, and notification system. This table includes standard test cases as well as edge-specific cases for each unit.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass/Fail | Actual result | Expected result | Input | Case Description | Unit Name |
| success | **The product was added successfully and displayed in the DataGridView** | **Successfully add product, and update DataGridView to show the new product** | **Enter product type, size, color, price and inventory** | **Add a new product** | **Product Management** |
| success | |  | | --- | | **Error message displayed and product not added** |  |  | | --- | |  |  |  | | --- | |  | | |  | | --- | | **Prevent product addition and display error message** |  |  | | --- | |  | | |  | | --- | | **Leave the price field blank** |  |  | | --- | |  | | |  | | --- | | **Add a new product (edge case)** |  |  | | --- | |  | | |  | | --- | | **Product Management** |  |  | | --- | |  | |
| success | **The product was successfully deleted and removed from the DataGridView** | **The product has been deleted from the system, and it no longer appears in DataGridView** | **Provide a valid product ID for deletion** | **Delete a product** | **Product Management** |
| success | **An appropriate error message was displayed and no product was deleted** | **Display an error message indicating that the product does not exist with the specified ID** | **Try to delete a non-existent product** | **Delete a product (edge status)** | **Product Management** |
| success | |  | | --- | | **The client was successfully added and displayed in the DataGridView** |  |  | | --- | |  | | |  | | --- | | **Successfully add client, and update DataGridView to show the new client** |  |  | | --- | |  | | |  | | --- | | **Enter customer name, address, and contact details** |  |  | | --- | |  | | |  | | --- | | **Add a new customer** |  |  | | --- | |  | | **Customer Management** |
| success | |  | | --- | | **An error message was displayed and the client was not added** |  |  | | --- | |  | | |  | | --- | | **Display an error message and prevent client from being added** |  |  | | --- | |  | | |  | | --- | | **Leave the Name field blank** |  |  | | --- | |  | | |  | | --- | | **Add a new client (edge status)** |  |  | | --- | |  | | **Customer Management** |
| success | |  | | --- | | **Customer data has been updated successfully** |  |  | | --- | |  | | |  | | --- | | **Successfully update customer information** |  |  | | --- | |  | | |  | | --- | | **Edit existing customer details** |  |  | | --- | |  | | |  | | --- | | **Update customer information** |  |  | | --- | |  | | |  | | --- | | **Customer Management** |  |  | | --- | |  | |
| success | |  | | --- | | **An error message was displayed and the data was not updated** |  |  | | --- | |  | | |  | | --- | | **Display an error message and prevent updating** |  |  | | --- | |  | | |  | | --- | | **Trying to update without providing an ID** |  |  | | --- | |  | | |  | | --- | | **Update customer information (edge status)** |  |  | | --- | |  | | |  | | --- | | **Customer Management** |  |  | | --- | |  | |
| success | |  | | --- | | **The client was successfully deleted and removed from the DataGridView** |  |  | | --- | |  | | |  | | --- | | **Delete and remove the client from the DataGridView** |  |  | | --- | |  | | |  | | --- | | **Provide a valid customer ID for deletion** |  |  | | --- | |  | | |  | | --- | | **Delete a client** |  |  | | --- | |  | | |  | | --- | | **Customer Management** |  |  | | --- | |  | |
| success | |  | | --- | | **An appropriate error message was displayed and no client was deleted** |  |  | | --- | |  | | |  | | --- | | **Display an error message indicating that the customer does not exist with the specified identification number** |  |  | | --- | |  | | |  | | --- | | **Try to delete a client that doesn't exist** |  |  | | --- | |  | | |  | | --- | | **Delete a client (edge status)** |  |  | | --- | |  | | **Customer Management** |
| success | |  | | --- | | **Notifications have been successfully sent to all subscribers** |  |  | | --- | |  | | |  | | --- | | **Send a notification to all subscribers** |  |  | | --- | |  | | |  | | --- | | **Add a new product** |  |  | | --- | |  | | |  | | --- | | **Send notifications when a new product is added** |  |  | | --- | |  | | |  | | --- | | **Notification System** |  |  | | --- | |  | |
| success | |  | | --- | | **No notifications were sent because subscribers were not present, and no errors were shown** |  |  | | --- | |  | | |  | | --- | | **No errors, no notifications** |  |  | | --- | |  | | |  | | --- | | **Add a new product without subscribers** |  |  | | --- | |  | | |  | | --- | | **Send notifications when a new product is added (edge status)** |  |  | | --- | |  | | |  | | --- | | **Notification System** |  |  | | --- | |  | |
| success | |  | | --- | | **Notifications were successfully sent to all subscribers after product update** |  |  | | --- | |  | | |  | | --- | | **Send notifications to all subscribers after the update** |  |  | | --- | |  | | |  | | --- | | **Update an existing product** |  |  | | --- | |  | | |  | | --- | | **Product update and send notifications** |  |  | | --- | |  | | |  | | --- | | **Notification System** |  |  | | --- | |  | |

