Assignment 1

Software Engineering and Testing

SUBMITTED BY: AMARESH MUDDEBIHAL

REG NUMBER: 20030141IT015

ALLIANCE UNIVERSITY

Assignment 1
Software Engineering and Testing

1. **Imagine that you are in charge of a mid-sized software development project that has a 12- month timeline. The project has a 20-person team consisting of developers, testers, and designers. Requirements gathering, design, development, testing, and deployment are the five stages of the project.**

   Ans:

   As a project manager for an example for such a case, with a medium-sized software development project that would be completed within a duration of 12 months, I would primarily ensure that such a project is always approached in stages: requirements gathering, design, development, testing, and deployment. Since the project has a team of 20 people comprising developers, testers, and designers, it is very important to align them effectively for delivery.

   First two months in the requirement gathering phase so we are going to understand and document all the needs and expectations of the stakeholders. It is this stage where I would facilitate meetings with clients, end-users, and other varied stakeholders to capture functional and non-functional requirements through interviews, surveys, and workshops so no detail goes unexplored. It would lead to detailed requirements specification document, which would then provide the basis for the next steps.

   The design phase, for its part, should take two months after the requirements gathering. In that phase, the team of designers and developers will collaborate in designing the architecture and the user interface of the system. The designers work on wireframes and prototypes to make the user experience as intuitive as possible, but the developers would design system architecture, databases, and technology selection during this phase. This phase would have design documentation-the system architecture and UI/UX designs. This documentation would help the development team in the succeeding phase.

   The development phase would last for four months after the design phase. This period would see the development team start coding the application according to the compiled requirements. The developers would do the work in sprints following Agile methodologies so that there was incremental delivery, hence having many chances for feedbacks and improvements. It would make use of version control and hold code reviews to maintain a high-quality code. This is the critical phase of the project, because the functionality of the software will be build and tested iteratively .

   After commencing the development work, the testing phase would initiate at the 9th month of which the testing procedure would run up to the 10th month. The different levels of testing that could be done include unit, integration, and system testing so that it finds no significant bugs and matches with the requirement, respectively. They would put out very elaborate test cases around those requirements and be working with developers to make any necessary fixes that present themselves during the testing phase of the software. This last stage would guarantee stability and dependability of the end product before release.

Finally, deployment would be in month 11, and the project would be entering into its final month of development. The system would, at this point, be released to a production environment. This would involve server configuration, UAT testing, and ensuring all criteria are met before the software goes live. In addition to this, post-deployment support will be planned-that is, training the users and getting documentation ready for maintaining the software over the long term. The next stage is the deployment stage, but the project manager must stay in close contact with the system to monitor its performance once the users start interacting with it.

My role during the entire process would be to monitor progress, determine any issues that might arise in the course of the process, and keep the project within scope and budget. I would need to keep stakeholders informed and engaged throughout the 12 months.

2. **Consider yourself assigned to examine a public library's Library Management System (LMS). Activities like catalog administration, member registration, book borrowing and return, and so on should be handled by the system.**

Ans: As a system analyst, the assigned task is to analyze and improve an LMS in a public library. The central aim is to analyze its workability, evident flaws, and ensure that the LMS provides support to all major activities, including catalog management, member registration, borrowing, and returning books, along with many other activities. In short, to be operated without any inconvenience with optimum efficiency, an LMS plays a highly significant role in giving the user hassle-free experience in the hands of the library staff while ensuring that books may be managed and activities of the members may be maintained track.

Catalog Management

The management of the library's catalog should be one of the primary roles of an LMS. It must be possible, for example, to easily input new books, update existing records for books, and delete aged or damaged books. This would especially include updating the key attributes such as book title, author, ISBN, genre, and year of publication as well as indicate if they are available, checked out, or reserved. Books should also be categorized in the LMS to enable easy searching so that both staff and members can easily find books. Features like barcode scanning or integration with RFID technology would help the process of cataloging and thus make it more accurate.

Member Registration and Management

The LMS should have provision for individual registration as members of the library. This will include name, address, contact details, and type of library membership that is adult, child, senior, etc. In addition, the system should allow management of member records in hand by staff to include updates on contact information and tracking overdue items or fines. The most crucial feature is the member authentication where the members can borrow or return books using the member ID or a barcode. Furthermore, the system should allow the user to have a view of his borrowing history, check what is borrowed, and monitor the fines in case of delay for returning books.

