Lab-4 : Write down the ATM system specifications and report the various bugs.

The ATM system is designed to perform various banking operations such as cash withdrawal, deposit, balance inquiry, and fund transfers. Below are the key specifications for an ATM system:

---

1. User Authentication

- **Card Authentication**: Users insert their ATM card (debit/credit) into the machine.
- **PIN Verification**: Users must input their PIN (Personal Identification Number) to gain access to their account.
- **Account Lock**: After three consecutive incorrect PIN entries, the system locks the user's account.

2. Main Operations

- **Cash Withdrawal:**
  - Users can withdraw money within their account balance limits.
  - The system checks for sufficient funds and available ATM cash.
  - Daily withdrawal limit enforced (e.g., $500 per day).
- **Cash Deposit:**
  - Users can deposit cash into their account.
  - The system should update the account balance after the deposit.
- **Balance Inquiry:**
  - Users can check their current account balance.
  - The system displays available and total balance.
- **Fund Transfer:**
  - Users can transfer money between accounts within the same bank or to accounts in other banks.
  - Necessary checks are performed (e.g., sufficient balance, recipient details).
- **Mini Statement:**
  - Users can view the last 10 transactions from their account.
- **PIN Change:**
  - Users can change their ATM PIN after correctly entering their current PIN.

3. User Interface

- Multi-Language Support: Users can select a preferred language.
- Touch Screen / Button-based Input: Depending on the hardware, the interface can support both.
- Receipts: Users can opt for a printed receipt after every transaction.

4. Security Features

- Encryption: All transactions and user details are encrypted.
- Session Timeout: If the user is inactive for a specified period, the session is terminated.
- Physical Security: Sensors and alarms are triggered in case of any unauthorized access or tampering.
- Camera Monitoring: The ATM is monitored by cameras for security.

5. Transaction Limits

- Daily Withdrawal and Deposit Limits: Each user has a daily transaction limit (e.g., maximum $500 withdrawal per day).
- Transfer Limit: Limits may be set for funds transfer depending on the account type (e.g., $1,000 per transfer).

6. Error Handling

- Insufficient Funds: A message is displayed when the account balance is too low to complete a withdrawal or transfer.
- Out of Service: The ATM displays an error if it runs out of cash or encounters a technical fault.
- Card Retention: After three incorrect PIN entries or card-related issues, the ATM may retain the card.
- Receipt Print Failure: If the printer fails to print a receipt, the system notifies the user.

Reported Bugs in ATM System:

1. Incorrect Balance After Withdrawal

- Bug Description: Sometimes, after a withdrawal transaction, the user's account balance does not update correctly, showing the wrong amount on the screen or receipt.
- Impact: Can cause users to believe they have more or less money than they actually do.
- Possible Cause: Delayed database update or transaction rollback failure.

2. Transaction Timeout Without Reverting Balance

- Bug Description: If a transaction times out during a fund transfer or withdrawal, the user's account is debited, but the transaction is incomplete (e.g., no cash is dispensed).
- Impact: Users lose money temporarily until the bank manually reverses the transaction.
- Possible Cause: Poor handling of transaction rollback on timeout.

3. Multiple Receipts Printed for One Transaction

- Bug Description: After completing a transaction, the ATM prints multiple copies of the receipt instead of one.
- Impact: Waste of paper and user confusion.
- Possible Cause: Faulty printer module or software issue sending multiple print commands.

4. Card Retention After Successful PIN Entry

- Bug Description: Some users report that their ATM card is retained even after entering the correct PIN.
- Impact: User inconvenience, and potential loss of access to the ATM card.
- Possible Cause: Faulty hardware or software logic related to the card reader.

5. Language Switching Issues

- Bug Description: After selecting a preferred language, the ATM sometimes reverts to the default language in the middle of a transaction.
- Impact: Confusion for non-native speakers, leading to incorrect inputs or cancellations.
- Possible Cause: Session state issue or improper handling of language preferences in the interface.

