**UML Models and Testing Levels**

Theresa Musselman

The University of Arizona Global Campus

CST499: Capstone for Computer Software Technology (CSF2501A)

Professor Joseph Rangitsch

April 7, 2025

**UML Models and Testing Levels**

System design is pivotal in the software development lifecycle (SDLC), bridging well-defined requirements and a functional, testable system. A robust design must meet the specified requirements and support testability, ensuring comprehensive validation at every stage. Effective testing evaluates the system's completeness (verifying that all requirements are fulfilled) and consistency (maintaining uniformity across design elements such as terminology, logic depth, and error handling). Beyond functionality, quality attributes like maintainability, reliability, and usability are vital considerations during testing. Established metrics such as cyclomatic complexity, cohesion, and coupling provide objective measures of design quality, helping to identify areas for improvement and ensuring the development of a high-performing, resilient system. To translate the requirements captured in the Educational Portal Management System’s SRS document into a detailed design, this paper will utilize Unified Modeling Language (UML) diagrams, including class, sequence, activity, state, and use case diagrams, to effectively capture both the static and dynamic aspects of the system while addressing key testing methodologies (component, integration, system, and acceptance) to ensure thorough validation and robust implementation.

Unified Modeling Language (UML) diagrams capture a system's static structure and dynamic behavior, offering a standardized and visual approach to analyzing, designing, and documenting software systems. These diagrams ensure that all requirements are addressed and provide critical documentation to assist with validation during testing. Additionally, UML diagrams can help identify reusable **design patterns** by visualizing common structures and relationships across system components, supporting more efficient and modular design practices. Figure 1 presents a system overview through a use case diagram, which is further detailed in Figures 2 through 6, each representing individual use case diagrams for the scenarios outlined in Figure 1. Figure 7 provides the class diagram for the Educational Portal Management System, highlighting its structural components. Additionally, Figure 8 depicts a sequence diagram representing interactions between system components, while Figure 9 illustrates the state diagram, capturing the system’s transitions across various states. To ensure a comprehensive approach to software design, Figure 11 details the high-level architectural design of the system, and Figure 12 outlines the general user interface of the application (Tsui et al., 2018). This collection of UML diagrams collectively supports the system's analysis, design, and validation processes while serving as a foundation for robust testing and implementation.

**Figure 1**

*Overall Use Case Diagram*

A diagram of a diagram

AI-generated content may be incorrect.

**Figure 2**

***Registration System Use Case Diagram***

A computer screen shot of a diagram

AI-generated content may be incorrect.

**Figure 3**

*Login System Use Case Diagram*

A screenshot of a computer

AI-generated content may be incorrect.

**Figure 4**

*Waitlist Management System Use Case Diagram*

A diagram of a diagram

AI-generated content may be incorrect.

**Figure 5**

*Course Management System Use Case Diagram*

A diagram of a course

AI-generated content may be incorrect.

**Figure 6**

*Enrollment System Use Case Diagram*

A diagram of a course

AI-generated content may be incorrect.

**Figure 7**

*Education Portal Management System Class Diagram*

A screenshot of a computer screen

AI-generated content may be incorrect.

**Figure 8**

*Course Enrollment with Waitlist Management Sequence Diagram*

A screenshot of a computer

AI-generated content may be incorrect.

**Figure 9**

*Course Enrollment with Waitlist Management Activity Diagram*

A diagram of a company

AI-generated content may be incorrect.

**Figure 10**

*Course Enrollment with Waitlist Management State Diagram*

A diagram of a course enrollment

AI-generated content may be incorrect.

**Figure 11**

***High-Level Architectural Design***

**A diagram of a software company

AI-generated content may be incorrect.**

**Figure 12**

*User Interface Design*

A screenshot of a computer

AI-generated content may be incorrect.

Component testing evaluates each software unit in a controlled and isolated environment to verify that it operates as intended and adheres to specified component requirements (Pp\_pankaj, 2019). This marks the initial testing phase following the programming stage. Isolation is crucial to ensure that any detected issues can be traced back specifically to the component being tested. Moreover, this approach enables a detailed internal assessment of the software unit. Functional, robustness and quality tests are integral to component testing. Functional tests ensure that all functionalities outlined in the component requirements are implemented correctly and behave as expected. These tests also address negative scenarios, function calls, test data, and special case handling (Spillner et al., 2014). Quality tests, on the other hand, often include static evaluations such as reviews. Metrics such as Halstead's Complexity Metric, McCabe’s Cyclomatic Complexity, and the Henry-Kafura Flow Metric are valuable tools for evaluating the complexity of individual components and supporting the testing process.