Book Lending and Return

Book loan is the center of any LMS. If a member borrows a book, then the transaction has to be recorded by the system with member ID, book ID, date of borrowing, and date by which the book should be returned to the library. It must also allow extensions and renewals of borrowed books, as long as it is not reserved by other members. When the books have already been returned, the system must automatically update the availability status of the books. Further, in case the books are overdue, the system will automatically calculate the late charges based on the number of days passed. To speed up the borrowing and returning process, the system may use barcode scanning or RFID facilities.

Book Reservations and Hold Management

Beyond mere borrowing and returning, the LMS should also facilitate members reserving (or holding) books currently borrowed by other members. The system should then alert the member that such book is now in place for retrieval. Managing the reservation of books will ensure that the same resources in the library are shared equitably, with least waiting time as possible, by members for popular books. The process helps the librarians monitor the stock even-better, knowing which books are the most in demand.

Reporting and Analytics

The LMS should have several reporting tools for library administrators in order to make the management of the library more efficient. Such reports should include some statistics on book circulation, member activity, items returned late, and financial reports, such as total fines collected. In any case, analytics should be used to identify the most popular genres or authors and guide purchasing and reduce the collection to the best extent possible in order to serve the community effectively.

Access and User Roles in the LMS

The LMS must be multiuser with differing permission level accesses. For example, the LMS ought to have the following user roles:

Library Staff: They handle admin functions of the catalogue, member sign up, book issuance, reservation of books, and collection of fines.

Members: They should be able to browse the books, see borrowing history, reserve books and pay fines.

Admin: Should have absolute access rights for all features of the system including report details, managing users and configuration of the system.

The system should also be user-friendly and enable easy navigation for the employees as well as the members. For the employees, it would be very productive with some dashboard view having easy access to key tasks such as loan issuance, checking availability of books, processing returns, etc. For the members, the interface must be clean and intuitive to search the catalog, to manage their borrowing activities, and to make payments.

Security and Data Protection

Security is the most important aspect in a library management system since it holds sensitive member data. It needs to have deep authentication mechanisms so that only authorized personnel can carry out this critical administrative operation, such as the insertion or deletion of entry in the catalog. Member data should also be kept private; therefore, protection to any personal data must be given based on privacy laws and regulations.

Interconnection with other systems

Integration with other systems to make the LMS work more efficiently: For example, integration with payment systems for the payment of fines, overdue fees, or even membership fees allows for hassle-free transactions for the member. It might also connect to an external book database, such as an ISBN database, in order to automatically pull up the book details, which will then save staff hours when adding new books to the catalogue.

Library Management System Evaluation Examines the capability of the system in managing key activities about the catalogue, registration of members, borrowing and returning books, reservations, and reporting. LMS is important for running an efficient library: it provides the necessary tools for staff to manage the collection of the library to ensure that members can easily get hold of the books and resources required. More features that could be added might include bar-code scanning and integration with other systems. Additionally, the LMS could offer more robust reporting, which might also benefit the staff and members in using the system.

3. **Create a use case model for a food delivery business that operates online. List the actors engaged in each of the at least five main use cases. Give a thorough use case statement for every use case.**

Ans:
Use Case Model for an Online Food Delivery Business

An online food delivery business involves various actors and processes to ensure smooth operations, from order placement to delivery. The system should facilitate interactions between customers, restaurants, delivery agents, and administrators. Below is a use case model for the food delivery system, detailing at least five main use cases and the actors involved in each case.

1. Place Order

The **Customer** is the primary actor involved in the "Place Order" use case. The customer initiates the process by browsing the food menu, selecting the desired items, and then proceeding to checkout. This action involves choosing the delivery address, reviewing the order details, and making the payment. The system calculates the total price, including any applicable taxes, delivery fees, and discounts, and allows the customer to confirm the order. Once confirmed, the system sends the order information to the **Restaurant**, which begins processing the order.

Use Case Statement:

- **Title:** Place Order
- **Primary Actor:** Customer
- **Goal:** To place an order for food from a restaurant.
- **Preconditions:** The customer must be logged into the system.
- **Main Flow:**
    1. Customer browses the restaurant menu.
    2. Customer adds selected items to the cart.
    3. Customer provides delivery details.
    4. System calculates total cost and applies discounts.
    5. Customer confirms the order and makes the payment.
    6. The system sends the order details to the restaurant for confirmation.
- **Postconditions:** The order is confirmed by the restaurant and queued for preparation.