6. Incorrect Mini-Statement Display

- Bug Description: The mini statement feature shows inaccurate or outdated transactions.
- Impact: Users are unable to track recent activity accurately.
- Possible Cause: Cache issues or delay in syncing transaction history from the main server.

7. Fund Transfer Incorrectly Denied

- Bug Description: Even when users have sufficient balance, some fund transfer requests are denied with an "insufficient funds" error.
- Impact: Inability to complete a valid transfer.

- Possible Cause: Calculation bug when accounting for fees or minimum balance requirements.

8. Failure to Dispense Cash but Still Debits Account

- Bug Description: ATM sometimes debits the user's account without dispensing cash during a withdrawal transaction.
- Impact: Temporary loss of funds, leading to user frustration.
- Possible Cause: Mechanical issue with cash dispenser or failure to verify cash dispensation before debiting the account.

9. System Freezes After PIN Entry

- Bug Description: Some users report that the ATM screen freezes after they input their PIN, requiring a restart or card removal.
- Impact: User inconvenience, especially if the session times out or the card is retained.
- Possible Cause: Software glitch or insufficient system resources.

10. Session Timeout Error During Long Transactions

- Bug Description: The ATM session times out too quickly, even when the user is in the middle of completing a transaction like a fund transfer.
- Impact: Transaction interruption, loss of progress, and user frustration.
- Possible Cause: Inadequate timeout settings or failure to detect ongoing user activity.

Lab-9: Explain the role of software engineering in Biomedical Engineering and in the field of Artificial Intelligence and Robotics.

Role of Software Engineering in Biomedical Engineering:

Software engineering plays a crucial role in **Biomedical Engineering** by developing and maintaining the software systems that power medical devices, diagnostic tools, and healthcare management platforms. It enables innovation in the healthcare field by facilitating data analysis, medical imaging, and the management of patient information. The integration of software into biomedical devices enhances precision, efficiency, and scalability.

Key Roles:

1. **Medical Device Software Development**:
    - Software engineers design and implement the embedded systems in medical devices such as pacemakers, insulin pumps, MRI scanners, and robotic surgical systems. These devices require reliable, real-time software for monitoring, diagnosis, and treatment.
    - Compliance with medical standards (e.g., ISO 13485, FDA regulations) is essential to ensure safety and reliability in software used for life-critical systems.
2. **Medical Imaging and Diagnostics**:
    - Software is essential in the acquisition, processing, and interpretation of medical images (e.g., MRI, CT scans, ultrasound). Engineers develop algorithms to enhance image clarity, detect abnormalities, and assist in diagnosis.
    - AI is often incorporated into imaging software for automating the detection of tumors, fractures, or other anomalies, improving the accuracy and speed of diagnosis.
3. **Health Informatics**:
    - Software engineering is responsible for creating Electronic Health Record (EHR) systems, data management platforms, and patient monitoring systems. These systems store and manage large volumes of sensitive medical data and support decision-making in hospitals and clinics.
    - Software must adhere to privacy laws and data security standards such as HIPAA, ensuring that sensitive medical data is protected.
4. **Bioinformatics**:
    - Software engineers contribute to bioinformatics by developing tools that analyze biological data, such as DNA sequences, protein structures, and genetic information. These systems assist in personalized medicine, drug discovery, and genomics research.

    - Algorithms and machine learning models developed by software engineers are used to interpret complex biological datasets.

5. **Wearable and Telemedicine Solutions**:
    - Software for wearable health devices (e.g., fitness trackers, heart rate monitors) allows patients to monitor their health in real-time, while telemedicine platforms enable remote consultations and diagnostics, particularly during times of limited in-person access.

Example Projects:

- Development of software for robotic-assisted surgery.
- Creating data analysis tools for genomic research.
- Software controlling real-time monitoring of patient vitals.

Role of Software Engineering in Artificial Intelligence (AI) and Robotics:

In **Artificial Intelligence (AI)** and **Robotics**, software engineering is fundamental to designing, developing, and deploying intelligent systems that can automate tasks, learn from data, and interact with the physical world. It enables robots and AI systems to operate with precision, adapt to environments, and solve complex tasks efficiently.