Halstead's complexity metric was among the first to quantitatively assess software complexity (Geeks for Geeks, 2024). It evaluates a program by analyzing the use of operators (actions or operations performed on data) and operands (values that the operators act on). Halstead introduced four key metrics to assess a program's overall quality, estimate its potential error rate, and predict the maintenance effort required. These metrics can be used to compare during static testing and identify problematic areas. Additionally, they can be used to prioritize testing and identify complex components that may require more testing. Lastly, effort (E) may be used as a basis for test resource allocation. The metrics and their measurements are as follows:

* n1 = number of unique operators
* n2 = number of unique operands
* N1 = total number of operators
* N2 = total number of operands

These metrics may then be used to calculate:

* Program length:
* Program vocabulary:
* Volume:
* Difficulty
* Effort:

McCabe’s cyclomatic complexity is another approach to identifying the complexity of a program. Cyclomatic complexity utilizes a control flow diagram to identify nodes (N) and edges (E). Nodes represent processing tasks, and edges show the control flow between the nodes. McCabe then relates the number of branches in the design to the program's complexity. Cyclomatic complexity, M, is defined as the following:

The number of connected components, p, typically represents entry or exit points in the control flow graph. Typically, p would be one unless a flow graph has multiple entry and exit points. The result, M, is the cyclomatic number and identifies the number of linearly independent paths within a program’s source code (Tsui et al., 2018). The cyclomatic number can determine the number of test cases required to achieve full coverage (Nanda, 2021). Additionally, high cyclomatic complexity can identify components prone to error. Figure 13 shows the scale used to define the complexity number (Nanda, 2021).

**Figure 13**

*Cyclomatic Number Complexity & Meaning*

A screenshot of a computer

AI-generated content may be incorrect.

The Henry-Kafura Information Flow metric evaluates the complexity of information exchange between modules. This metric considers local flows, indirect local flows, and global flows, including parameter passing, global variable access, inputs, and outputs. The complexity is calculated using the following formula:

Where is the Henry-Kafura complexity for module *p*. Then, is the module's internal complexity (or module length), which can be measured by methods such as lines of code (LOC), cyclomatic, or the Halstead metric. Fan-in is defined as the number of local flows into M plus the number of data structures from which M retrieves information (TutorialsPoint, n.d.). Fan-out is defined as the number of local flows out of M plus the number of structures updated by M. The Henry-Kafura metric helps identify bottlenecks in the program. This metric can be used in static reviews to assess a component's complexity and maintainability. Additionally, analyzing the fan-in and fan-out flows may assist with identifying dependencies that may alter component testing behavior.

Integration testing represents the second level of testing, commencing after the completion of component testing. It aims to confirm that two or more software units interact correctly across various use cases (Sanjoy\_62, 2024). This phase focuses on identifying issues related to interfaces, collaboration, and interoperability (Spillner et al., 2014). Initially, integration testing validates the interaction between individual software units (component integration testing). Once these interactions are confirmed, the testing shifts towards examining connections between subsystems. Eventually, the process extends to testing interfaces with external systems and hardware (system integration testing) (Spillner et al., 2014). Integration testing can also include verifying configuration programs, configuration data, and correct database access. Reusing component test drivers is common during this phase, and monitoring programs may be utilized to evaluate outcomes. Integration strategies such as top-down, bottom-up, ad hoc, or backbone can be applied, with the choice of strategy depending on the system's architecture, the project plan, and the test plan (Spillner et al., 2014). Henry-Kafura Information Flow Metric, Coupling Between Objects (CBO), and McCabe's Cyclomatic Complexity to evaluate internal complexity and external interactions.

The Henry-Kafura metric can assist in resource allocation and prioritization by identifying components with high dependencies that may require additional testing effort. Modules with high **fan-in** or **fan-out** are particularly critical to test, as they rely heavily on interactions with other components and are more prone to errors. Additionally, the metric is helpful for pinpointing bottlenecks within the system. Furthermore, it can be used to validate expected versus actual input and output flows, ensuring accurate system behavior and data interactions. McCabe’s Cyclomatic Complexity, like with component testing, is essential for determining the number of test cases required to achieve full test coverage. Additionally, McCabe’s flowchart analysis provides a clear understanding of control flow, aiding in designing test cases that cover all conditions, branches, and loops, including edge cases.