2. Confirm Order

The main actor involved in the "Confirm Order" use case is the Restaurant. Based on the order received from the customer, the restaurant checks on its inventory and confirms the availability of the items ordered. In case any of the items is not available, the restaurant communicates with the customer on the issue of unavailability of the item. Then when all items are confirmed available, the restaurant confirms the order. In this manner, the order gets prepared and is dispatched subsequently.

Use Case Statement:

- **Title:** Confirm Order
- **Primary Actor:** Restaurant
- **Goal:** To confirm the customer's order and notify them if any items are unavailable.
- **Preconditions:** The restaurant has received the order from the system.
- **Main Flow:**
    1. Restaurant reviews the order details.
    2. The system checks the availability of each item.
    3. If any item is unavailable, the restaurant informs the customer and suggests alternatives.

4. If all items are available, the restaurant confirms the order.
5. The system updates the order status to "confirmed."

- **Postconditions:** The order is confirmed, and preparation begins.

3. Track Order

Track Order" is the primary actor in the use case. Using the system, with the help of it, order status can be tracked online after the order has been placed. Whether food is being prepared or it is already ready for delivery, or even out for delivery, all sort of information about the status is delivered with the help of the system. Either the system initiates notifications or allows the customer to check on the status of the order himself. The administrator could also be required to keep tabs on the status of orders and resolve any problems that result.

Use Case Statement:

- **Title:** Track Order
- **Primary Actor:** Customer
- **Goal:** To track the real-time status of an order.
- **Preconditions:** The order has been confirmed by the restaurant.
- **Main Flow:**
    1. Customer logs into the system to view order status.
    2. The system displays the current status of the order (e.g., being prepared, ready for dispatch, out for delivery).
    3. The system provides updates on estimated delivery time.
- **Postconditions:** The customer is informed about the status of their order.

4. Deliver Order

Track Order" is the primary actor in the use case. Using the system, with the help of it, order status can be tracked online after the order has been placed. Whether food is being prepared or it is already ready for delivery, or even out for delivery, all sort of information about the status is delivered with the help of the system. Either the system initiates notifications or allows the customer to check on the status of the order himself. The administrator could also be required to keep tabs on the status of orders and resolve any problems that result.

Use Case Statement:

- **Title:** Deliver Order
- **Primary Actor:** Delivery Agent
- **Goal:** To deliver the food to the customer’s address.
- **Preconditions:** The order is confirmed and prepared by the restaurant.

- **Main Flow:**
    1. The delivery agent receives the order details and picks up the food from the restaurant.
    2. The agent marks the order as "out for delivery" in the system.
    3. The delivery agent navigates to the customer’s address.
    4. The delivery agent delivers the food and updates the order status to "completed."
- **Postconditions:** The order is successfully delivered, and the system updates the status to "completed."

5. Manage System

Such an actor mainly carries out the use case "Manage System." These food delivery platform administration conducts this run by retrieving menus from restaurants, processing customer accounts, and placing orders. They provide them with an addition of new restaurants, updated menus, updated customer accounts, process payments, and retrieve reports. Such an admin can track the movement of orders and make sure that delivery agents are also taking their work seriously.

Use Case Statement:

- **Title:** Manage System
- **Primary Actor:** Admin
- **Goal:** To manage and oversee the entire food delivery system.
- **Preconditions:** The admin must be logged into the system.
- **Main Flow:**
    1. The admin adds or updates restaurant details and menus.
    2. The admin manages customer accounts, such as suspending fraudulent accounts or resolving disputes.
    3. The admin monitors order status and intervenes if there are issues with any particular order.
    4. The admin generates reports for financial tracking, customer activity, and system performance.
- **Postconditions:** The system reflects all updates and changes made by the admin, and the platform continues to run smoothly.

This use case model reflects the strategic functions on which the food delivery platform focuses, that is, key interactions between customers, restaurants, delivery agents, and admins. It can be divided into use cases as follows: make an order, confirm order, track status of order, deliver food, and finally, system management. This means defining use cases and actors of the systems, thus structuring the system in such a way that all tasks are taken care of efficiently with respect to both customers and service providers hence making it a seamless and enjoyable experience.