Key Roles in AI:

1. **Algorithm Development**:
    - Software engineers create and implement machine learning (ML) and deep learning (DL) algorithms that power AI applications. These algorithms allow systems to analyze data, recognize patterns, make predictions, and automate decision-making processes.
    - Engineers optimize AI models for scalability, efficiency, and real-world application.
2. **Data Management and Processing**:
    - In AI systems, vast amounts of data are collected and processed for model training and inference. Software engineering ensures that these data pipelines are efficient, secure, and capable of handling large-scale datasets.
    - Engineers design systems that manage data preprocessing, feature extraction, and data augmentation.
3. **Natural Language Processing (NLP)**:

   - In AI applications like chatbots, voice assistants, and translation systems, software engineers build the NLP systems that enable machines to understand and generate human language.
   - Engineers create APIs and develop frameworks that integrate NLP capabilities into larger software ecosystems.
4. **AI in Healthcare, Finance, and Other Industries**:
   - AI applications in various fields (healthcare, finance, marketing) rely heavily on software engineering for deployment and scaling. In healthcare, AI is used for diagnostics, drug discovery, and predictive analytics. In finance, it powers fraud detection and algorithmic trading.
   - Software engineers build the platforms that deliver AI capabilities to end-users, ensuring they are user-friendly, secure, and maintainable.

Key Roles in Robotics:

1. **Control Systems**:
   - In robotics, software engineers develop the control systems that guide a robot's movement, decision-making, and interaction with the environment. This involves writing software for real-time operation, sensor integration, and actuator control.
   - Engineers design algorithms for motion planning, pathfinding, and obstacle avoidance, allowing robots to perform tasks autonomously or semi-autonomously.
2. **Robot Operating System (ROS) and Middleware**:
   - Many robots run on frameworks like ROS (Robot Operating System), which provides libraries and tools for developing robotic applications. Software engineers design and maintain the middleware that enables communication between robot components (sensors, controllers, actuators).
   - Engineers contribute to the development of simulation environments where robots can be tested virtually before real-world deployment.
3. **Autonomous Systems**:
   - In autonomous vehicles, drones, and industrial robots, software engineers develop systems that allow these machines to perceive their environment, make decisions, and navigate independently. This involves integrating AI techniques such as computer vision and reinforcement learning.
   - Engineers design the architecture for sensor fusion, enabling the robot to process inputs from multiple sensors (e.g., cameras, LIDAR, GPS) and react appropriately to the environment.
4. **Human-Robot Interaction (HRI)**:

    - Robotics systems that interact with humans (e.g., service robots, collaborative industrial robots) require user-friendly interfaces and safety features. Software engineers develop HRI systems to ensure that robots can communicate effectively and operate safely in human environments.

Example Projects:

- Developing autonomous vehicle navigation software.
- Building a robotic arm control system for precise manufacturing tasks.
- Implementing AI models for visual perception and object recognition in robots.

Conclusion:

Software engineering plays a foundational role in **Biomedical Engineering**, enabling the creation of advanced medical technologies that improve patient care and diagnosis. In **AI and Robotics**, software engineers are at the forefront of designing intelligent systems and autonomous machines that push the boundaries of automation, adaptability, and human-robot interaction. Both fields rely heavily on robust, efficient, and secure software to drive innovation and ensure safety and reliability in real-world applications.

10. Study the various phases of Water-fall model. Which phase is the most dominated one?

The Waterfall Model is a linear and sequential software development methodology that divides the project into distinct phases.

Each phase must be completed before moving on to the next, and there is no overlap between phases. The phases are as follows:

1. Requirement Gathering and Analysis:

In this phase, all potential system requirements are gathered from the client and documented.

This phase involves analyzing and understanding the needs and setting the system specifications.

Output: A requirement specification document.

2. System Design:

Based on the requirements, the system architecture and design are created.

This includes the high-level design (HLD) of the overall system and the low-level design (LLD) for the individual components.