Coupling Between Objects (CBO) is a part of Chidamber and Kemerer's six C-K metrics. The six C-K metrics for good OO design are: weighted methods per class (WMC), depth of inheritance tree (DIT), number of children (NOC), coupling between object classes (CBO), response for a class (RFC), and lack of cohesion in methods (LCOM) (Racim, 2024). WMC evaluates maintainability by measuring the effort required to maintain a class based on the sum of complexities of all methods in a class. DIT identifies the longest path from a class to its root class to identify the level of inheritance and the potential complexity of the class. NOC evaluates complexity and potential class reuse by evaluating the inheritance hierarchy. CBO assesses modularity and mobility, measuring the degree of interdependence between classes. The RFC metric counts the number of methods that can be executed in response to a message received by an instance of the class to evaluate the complexity of the class’s behavior. LCOM evaluates a program's maintainability and understandability by evaluating the lack of cohesion among the methods of a class. CBO plays a critical role in integration testing by identifying and prioritizing tightly coupled classes or modules that require thorough validation. High CBO values may also aid in resource allocation, as such areas often require mocks or stubs to isolate components during testing. Addressing high CBO enhances modularity, making systems more maintainable and testable overall.

System testing, as the third level of testing, assesses the integrated system to ensure its functionality and performance align with both functional and nonfunctional design requirements. These requirements are typically detailed in system requirement specifications (SRS), risk analyses, user manuals, and more (Pp\_pankaj, 2024). When requirements are missing, incomplete, or inaccurate, exploratory testing can be conducted to determine the necessary system requirements. However, it is crucial to establish comprehensive requirements during the initial stages of the SDLC model, as relying on exploratory testing may lead to delays in product deployment. System testing is conducted from the customer's perspective, with test environments designed to simulate real-world deployment conditions closely. Response for a Class (RFC), Card and Glass's higher-level complexity metrics, and Bieman and Ott’s program slice-based and data slice-based cohesion metrics are valuable tools for enhancing system testing.

Response for a Class (RFC) is one of the six Chidamber and Kemerer (C-K) metrics. RFC quantifies behavioral complexity by measuring the number of potential methods that can be executed in response to a message received by a class. A higher RFC value indicates greater complexity, which increases the likelihood of faults in the system. Consequently, RFC is a valuable metric for resource allocation and test design, helping to ensure that complex areas are thoroughly tested and receive full coverage.

Card and Glass's higher-level complexity uses three equations to measure structural, data, and system complexity. Structural and data complexity is dependent on the fan-out of the module. The structural complexity of module *k* is defined as:

The data complexity of module *k* is defined as:

Where pk is the total number of input and output variables of module *k*. The system complexity is then defined as:

Card and Glass metrics can identify and prioritize complex areas within a system that require more thorough testing. These metrics analyze structural and data complexity, helping pinpoint components or modules that may benefit from refactoring to improve maintainability and scalability. Bieman and Ott’s program slice- and data slice-based cohesion measure is a modern design metric. This measure evaluates the cohesion of a module by analyzing variable declarations and executable logic statements. It uses program slices, which are all statements that affect a specific variable of interest, to model and measure cohesion. The approach focuses on the relationships between data tokens (variables or constants) and their contributions to the module's functionality, specifically through glue tokens (data tokens in more than one data slice) and superglue tokens (data tokens in every data slice of the program). Using the slice-based cohesion metric as a comparison in system testing ensures that the system achieves sufficient modularity and enhanced maintainability. This metric evaluates the similarity and focus of a module’s slices (subsets of code contributing to specific computations), helping identify low-cohesion modules requiring additional testing effort to ensure no unexpected behavior. Additionally, the metric supports identifying modules that may benefit from refactoring to improve modularity and reduce the likelihood of defects during integration. By addressing low cohesion, testers can ensure more robust system behavior, better scalability, and easier future maintenance.