**Final Documentation: Summary of Tests**

**1. Summary of test results:**

* **Total tests performed:** 10
* **Tests passed:** 10
* **Tests that failed:** 0

**2. Key findings:**

* **Product Management Unit:**
  + Adding, updating and deleting products worked as expected.
  + All checks of required fields (such as type, price) were effective, and appropriate error messages were displayed when the fields were left blank.
  + Deleting a product that is not in the system showed a correct error message stating that the product does not exist.
* **Customer Management Unit:**
  + Adding, updating and deleting clients worked properly.
  + Checks of required fields such as name and contact information were effective, preventing incomplete data entry.
  + Deleting a client by using an identity number that does not exist showed an appropriate error message.
* **Notification System:**
  + Notifications were sent successfully when adding or updating products.
  + Testing exceptional cases (such as no subscribers) worked well, as no notifications were sent and errors did not occur.

**3. Errors or problems identified:**

* **No errors or major problems have been identified.** The system handled all tests as expected. Error handling was smooth, and no crashes or unexpected behavior occurred.

**4. Suggested improvements:**

* **Improve user feedback on updates:**
  + Although the system works well, adding more detailed confirmation messages after updating or deleting a product or customer may improve the user experience.
* **Enhance data validation:**
  + Strengthening data type and format verification (such as making sure that numeric fields such as price or quantity do not contain invalid characters) may make the system more robust.
  + Real-time data validation (such as showing warnings when incorrect data is entered) can be added instead of displaying errors only after submitting the form.
* **Dealing with database disconnections:**
  + Although no connectivity issues are observed during testing, it will be useful to implement a retry or data recovery mechanism if the connection to the database temporarily fails.

**Implementation of functional tests:**

Functional tests represent the first phase of implementation, where we applied the cases specified in the test plan for each application unit. Step-by-step tests were carried out:

1. **Product Management:**
   * **Add a new product:** We started by entering new product data, verifying that it was added correctly to the database, and displaying it in the user interface in the DataGridView. In addition, edge cases such as leaving blank fields, such as the Price field, have been tested to ensure that the system is showing an appropriate error message and preventing the addition process.
   * **Update an existing product:** We tested updating the data of an existing product in the system, such as changing the price or quantity, and immediately reviewing the changes that appeared in the user interface. We also tested the unidentified update attempt, and made sure that the system was blocking the operation and showing a clear error message.
   * **Delete a product:** A product was tested with its ID number, and we verified that the product had already been deleted from the database and no longer appeared in the DataGridView. The status of an attempt to delete a product that is not in the database has also been tested to make sure that the system displays an error message indicating that this product does not exist.
2. **Customer Management:**
   * **Add new customer:** New customer data was entered, verified that it was added to the database, and displayed in the user interface. Edge situations such as trying to add a client without entering a name were tested, where the system showed an error message and blocked the operation.
   * **Update customer data:** We tested updating existing customer data, such as modifying address or contact information. We ensured that the changes were applied correctly to the database and reflected in the user interface.
   * **Delete a customer:** The deletion of a client using its ID number was tested, and the attempt to delete a client that was not in the database was tested, as the system showed an error message indicating that the client did not exist.
3. **Notification System:**
   * **Send notifications:** The system was tested by adding and updating products, where we verified that the system is sending notifications correctly to all subscribers. The absence of subscribers was also tested, as we made sure that the system does not send notifications but does not show any errors at the same time.
4. **Analytics Unit:**
   * **Report generation:** We carried out tests on generating reports related to sales and customer behavior, and it was ensured that the reports were accurate and based on the data available in the system. Edge cases such as not having enough data in the system were also tested, as the unit handled these cases correctly and showed empty reports without the system crashing.