Output: System design documents and architecture diagrams.

3. Implementation (Coding):

The actual coding and development of the system take place in this phase. Developers write the code based on the design documents.

Output: Functional software modules.

4. Integration and Testing:

After coding, the system undergoes various levels of testing to detect defects and verify that it meets the requirements.

This phase includes unit testing, integration testing, system testing, and acceptance testing.

Output: Tested software.

5. Deployment:

Once testing is complete and the system is deemed stable, the software is deployed to the production environment,

where it becomes operational for users.

Output: Deployed software.

6. Maintenance:

After deployment, the system enters the maintenance phase, where it is monitored for any issues.

Bugs are fixed, updates are applied, and new features may be added over time.

Output: Updated and maintained system.

Most Dominant Phase:

Requirement Gathering and Analysis is considered the most dominant phase in the Waterfall model.

This phase is critical because errors or incomplete information at this stage can lead to project failure.

The success of all subsequent phases depends on having accurate, complete, and well-understood requirements.

If mistakes are made here, they can be very costly to fix later in the process.

Lab-11

The **COCOMO (Constructive Cost Model)** is a software engineering model that estimates the effort, time, and cost required to complete a software development project based on its size, complexity, and other factors. There are several versions of COCOMO, but the most commonly used is **COCOMO II**.

Steps to Estimate Effort Using the COCOMO Model:

1. **Determine the Size of the Project (KLOC)**:
    - The first step is to estimate the size of the software project in terms of **Kilo Lines of Code (KLOC)**. One KLOC is 1,000 lines of code.
2. **Classify the Type of Project**: The COCOMO model defines three types of projects:
    - **Organic**: Simple projects with small teams and familiar environments.
    - **Semi-Detached**: Intermediate projects with medium-sized teams and varying experience levels.
    - **Embedded**: Complex projects with strict requirements and heavy hardware/software interaction.
3. **Calculate the Effort (in Person-Months)**: The basic effort equation for the **COCOMO I model** is:

The **COCOMO (Constructive Cost Model)** is a software engineering model that estimates the effort, time, and cost required to complete a software development project based on its size, complexity, and other factors. There are several versions of COCOMO, but the most commonly used is **COCOMO II**.

Steps to Estimate Effort Using the COCOMO Model:

1. **Determine the Size of the Project (KLOC)**:
    - The first step is to estimate the size of the software project in terms of **Kilo Lines of Code (KLOC)**. One KLOC is 1,000 lines of code.
2. **Classify the Type of Project**: The COCOMO model defines three types of projects:
    - **Organic**: Simple projects with small teams and familiar environments.
    - **Semi-Detached**: Intermediate projects with medium-sized teams and varying experience levels.
    - **Embedded**: Complex projects with strict requirements and heavy hardware/software interaction.
3. **Calculate the Effort (in Person-Months)**: The basic effort equation for the **COCOMO I model** is:

$$\text{Effort (PM)} = a \times (\text{KLOC})^b$$

Where:

- a and b are constants that vary based on the type of project.
- KLOC is the estimated number of lines of code.

Constants for different project types:

- Organic: a=2.4,b=1.05a = 2.4, b = 1.05a=2.4,b=1.05
- Semi-Detached: a=3.0,b=1.12a = 3.0, b = 1.12a=3.0,b=1.12
- Embedded: a=3.6,b=1.20a = 3.6, b = 1.20a=3.6,b=1.20

4. **Estimate the Time Required (Development Time)**: The formula for development time (TDEV) in months is:

$$\text{Time (TDEV)} = c \times (\text{Effort})^d$$

Where:

- c and d are constants based on the project type.

