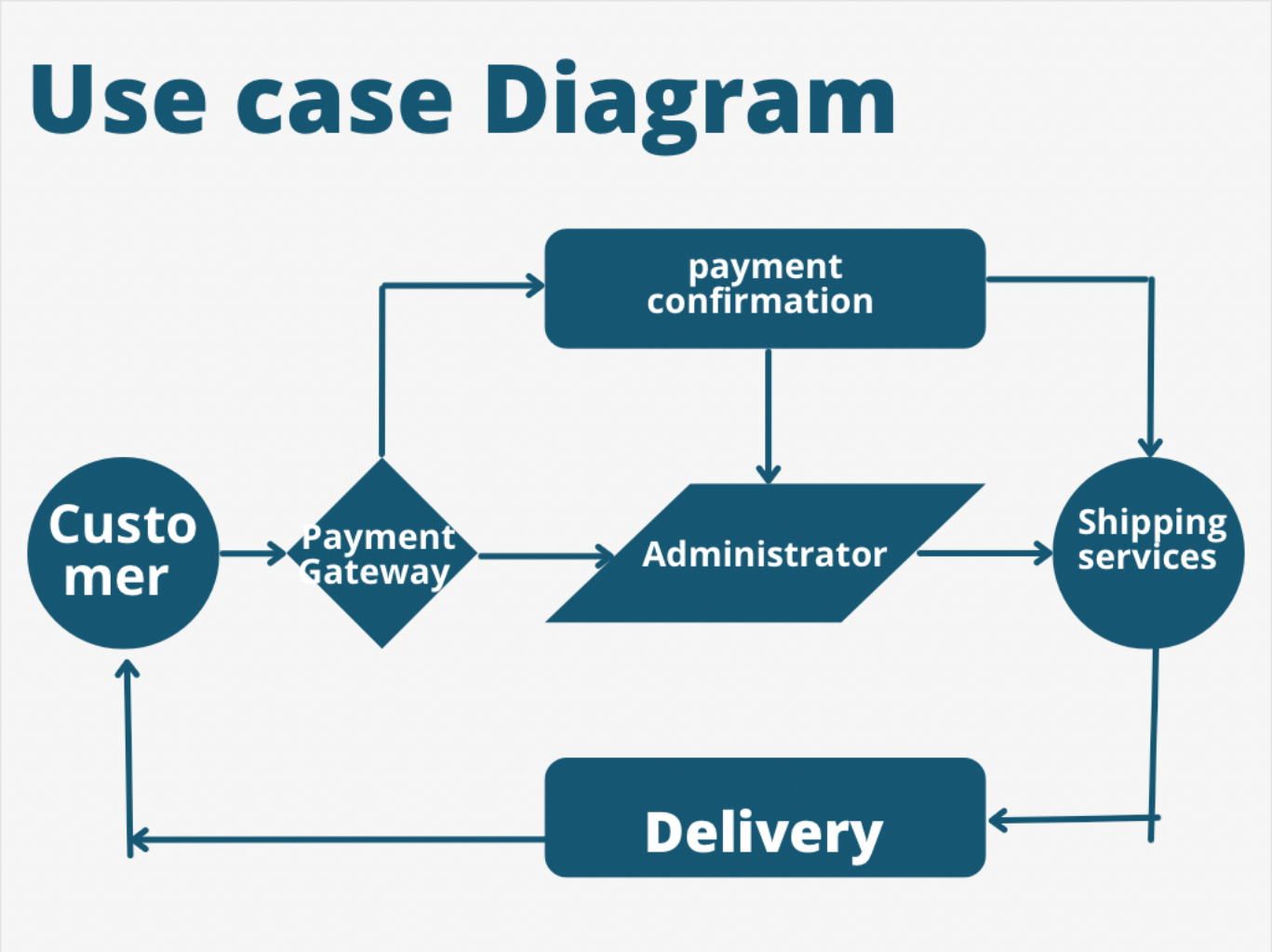
Assignment
Nora Asare Appiah
10852033




Some common actors in an e-commerce platform :

- Customer: The primary actor who interacts with the platform, browses products, places orders, and manages their account.
- Administrator: An actor responsible for managing the platform, including managing products, handling customer support, and monitoring system performance.
- Payment Gateway: An external system or service that handles the payment processing for orders.
- Shipping Service: Another external system or service responsible for managing the delivery of orders.

Arguing against the statement: "Developers should not be involved in testing their code but that all testing should be the responsibility of a separate team.

I disagree with the statement that developers should not be involved in testing their code and that testing should be entirely handled by a separate team. Here are a few reasons to support my argument:

a. Contextual Knowledge: Developers possess in-depth knowledge about the code they have written. They understand the implementation details, potential edge cases, and dependencies involved. This contextual knowledge is valuable during testing because it allows developers to design and execute tests more effectively. Handing off testing entirely to a separate team might lead to a lack of understanding and an inefficient testing process.
b. Early Detection of Issues: Developers testing their code can identify and address issues at an early stage, reducing the overall cost and time required for bug fixing. They can perform unit testing, integration testing, and regression testing during the development process itself. If a separate team solely handles testing, issues may be discovered late in the cycle, leading to delays and increased development costs.
c. Collaboration and Feedback: When developers are actively involved in testing, they can collaborate closely with the testing team. This collaboration fosters a feedback loop where developers gain insights into the real-world behavior of their code, leading to improvements and better-quality software. Developers can work hand-in-hand with testers to resolve issues, clarify requirements, and align on the expected behavior of the software.
d. Faster bug resolution: Developers can quickly reproduce and debug issues they encounter during testing. They have the necessary context and knowledge to investigate the root causes of problems and provide timely fixes, resulting in faster bug resolution.
e). Shared responsibility: Developers have a vested interest in the quality of their code. By involving them in testing, they take ownership of the software's quality and become active participants in ensuring that it meets the required standards.