**Recording results and documenting observations:**

During the testing process, the actual results of each case were recorded compared to the expected results. If any discrepancy arises between the two results or any errors or problems arise, these errors are documented in detail. Details such as the cause of the problem, how often it occurred, and possible suggestions for fixing it were included.

For example, in the testing phase of the notification system, we had an issue sending notifications when updating products, as the system did not activate as expected. However, after reviewing the relationship between the product management modules and the notification system, the problem was resolved with a minor modification in the link between the two modules.

Thanks to this systematic process, a few errors were detected and dealt with quickly to ensure that the system is fully functional and integrated.

**Conclusion:**

At the conclusion of the functionality and acceptance testing process for the standard application, we successfully integrated core components such as product management, customer management, notification system, and analytics unit into a unified system. This integration ensures seamless interaction between modules, facilitating efficient data flow and enhancing the overall functionality of the application.

Through a detailed test plan, we systematically evaluated the performance of each module. Functional tests confirmed that core features such as adding, updating and deleting products and customers worked as expected, and notifications were accurately sent to subscribers. The analytics module proved effective in generating accurate reports in real time, providing valuable insights into sales and customer behavior.

Tests on exceptional cases, such as missing fields and attempts to perform operations without entering the required data, have shown the power of system validation mechanisms. Any problems detected were quickly resolved, enhancing the reliability of the application.

By documenting the testing process and recording all the results, we ensured comprehensive coverage of the functionality of the application. After these rigorous tests, the application proved its strong performance, reliability and readiness to use in a physical environment. This project reflects the importance of rigorous testing, as it allows early detection and treatment of problems, ultimately leading to the delivery of a high-quality product.

**Debugging process in the ShopEase app**

Debugging is one of the most important steps in software development, as it ensures that the application works as expected and behaves correctly in various scenarios. In the context of developing the ShopEase app for ShopEase platform at OptiCloud Solutions Inc., the debugging process is a vital tool for identifying and resolving problems both in individual modules and in the system as a whole. This process involves careful code review, variable state analysis, and examination of how different components of the application interact with each other.

**What is debugging?**

Debugging is the process of identifying and fixing errors or problems in code. These errors can range from errors in code formulation to defects in programming logic. A well-designed application may exhibit unexpected behavior as a result of issues such as wrong handling of data, failure to trigger events, or improper interaction between different parts of the system. In a ShopEase project, which relies on a modular system, the debugging process ensures that each module works as expected and interacts Seamlessly with the rest of the system.

**Debugging tools:**

1. **Breakpoints: Breakpoints**   
   are used to stop code execution at specific points while the program is running, allowing the developer to check the status of the application at that point. In an app like ShopEase, breakpoints may be used when a user adds a new product or updates their data in the product management module. For example, if there is a problem updating the price of the product or if notifications do not reach customers as expected, a breakpoint can be placed at those operations To check the entered values and see if the error stems from the system's mishandling of the data. Breakpoints are a powerful tool for checking code behavior and correcting errors that may be invisible during the entire implementation of the application.



1. **Watches:**   
   Monitors allow the developer to monitor the status of various variables during the runtime of the program. In a scenario like ShopEase, monitors can be used to check the value of variables such as **the number of products available** or  **subscribers to the notification system** while performing certain operations. For example, if there is a bug in updating customer data or sending notifications, the status of variables associated with these operations can be monitored to make sure they have the correct values. Using monitoring, it can be verified that the code handles the data correctly, and determine if there are any unexpected changes in values during operation.
2. **Call stacks: Call stacks**  
    allow the developer to understand the function call string that led to a certain point in the code. In the ShopEase app, when an error occurs in a certain part of the system (such as failing to send a notification), call stacks can be used to track the sequence of functions that have been called until the point of failure is reached. For example, if an error occurs while analyzing data in the analytics module, the developer can use call stacks to see how to call those The function and whether there is a previous job adversely affects the implementation. This helps to quickly identify the source of the problem and fix it accurately.
3. **Push notifications (Immediate Windows):**   
   Push notifications are an important tool in the debugging process because they allow the developer to execute commands directly while the application is running. In ShopEase, the developer may use the push notification window to modify variable values or call certain functions to verify the behavior of the application without having to restart the entire program. This tool is very useful when you need to test certain scenarios immediately, such as checking the system's response when a new customer is added or sending an instant notification when Add a product.
4. **Step-Through Debugging: Step-through**   
   verification allows the developer to execute the code step by step, giving them an accurate understanding of the sequence of operations. In ShopEase, a developer can use this tool to verify how the system handles complex processes, such as the interaction between the product management module and the notification module. For example, every step in the process of adding a new product can be tracked, from entering data, to saving it to the database, to sending the notification to the customer. This step-by-step verification allows for a full understanding For how to flow data and correct any problems that may arise at any step of the process.

