**UML Design Modeling**

Xenia Reid

The University of Arizona Global Campus

CST499: Capstone for Computer Software Technology

Professor Joseph Rangitsch

July 15, 2025

**UML Design Modeling**

**Use case diagram for the Online Course Enrollment System (OCES)**

A diagram of a person with text

AI-generated content may be incorrect.

**State Diagram for the** **Online Course Enrollment System (OCES)**  
A diagram of a course

AI-generated content may be incorrect.

**Class diagram for the Online Course Enrollment System (OCES)**

A diagram of a server

AI-generated content may be incorrect.

**Sequence Diagram for the Online Course Enrollment System (OCES)**

A diagram of a course

AI-generated content may be incorrect.

**Activity Diagram for the Online Course Enrollment System (OCES)**

A diagram of a course

AI-generated content may be incorrect.

In the software development lifecycle, thorough testing plays a crucial role in verifying a system’s functionality, reliability, and usability. Without a structured approach to testing, software can be released with significant defects that negatively impact the user experience, operational performance, or system security. To mitigate these risks, testing is carried out at various levels, each serving a specific function and covering a defined scope. These levels include component testing, integration testing, system testing, and acceptance testing. Each stage contributes to identifying defects and ensuring the software meets its intended requirements. This paper explores the purposes, methodologies, and significance of these four core levels of software testing, emphasizing how they collectively provide a strong framework for validating software across increasing levels of complexity.

Component testing, often referred to as unit testing, represents the most detailed level of software testing. Its purpose is to validate the functionality of individual software modules or components independently. In this context, a component is the smallest unit of software that can be tested, such as a method or class. This type of testing is usually performed by developers during or right after the coding phase, using test stubs and drivers to imitate the behavior of interacting components (Tsui et al., 2018). “Tools used for unit testing are Junit(Java), Pytest (Python), SOAPUI(APIs), etc.” (Testlio, n.d.).

Component testing is essential because it allows developers to update and modify existing components, enabling the creation of updated models. It leverages operational scripts and test stubs to mimic a component’s behavior from a black-box perspective, improving the accuracy of testing. These stubs can be reused and adjusted as components change, offering greater flexibility and efficiency. Unlike unit testing, component testing can be performed on live web servers, helping to detect real-world issues more effectively. It also allows bugs to be identified and resolved early in the process, reducing time, cost, and rework. Moreover, it ensures that components are stable and ready before they reach the QA stage (LambdaTest, n.d). Component testing serves as an important initial safeguard against errors, especially those involving logic and syntax. However, it does not uncover problems rising from interactions between components, highlighting the need for further levels of testing.

Integration testing evaluates the interactions between integrated components or systems. After individual components are validated through unit testing, they are assembled to function together. This level of testing focuses on identifying problems in communication, data exchange, and interface behavior between modules. The primary objectives are to ensure that software components function and interact correctly after integration, that no new issues arise from combining units, and that each component maintains its intended functionality. It also verifies that data flows accurately between modules (Testlio, n.d.). As Spillner et al. state, “The goal of the integration test is to expose faults in the interfaces and in the interaction between integrated components” (pg. 50).

There are several strategies for integration testing, including top-down, bottom-up, big-bang, ad hoc, and backbone. Each approach has trade-offs. For instance, the top-down approach allows early verification of high-level logic and control flow, while the bottom-up approach ensures low-level interactions are stable before progressing upward (Spillner et al., 2014). Tools like “Selenium(Java), Cypress(JS), WebdriverIO(Java and JS), Cucumber(Java and JS)” can support this level (Testlio, n.d.).

Integration testing is particularly important in modular systems or service-oriented architectures (SOA), where components rely heavily on APIs and messaging protocols. As noted by Jorgensen (2013), integration testing is vital for uncovering interface mismatches, data conversion errors, and communication delays, all of which may go unnoticed in isolated unit tests.

After integration testing is complete, it is followed by system testing. “System testing checks if the integrated product meets the specified requirements” (Spillner et al., 2014, pg. 58). System testing is a performance-driven, comprehensive evaluation of the entire software system. This end-to-end process encompasses functional testing, non-functional testing, interface testing, stress testing, and recovery testing. This type of testing is performed by a dedicated QA team in a staging environment that mimics the production setting. System testing follows a black-box approach, meaning testers evaluate the system based on inputs and expected outputs without examining the internal code structure (Powell & Smalley, 2025).

System testing includes multiple types of sub-tests, such as functional, performance, security, and usability testing. Each type focuses on verifying a specific quality attribute of the system. For instance, performance testing evaluates whether operations are completed within an acceptable time frame, whereas security testing verifies that authentication and access control mechanisms function correctly.

As Tsui et al. (2018) explain, system testing validates the system as a whole, offering a final verification that it meets all functional and nonfunctional requirements. According to Beizer (1995), system testing is essential for identifying high-level design flaws and inconsistencies that may not manifest at lower levels of testing.

Acceptance testing is the final testing phase before a software product is released to end-users. It is conducted to confirm that the software meets business needs and user expectations (Gillis, 2024). There are two primary types of acceptance testing: alpha testing, performed within the organization by the client or QA team, and beta testing, conducted by a group of actual users in a live environment that are not from within the organization (Tsui et al., 2018).

Acceptance testing usually follows scenario-based evaluations guided by previously documented user stories or requirements. Its purpose is to confirm that the system meets stakeholder expectations and complies with relevant regulatory, contractual, or usability standards (Spillner et al., 2014). This level of testing is crucial for user confidence and project sign-off. As highlighted by Kaner, Falk, and Nguyen (1999), acceptance testing bridges the gap between technical verification and user validation, ensuring that the system is not just built right but also the right system.

Software testing is a layered process that gradually builds confidence in a system’s correctness and readiness. At the foundational level, component testing validates the functionality of individual modules. Integration testing then checks that these modules interact correctly. System testing evaluates whether the complete system meets its defined requirements, while acceptance testing confirms that the final product aligns with user expectations. By systematically applying each of these testing stages, teams can reduce risks, enhance quality, and increase stakeholder confidence. A robust understanding and execution of each testing phase is crucial for delivering reliable, maintainable, and user-friendly software systems.

**References**

Beizer, B. (1995). *Software testing techniques* (2nd ed.). Dreamtech Press.

Gillis, A. (2024, October 22). *What is acceptance testing?*. TechTarget. https://www.techtarget.com/searchsoftwarequality/definition/acceptance-test.

Jorgensen, P. C. (2013). *Software testing: A craftsman’s approach* (4th ed.). CRC Press.

Kaner, C., Falk, J., & Nguyen, H. Q. (1999). *Testing computer software* (2nd ed.). Wiley.

LambdaTest. (n.d.). *What is component testing: Techniques, tools, and process.* Retrieved from https://www.lambdatest.com/learning-hub/component-testing#two.  
Spillner, A., Linz, T., & Schaefer, H. (2014). *Software testing foundations: A study guide for the certified tester exam* (4th ed.). Rocky Nook.

Testlio. (n.d.). *What is integration testing?* Testlio. https://testlio.com/blog/what-is-integration-testing/#h-what-is-software-integration-testing.

Tsui, F., Karam, O., & Bernal, B. (2018). *Essentials of software engineering* (4th ed.). Jones & Bartlett Learning.

Powell, P. & Smally, I. (2025, April 11). *What is system testing?*. IBM. https://www.ibm.com/think/topics/system-testing.