4. **Create a class diagram for a course management system that can be used by academics to establish courses, students to enroll in them, and administrative personnel to oversee enrollments. Add methods, attributes, and connections between the classes.**

Ans :

This class diagram depicts a Course management system.

A Course Management System provides the functions for course creation as well as the management of courses, student enrollments, and administrative oversight. Classes for the system exist under the names of Course, Student, Academic, and Administrator; each class performs separate functionalities. A description of the class diagram detailing the classes, their attributes, methods, and relationships follows.

1. Class: Course

The Course class represents a course that is taught by an academic institution. The class holds details such as name for the course, code for the course, academic offering for the course, students enrolled in it, and the schedule. The Course class supports creating new courses, modifying existing ones, and deleting courses. It also keeps track of the maximum number of students who can enroll into the course, and student records of the ones enrolled.

Attributes:

- courseCode (String): A unique identifier for the course.
- courseName (String): The name of the course.
- courseDescription (String): A brief description of what the course entails.
- maxEnrollment (Integer): The maximum number of students allowed in the course.
- schedule (String): The time and days the course is held.
- academic (Academic): The academic responsible for teaching the course.
- students (List<Student>): A list of students currently enrolled in the course.

Methods:

- addStudent(student: Student): Adds a student to the course if the maximum enrollment limit is not exceeded.
- removeStudent(student: Student): Removes a student from the course.
- updateCourseDetails(): Updates course details such as schedule or description.

- getEnrollmentStatus(): Returns the number of students enrolled and the maximum capacity of the course.

2. Class: Student

The **Student** class represents a student who can enroll in courses offered by the academic institution. It contains details about the student, including personal information and a list of courses they are enrolled in. Students can enroll in or drop courses, and the system keeps track of their enrollment history.

Attributes:

- studentID (String): A unique identifier for the student.
- studentName (String): The name of the student.
- email (String): The student's contact email.
- enrolledCourses (List<Course>): A list of courses the student is currently enrolled in.
- completedCourses (List<Course>): A list of courses the student has completed.

Methods:

- enrollInCourse(course: Course): Enrolls the student in a course, provided they meet the prerequisites and the course has available spots.
- dropCourse(course: Course): Removes the student from a course they are enrolled in.
- getEnrolledCourses(): Returns a list of courses the student is currently enrolled in.
- getCompletedCourses(): Returns a list of completed courses.

3. Class: Academic

The **Academic** class represents the academic or instructor who creates, manages, and teaches courses. Academics are responsible for setting course schedules, creating course materials, and ensuring that students meet the course's learning objectives. This class also includes methods for viewing student enrollments and assigning grades.

Attributes:

- academicID (String): A unique identifier for the academic.
- academicName (String): The name of the academic.
- coursesTaught (List<Course>): A list of courses taught by the academic.
- email (String): The academic's contact email.

Methods:

- createCourse(course: Course): Creates a new course and adds it to the system.

- updateCourseDetails(course: Course): Modifies course information such as schedule or course materials.
- viewStudentEnrollments(course: Course): Views the list of students currently enrolled in a specific course.
- assignGrades(course: Course, student: Student, grade: String): Assigns grades to a student for a particular course.

4. Class: Administrator

The **Administrator** class represents the administrative personnel responsible for overseeing the overall functioning of the course management system. The administrator has the ability to manage all aspects of the system, including adding or removing students, enrolling them in courses, viewing course enrollments, and generating reports.

Attributes:

- adminID (String): A unique identifier for the administrator.
- adminName (String): The name of the administrator.
- email (String): The administrator's contact email.

Methods:

- addStudent(student: Student): Adds a new student to the system.
- removeStudent(student: Student): Removes a student from the system.
- enrollStudentInCourse(student: Student, course: Course): Enrolls a student in a course.
- generateReport(): Generates a report summarizing enrollment, course offerings, or academic performance.

Relationships Between Classes

- **Student ↔ Course (Many-to-Many Relationship):** A student can enroll in many courses, and each course can have many students enrolled. This is represented by a list of **students** in the **Course** class and a list of **enrolledCourses** in the **Student** class.
- **Academic ↔ Course (One-to-Many Relationship):** An academic teaches multiple courses, but each course is taught by only one academic. This is represented by an **academic** attribute in the **Course** class, which links back to the **Academic** class.
- **Administrator ↔ Student (One-to-Many Relationship):** The administrator has the ability to manage multiple students. This relationship allows the administrator to add, remove, or modify student information in the system.
- **Administrator ↔ Course (One-to-Many Relationship):** The administrator can oversee multiple courses, including creating new ones or modifying existing ones.