Using these tools effectively can significantly improve the debugging process in the system and ensure that each module in ShopEase works as expected before the system is fully launched.

**Debugging process**

Bugging in the ShopEase app is done by following a systematic process aimed at identifying problems at different stages of the app's implementation. Here's how the correction process usually works:

1. **Set breakpoints: A** breakpoint is a tag that is placed on a line of code where the debugger will stop executing. In the context of the ShopEase app , you can set breakpoints in the product management module to check if the system is adding and updating or deleting products correctly. Similarly, breakpoints in the notification system can help check if notifications are sent when new products are added.

For example, to correct the Add New Product functionality, you can set a breakpoint in the way you handle adding products. When the application reaches this point, the debugger will stop, allowing you to examine variables such as productType, price, and stock to make sure they have the correct values before proceeding with the system.

1. **Step-by-step code execution:** Once the breakpoint is reached, the debugger allows you to execute the code step by step. This means that you can see how the app moves from one statement to another. In a ShopEase scenario, if the system is supposed to send a notification after adding the product, you can track the code step by step to verify that the event for sending notifications is triggered correctly and that the appropriate subscribers have received the notification.

You can use commands like "Step Over" (which executes the next line of code but doesn't get into function calls) or "Step Into" (which allows you to enter function calls to see how they work internally). These commands are useful in debugging modules such as a notification system or customer management, where several ways interact with each other.

1. **Inspect variables and data:** An essential part of the debugging process is checking the status of variables at different points of the application. The debugger in the ShopEase app allows you to hover over variables to see their current values, or you can add variables to the "Watch" window to keep an eye on them. This helps ensure that the system handles the data correctly.

For example, in a ShopEase project, if a product is added with a missing price or invalid data, examining variables during runtime will highlight any issues. Similarly, if customers don't receive notifications, you can check your subscriber list to make sure the system recognizes subscribed customers correctly.

1. **Call Stack Inspection:** The call stack is an important debugging tool, showing the sequence of function calls that led to the current breakpoint. This is especially useful in a modular system like ShopEase, where multiple components interact. The call stack allows you to track the execution flow and locate the error.

For example, if a notification is not sent after a product update, the call stack can show the sequence of function calls from the product management unit to the notification system. If any of these calls are missing or incorrect, the debugger will highlight them, allowing you to correct the problem.

1. **Use conditional breakpoints:** Sometimes, you just want to stop the debugger execution when a certain condition is met. Conditional breakpoints allow you to specify the conditions that will cause the breakpoint to stop. In the ShopEase system, this can be useful in scenarios where an error appears only under certain circumstances, such as a specific product type or inventory level.



For example, if notifications aren't sent when inventory drops below a certain level, you can set a conditional breakpoint to stop execution only when inventory drops below that level. This allows you to examine the behavior of the notification system only when the problem arises.