While involving developers in testing is beneficial, it is also important to have an independent testing team to provide an unbiased perspective and conduct comprehensive testing, including regression testing, integration testing, and user acceptance testing.

3. The challenge of assessing software quality when the customer continually changes the requirements can be addressed in a real-life software development project through the following approaches:

a. Effective Communication: Establishing open and continuous communication channels with the customer is crucial. It helps ensure that the development team stays informed about changing requirements and can adapt accordingly. Regular meetings, feedback sessions, and documentation of requirements changes can help manage expectations and maintain transparency.

b. Agile Development Methodologies: Agile methodologies, such as Scrum or Kanban, are well-suited for projects with evolving requirements. These methodologies emphasize iterative development, frequent customer collaboration, and adapting to change. By breaking down the development process into smaller iterations, it becomes easier to incorporate changing

requirements in a controlled and manageable manner.

c. Prioritization and Impact Analysis: When faced with changing requirements, it's important to prioritize and assess the impact of each change. Collaborate with the customer to determine the criticality and urgency of the changes and evaluate their potential implications on the existing system. This helps in making informed decisions about what changes can be accommodated within the given timeframe and budget.

d. Test-Driven Development (TDD): TDD can be a valuable approach in situations where requirements are subject to frequent changes. By writing automated tests before implementing new functionality or modifying existing code, TDD ensures that the software remains robust and verifiable throughout the development process. Whenever a requirement changes, the corresponding tests can be updated or added to reflect the new expectations.

e. Change Management Process: Implementing a well-defined change management process helps track and document requirement changes. This process should include capturing change requests, assessing their impact, obtaining appropriate approvals, and ensuring traceability of changes. A change control board or similar governing body can help manage and evaluate changes, ensuring they align with the project's goals and objectives.

Addressing the challenge of changing requirements requires a combination of effective communication, flexible development methodologies, and sound change management practices. By embracing adaptability and maintaining a customer-centric approach, software development projects can navigate evolving requirements successfully.

4. Yes, it is possible for a program to be technically correct and yet not exhibit good quality. Technical correctness refers to whether a program adheres to its intended behavior and fulfills its specified requirements. On the other hand, software quality encompasses several other aspects, such as maintainability, scalability, usability, performance, security, and reliability. A program can be technically correct but fall short in one or more of these quality attributes. Here are a few examples:

a. Performance: A program might function correctly but exhibit poor performance characteristics, such as slow response times or high resource utilization. While it achieves the expected output, it may not meet user expectations in terms of speed or efficiency.

b. Usability: A technically correct program can lack a user-friendly interface, making it challenging for users to understand or navigate. The program might fulfill its intended purpose but provide a subpar user experience, leading to frustration and reduced usability.

c. Security: A program might correctly perform its primary tasks but have vulnerabilities or weak

security controls. If it exposes sensitive data or allows unauthorized access, it compromises the overall security of the system, even if it functions as expected.

d. Maintainability: A technically correct program might have poor code structure, lack proper documentation, or be difficult to modify and maintain. It might work correctly at present but become challenging to enhance or adapt to future requirements, hindering its long-term quality.

These examples illustrate that technical correctness alone does not guarantee good software quality. Achieving high-quality software involves addressing various non-functional requirements, meeting user expectations, and considering the broader system attributes beyond functional correctness.

5. In a system with a high availability requirement, it is essential to explicitly handle all exceptions to ensure uninterrupted operation. Here's why:

a. Fault Isolation: By explicitly handling exceptions, you can isolate faults and prevent them from cascading through the system. Unhandled exceptions can propagate to higher levels of the system, potentially causing crashes, system instability, or unexpected behavior. Handling exceptions allows for better error containment and minimizes the impact on the system's availability.

b. Graceful Recovery: Explicitly handling exceptions enables developers to implement graceful recovery mechanisms. When an exception occurs, the system can perform appropriate actions, such as logging the error, rolling back transactions, notifying administrators, or failing over to alternative resources. These recovery measures help maintain system availability and ensure minimal disruption to users.

c. Error Reporting and Monitoring: When exceptions are handled, it becomes easier to track and monitor system errors. By logging exception details and associated contextual information, developers can gain insights into the root causes of failures. This information is valuable for identifying recurring issues, analyzing trends, and proactively addressing potential vulnerabilities or bottlenecks.

d. Reducing Downtime: High availability systems aim to minimize downtime and provide continuous service. Explicitly handling exceptions allows for faster error resolution and reduces the time needed to recover from failures. Proper handling can include retry mechanisms, failover to redundant components, or automatic system healing, ensuring the system remains available even in the presence of errors.

By explicitly handling exceptions, a high availability system can maintain its resilience, quickly recover from failures, and provide uninterrupted service to its users. Neglecting exception

handling could lead to increased downtime, degraded user experience, and compromised availability.

6. Layered architecture vs. Event-driven architecture:
Layered Architecture:

- Layered architecture divides an application into multiple layers, each responsible for a specific set of functionalities.
- Each layer provides services to the layer above it, encapsulating its implementation details.
- Layers communicate in a strict hierarchical manner, with higher layers depending on lower lavers.
- Examples: Traditional web applications with separate layers for presentation, business logic, and data access.

Event-driven Architecture:

- Event-driven architecture focuses on the production, detection, and consumption of events.
- Components within the system communicate asynchronously through events, which represent significant occurrences or state changes.
- Events can trigger actions in other components, leading to loosely coupled and highly scalable systems.
- Examples: Internet of Things (loT) systems, realtime streaming applications, event-driven microservices.