The Course Management System's class diagram includes key classes such as **Course**, **Student**, **Academic**, and **Administrator**, each with their own attributes and methods that define the functionality of the system. The relationships between the classes ensure that students can enroll in courses, academics can manage their courses, and administrators oversee the system's operation. By defining these classes and their interactions, the CMS can effectively manage course creation, student enrollments, and academic oversight.

5. **Make a sequence diagram for a customer relationship management (CRM) system that illustrates the exchanges that take place when a customer care agent fixes a customer's problem. Add all pertinent system elements and their relationships.**

Ans: **Sequence Diagram for a CRM System: Fixing a Customer's Problem**

A Customer Relationship Management (CRM) system helps businesses interact with customers and manage their queries, feedback, and issues. Below is a sequence diagram that illustrates how the system handles the interaction when a customer care agent addresses a customer's problem. The diagram outlines the flow of interactions between the system elements, such as the **Customer**, **CRM System**, **Customer Care Agent**, and **Database**.

Actors and System Elements:

1. **Customer**: The person experiencing an issue with a product or service.
2. **CRM System**: The platform used by the customer care agent to track, resolve, and manage customer issues.
3. **Customer Care Agent**: The person who interacts with the customer to resolve their problem using the CRM system.
4. **Database**: Stores customer data, problem tickets, interaction logs, and issue resolutions.

Sequence Diagram Steps:

1. **Customer Raises an Issue**
    - The **Customer** initiates the process by calling customer service, visiting the company's help center, or submitting a support ticket online.
    - The **CRM System** receives the issue request (such as a ticket or call).

Interaction:

- **Customer → CRM System**: "Submit Issue" or "Log Ticket."

2. **CRM System Creates a Ticket**

- The **CRM System** creates a unique issue ticket for the customer's problem.
- This ticket is assigned to a **Customer Care Agent** based on availability or expertise.

Interaction:

- **CRM System → Database**: "Create Ticket" (stores the ticket details).
- **CRM System → Customer Care Agent**: "Assign Ticket."

3. **Customer Care Agent Reviews the Ticket**
    - The **Customer Care Agent** reviews the ticket information, such as the issue details, previous interactions, and customer history, to better understand the problem.

Interaction:

- **Customer Care Agent → CRM System**: "Retrieve Ticket Information."

4. **Agent Communicates with the Customer**
    - The **Customer Care Agent** contacts the customer, either via phone, chat, or email, to discuss the issue and gather more details.
    - If necessary, the agent may ask for clarification on the issue.

Interaction:

- **Customer Care Agent ↔ Customer**: "Request More Information" (communication through chat or call).

5. **Problem Diagnosis and Solution**
    - Based on the information gathered, the **Customer Care Agent** diagnoses the issue.
    - The agent checks the **CRM System** for any known solutions or troubleshooting guides that may help resolve the issue.
    - If the issue requires a technical fix or approval, the **Customer Care Agent** might consult with other departments or escalate the problem to a higher tier.

Interaction:

- **Customer Care Agent → CRM System**: "Search Knowledge Base" (to find solutions).
- **CRM System → Database**: "Fetch Solution" (fetching potential solutions).
- **Customer Care Agent ↔ Customer**: "Provide Solution" (guiding the customer through the resolution process).

6. **Resolution Confirmed**

- Once the issue is resolved or a solution is provided (such as troubleshooting steps, a refund, or a replacement), the **Customer Care Agent** confirms the solution with the customer.
- The agent marks the ticket as resolved in the CRM system.

Interaction:

- **Customer Care Agent ↔ Customer**: "Confirm Resolution."
- **Customer Care Agent → CRM System**: "Mark Ticket as Resolved."
- **CRM System → Database**: "Update Ticket Status" (change ticket status to "Closed" or "Resolved").

7. **Post-Resolution Feedback**

- After the resolution, the **CRM System** may prompt the **Customer** to provide feedback on the resolution quality or agent performance.
- The feedback is collected and stored for future improvements.