Acceptance testing represents the fourth level of testing and can be conducted either after all other testing phases are completed or concurrently with them. Common types of acceptance tests include contract acceptance, user acceptance, operational acceptance, and field testing, such as alpha and beta testing (Spillner et al., 2014). Contract acceptance testing involves the customer verifying whether the product fulfills the requirements specified in the development contract, as well as adhering to legal, governmental, and safety standards. User acceptance testing is beneficial when customers and end-users belong to different groups, as it evaluates whether the system meets end-user expectations. Operational acceptance, typically performed by a system administrator, ensures that critical operational aspects such as backups, disaster recovery, user management, and security vulnerabilities are adequately addressed (Spillner et al., 2014). Field testing allows the product to be tested in real operational environments by a selected end-user group using a stable pre-release version. This includes alpha testing, conducted at the developer’s site, and beta testing, carried out at the customer’s location.

Most metrics are applied during test design and potential validation and are typically evaluated before acceptance testing. However, to ensure completeness and consistency during testing, a requirements matrix can be utilized to verify that all requirements are met. Additionally, requirements coverage metrics can be employed to determine the percentage of completed requirements. Metrics can also evaluate the percentage of mandatory fields populated in a database, ensuring data completeness. Furthermore, comparing the planned and executed test cases can reveal the percentage of test case completeness.

In conclusion, translating requirements into a robust system design is the cornerstone of successful software development, bridging the gap between user needs and implementation. Utilizing UML diagrams effectively captures both the static and dynamic aspects of the system, ensuring a comprehensive design that supports validation and testing. Structured testing methodologies (component, integration, system, and acceptance testing) provide a step-by-step approach to validate functionality, completeness, consistency, and quality attributes. By incorporating metrics such as Henry-Kafura, McCabe’s Cyclomatic Complexity, and slice-based cohesion, the development process is enhanced with objective measures to identify areas of complexity, guide resource allocation, and support modularity and maintainability. Finally, a customer-centric focus throughout testing ensures that the system aligns with real-world deployment conditions and satisfies both functional and non-functional requirements. Together, these practices enable the delivery of a high-quality, resilient system ready to meet the demands of end-users and stakeholders.

**References**

Akash1295. (2024, June 5). *Cyclomatic Complexity.* Geeks for Geeks.

[Cyclomatic Complexity - GeeksforGeeks](https://www.geeksforgeeks.org/cyclomatic-complexity/)

Geeks for Geeks. (2024, June 12). *Halstead’s Software Metrics – Software Engineering.* Geeks

for Geeks. [Halstead’s Software Metrics - Software Engineering - GeeksforGeeks](https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/)

Nanda, V. (2021, December 17). *McCabe's Cyclomatic Complexity: Calculate with Flow Graph.*

TutorialsPoint. [Mccabe's Cyclomatic Complexity: Calculate with Flow Graph](https://www.tutorialspoint.com/mccabe-s-cyclomatic-complexity-calculate-with-flow-graph)

Pp\_pankaj. (2019, August 14). *Component Software Testing.* Geeks for Geeks.

[Component Software Testing - GeeksforGeeks](https://www.geeksforgeeks.org/component-software-testing/)

Pp\_pankaj. (2024, September 26). *System Testing – Software Engineering.* Geeks for Geeks.

[System Testing - Software Engineering - GeeksforGeeks](https://www.geeksforgeeks.org/system-testing/)

Racim, B. (2024, February 18). *Analyzing CK Metric Results and Quality Assessment.* Medium.

[Analyzing CK Metrics Results and Quality Assessment | by Benkaddour Racim | Medium](https://medium.com/@benkaddourmed54/analyzing-ck-metrics-results-and-quality-assessment-a70ba56534f0)

Sanjoy\_62. (2024, September 26). *Integration Testing – Software Engineering.* Geeks for Geeks.

[Integration Testing - Software Engineering - GeeksforGeeks](https://www.geeksforgeeks.org/software-engineering-integration-testing/)

Spillner, A., Linz, T., & Schaefer, H. (2014).

[Software testing foundations: A Study Guide for the Certified Tester Exam (4th ed.)](https://uagc.instructure.com/courses/140477/modules/items/7169874).

Rocky Nook.

Tsui, F., Karam, O., & Bernal, B. (2018). [Essentials of Software Engineering](https://uagc.instructure.com/courses/143469/modules/items/7317752) (4th ed.). Jones &

Bartlett Learning.

TutorialsPoint. (n.d.). *Measuring the Structure*. TutorialsPoint. [Measuring the Structure](https://www.tutorialspoint.com/software_quality_management/software_quality_management_measuring_structure.htm)