Constants for different project types:

- Organic: c=2.5,d=0.38c = 2.5, d = 0.38c=2.5,d=0.38
- Semi-Detached: c=2.5,d=0.35c = 2.5, d = 0.35c=2.5,d=0.35
- Embedded: c=2.5,d=0.32c = 2.5, d = 0.32c=2.5,d=0.32

5. **Calculate the Number of People (Team Size)**: The team size can be determined by dividing the total effort (person-months) by the time estimate (in months):

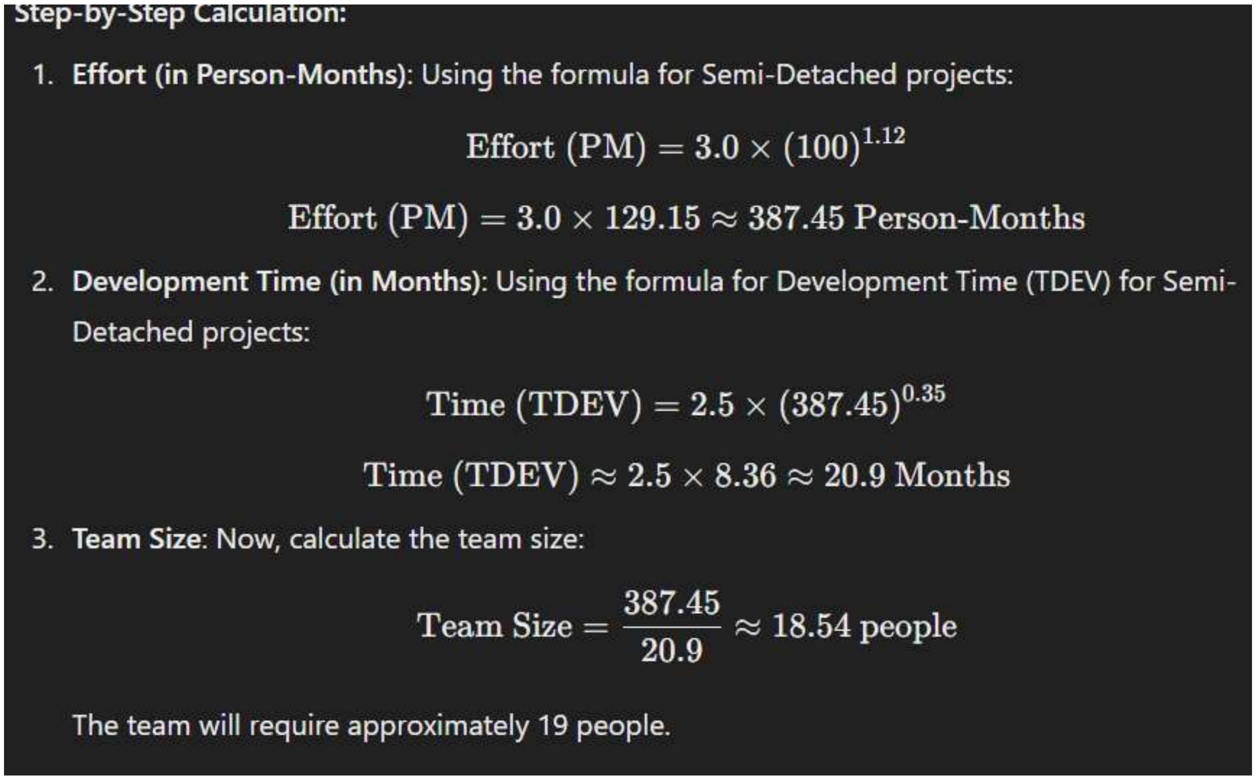
$$\text{Team Size} = \frac{\text{Effort}}{\text{Time}}$$

Example of COCOMO Estimation for an Industrial Project:

Problem: Industrial Automation Software

- **Project Type**: Semi-Detached
- **Estimated Lines of Code (KLOC)**: 100 KLOC (100,000 lines of code)

Step-by-Step Calculation:



Step-by-Step Calculation:

1. **Effort (in Person-Months)**: Using the formula for Semi-Detached projects:

$$\text{Effort (PM)} = 3.0 \times (100)^{1.12}$$

$$\text{Effort (PM)} = 3.0 \times 129.15 \approx 387.45 \text{ Person-Months}$$

2. **Development Time (in Months)**: Using the formula for Development Time (TDEV) for Semi-Detached projects:

$$\text{Time (TDEV)} = 2.5 \times (387.45)^{0.35}$$

$$\text{Time (TDEV)} \approx 2.5 \times 8.36 \approx 20.9 \text{ Months}$$

3. **Team Size**: Now, calculate the team size:

$$\text{Team Size} = \frac{387.45}{20.9} \approx 18.54 \text{ people}$$

The team will require approximately 19 people.