Interaction:

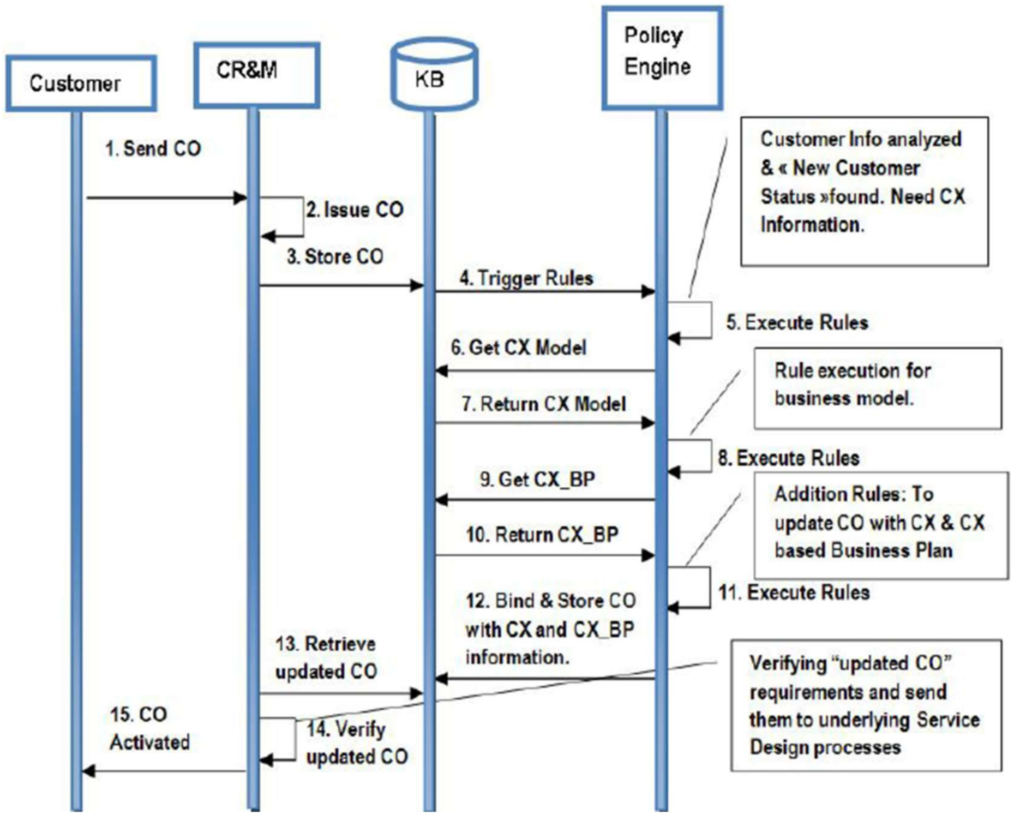
- **CRM System → Customer**: "Request Feedback."
- **Customer → CRM System**: "Provide Feedback."

8. **Ticket Closure**

- The **CRM System** ensures that the ticket is closed and archives the interaction details for future reference.

Interaction:

- **CRM System → Database**: "Archive Ticket" (store details of the interaction).


Customer
CR&M
KB
Policy Engine
1. Send CO
2. Issue CO
3. Store CO
4. Trigger Rules
5. Execute Rules
6. Get CX Model
7. Return CX Model
8. Execute Rules
9. Get CX_BP
10. Return CX_BP
11. Execute Rules
12. Bind & Store CO with CX and CX_BP information.
13. Retrieve updated CO
14. Verify updated CO
15. CO Activated
Customer Info analyzed & « New Customer Status »found. Need CX Information.
Rule execution for business model.
Addition Rules: To update CO with CX & CX based Business Plan
Verifying "updated CO" requirements and send them to underlying Service Design processes

9.

Key Interactions:

1. **Customer → CRM System**: Submitting the issue or request.
2. **CRM System → Database**: Creating and storing the ticket details.
3. **CRM System → Customer Care Agent**: Assigning the ticket to an agent.
4. **Customer Care Agent → CRM System**: Retrieving ticket information and solutions.
5. **Customer Care Agent ↔ Customer**: Communicating with the customer to gather more details or provide a solution.
6. **CRM System → Customer**: Requesting feedback after the issue is resolved.

The sequence diagram illustrates the step-by-step process of resolving a customer issue through a CRM system. It highlights the interactions between the **Customer**, **CRM System**, **Customer Care Agent**, and **Database**, from the moment a problem is logged until the issue is resolved and feedback is gathered. This process ensures efficient issue resolution and customer satisfaction.