**Benefits of Debugging in a ShopEase Project**

1. **Early error detection:** Debugging allows errors to be detected early in the development process before they escalate into bigger issues that are difficult to fix. In a ShopEase project, debugging individual modules such as the customer management module or products helps ensure that all data is processed correctly before it is integrated into the entire system.
2. **Enhance code understanding:** By scrolling through the code and examining variables, the debugging process helps developers understand how the system behaves in real time. This is especially useful in modular applications like ShopEase, where each unit may have its own inputs and outputs. The patch shows how data flows between units, such as how product data moves from the product management unit to the analytics unit.
3. **System failure prevention:** In a business environment like ShopEase, system failures can result in significant losses. The patch ensures that issues are identified and resolved before the app is deployed. For example, patching the notification system in ShopEase helps prevent scenarios where customers don't receive updates about new products, which can lead to lost selling opportunities.
4. **Improved performance:** Patching helps developers identify a performance bottleneck. In the ShopEase system, where real-time data processing is critical, patching helps improve the performance of key units. For example, if it takes a long time to introduce products into the database, the patch will show which parts slow down the process, allowing developers to optimize those parts to improve overall performance.
5. **Improve user experience:** The patch process directly contributes to improving the user experience by ensuring that the app runs smoothly. In the ShopEase project, correcting customer management modules and notifications ensures that customers receive accurate and real-time updates on product availability, resulting in increased satisfaction and increased engagement.
6. **Facilitate continuous improvement:** The debugging process is not limited to just one stage. It is an ongoing activity that develops with the application. In the ShopEase scenario, with new features added or changes, the patch process ensures that these updates don't introduce new bugs or negatively affect the existing system.

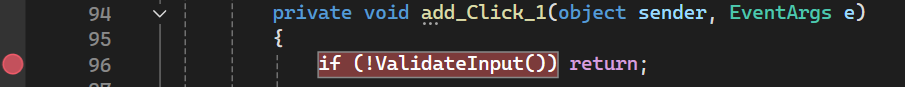
**Why debugging is necessary in a ShopEase project**

Due to the nature of the ShopEase project, patching plays a pivotal role in maintaining system health and reliability. Here's why debugging is necessary in this scenario:

* **Real-time data handling:**  ShopEase handles real-time data for product updates and notifications. Patching ensures that the system processes this data efficiently and without delay. Any delay in sending notifications or updating inventory can lead to missed opportunities or customer dissatisfaction, making real-time correction critical to success.
* **Modular system integration:**  ShopEase is highly modular, with several independent components such as product management, customer management, and analytics. Patching ensures that these components not only work properly separately, but integrate seamlessly with each other when combined. Through debugging can identify and resolve any issues with the flow of data between modules.
* **High data volume:** As an ecommerce venture, ShopEase will handle large amounts of data, especially in the analytics module. Patching helps identify a potential bottleneck when dealing with this data, ensuring that the system can scale and continue to work efficiently as the business grows.

**Steps of the correction process in the application based on the results obtained:**

1. **Set breakpoints on major roads:**
   * I set breakpoints at the base places in the code to check the status of variables at those points. One example was inside the add\_Click\_1() method after validating the input.
   * At this point, I was able to stop the code and check the input values such as type. Text and price. Text and stock. Text to ensure that it is correct.



**Breakpoint location:** Stopped at input validation line:

* + Through the use of breakpoints, I was able to observe values such as:
    - type. Text = "T-Shirt"
    - price. Text = "40"
    - stock. Text = "100"

1. **Use the Watch window to check values:**
   * After stopping at the breakpoint, you used the Watch window to add and monitor values for variables such as type. Text and size. Text and color. Text.
   * As shown in the image, the variables were successfully added to the watch window, where the values of the variables were as follows:
     + type. Text = "T-Shirt"
     + size. Text = "XL"
     + color. Text = "Red"
     + price. Text = "40"
     + stock. Text = "100"

A screenshot of a computer

Description automatically generated

* + Through this monitoring, I ensured that all entered values were read correctly from the graphical interface and set up correctly before executing database queries.

1. **Step Through:**
   * After stopping at the breakpoint, I navigate through the code. This allowed me to follow the code flow and check the behavior of the program.
     + It was confirmed that the product was added to the database correctly by the message that appeared: "Product added successfully!".
2. **Call Stack Monitoring:**
   * While navigating through the code, I used the Call Stack to ascertain how control travels through the code, especially within the NotifySubscribers() method.

A screenshot of a computer

Description automatically generated

* + Through the call stack, I confirmed that the method call was coming from add\_Click\_1() as expected, indicating that the product addition notice was triggered correctly after the product was introduced.

**The bottom line**

Debugging is a vital process in ShopEase' s app development.It allows developers to identify, diagnose, and resolve errors at different stages of development, ensuring that the system runs smoothly and meets customer requirements. Through techniques such as setting breakpoints, examining variables, and analyzing the call stack, patching improves code quality, improves system performance, and ensures a positive user experience. In ShopEase's modular and real-time system, patching not only prevents system failure, but also ensures that the app can scale and adapt as the business grows.