Summary of COCOMO Estimation:

- **Effort (Person-Months)**: 387.45 PM
- **Development Time**: 20.9 Months
- **Team Size**: 19 People

Conclusion:

Using the COCOMO model, we estimate that developing the industrial automation software will require approximately **387 person-months** of effort, taking around **21 months** to complete with a team of around **19 people**.

For more detailed and accurate results, you can also consider additional cost drivers in the **COCOMO II model** such as personnel experience, product complexity, and required reliability.

Lab-12:

Q12.

Reasons Behind the Software Crisis

the crisis primarily arose due to the following reasons:

Increasing Complexity: Software systems became more complex as the demand for more functionalities grew, making it harder to design, build, and maintain systems.

Lack of Standardization: There was no standardized methodology or best practices for software development, leading to inconsistent quality and processes.

Poor Estimation of Resources: Underestimating the time, cost, and resources required to develop complex software systems led to project failures or delays.

Inadequate Testing: Insufficient or improper testing of software before release often resulted in undetected bugs and system crashes.

Poor Communication: Ineffective communication between clients and developers led to misunderstandings in requirements, causing the software to not meet client expectations.

Maintenance Issues: As systems grew more complex, maintaining software (fixing bugs, adding new features) became increasingly difficult and expensive.

Possible Solutions for the Following Scenarios

Case 1: Air Ticket Reservation Software Crash

Reasons for the Failure:

Time-based Bug: The software likely had a defect related to date or time handling. It may have failed to handle a specific time transition (e.g., a noon-to-midnight calculation error or a 12-hour format bug).

Inadequate Testing: The issue may not have been tested properly for different time scenarios, leading to the crash.

Lack of Redundancy: There might not have been an alternative or backup system in place to handle failures, leading to a 5-hour outage.

Possible Solutions:

Thorough Testing: Ensure the system undergoes rigorous testing under various time scenarios, including boundary conditions like midnight/noon transitions.

Continuous Monitoring and Logging: Implement real-time monitoring and logging to detect and diagnose issues before they lead to system crashes.

Backup and Redundancy: Introduce failover mechanisms or backup systems to take over in case of failure to avoid prolonged downtime.

Use Defensive Programming: Anticipate potential edge cases like time-related bugs and handle them proactively in the code.

Case 2: Financial System Malfunction

Reasons for the Failure:

Poor Modularization: The software may not have been designed in a modular way, making it difficult to isolate and debug specific components.

Inadequate Documentation: Lack of proper documentation for the complex system could have hindered the debugging process.

Insufficient Unit Testing: The issue might not have been caught early due to insufficient unit or component testing during development.

Complexity of the System: The sheer size and interdependencies between modules made it difficult to trace the root cause of the malfunction.

Possible Solutions:

Modular Design: Break the system into smaller, independent modules that can be tested, debugged, and updated individually. This reduces the complexity of troubleshooting.

Extensive Documentation: Maintain thorough documentation of the system architecture, modules, and code to aid the development team in understanding and identifying potential defects.

Automated Testing: Implement automated unit, integration, and system testing to ensure that every component works correctly, and detect defects early in development.

Code Reviews and Static Analysis: Regularly perform code reviews and use static analysis tools to identify potential bugs or malfunctions before they affect the production environment.

Version Control and Debugging Tools: Use version control to manage changes and debugging tools to trace issues in complex systems effectively.

Conclusion:

Both scenarios highlight common issues in software development: insufficient testing and the difficulty of managing complex systems.

The key solutions include improving the testing process, adopting modular design principles, and ensuring continuous monitoring, backup systems, and thorough documentation to avoid such failures in the future.