**UML Design Modeling**

Charlie Farris

University of Arizona Global Campus

CST499: Capstone for Computer Software Technology

Dr. Charmelia Butler

November 13, 2022

**UML Design Modeling**

The deployment of good software consists of many steps beginning with requirements gathering, then design modeling. According to Booch (2005), “Modeling is a central part of all the activities” involved in getting good software developed and deployed. Modeling provides a visual of the system’s architecture and allows the designer to have complete control of that architecture. It also provides a better means by which to understand the system being built and can uncover possibilities for reuse and simplification of code. Finally, models aid in risk management.

**Testing**

Once a software project is designed and the specifications at each level are developed, the system should pass verification at each of the four levels of testing as the code comes to completion. This allows for the system as a hole to pass validation. The four levels of testing are component testing, integration testing, system testing, system integration testing, and acceptance testing.

Software component testing is used to verify each individual component of the system being built or updated. Components range from program modules and classes to database scripts and data conversion elements. The component being tested needs to be in an independent and controllable state and needs to be understood by the user. Being able to test at the component level allows testers to identify defects early on that can be resolved quickly compared to the time it would take if the defect was not found until later levels of testing (para. 3 – 5, 2019). Spillner et al., in their book Software testing foundations, state that Component testing is White box and Black box testing. According to other reputable sites (GeeksforGeeks, Javapoint, Guru99, and others), there is confusion on this point. Many see that this first level of testing is broken up into component and unit testing. Unit testing is done by development and is considered white box testing. Whereas component testing is completed by testers and is considered black box testing as the testers do not know what the actual code looks like. Component testing is the first attempt at testing the software components. The objectives of component testing are to prove or disprove the functionality, robustness, efficiency, and maintainability of the component. There are several test strategies to choose from when conducting component testing. White box testing requires the source code to be analyzed in addition to the testing. In black box testing, input is provided to the component, and the output is collected. This testing is an iterative approach as the code is tested with the previously developed component test cases, then the code is improved until it passes the tests.

Integration testing follows component testing and begins the process of testing larger sections of the system by testing the interfaces between the components. Spillner et al. inform us that this is done by testing structural units, subsystems, limited combined subsystems, and database connections (2014, Chpt. 3). Data loggers, known as Monitors, are very useful in integration tests as they record the information passing between components. To integrate the components, four generic strategies the test manager can choose from are Top-down integration, Bottom-up Integration, Ad hoc integration, and Backbone integration. Top-down and Bottom-up integration is used when the system design is structured hierarchically. Ad hoc integration involves integrating components as development and component testing are completed. Backbone integration involves building a backbone to add components gradually. Non-incremental integration is also called 'big bang integration' and should be avoided as much as possible. This integration strategy involves waiting until all components have been developed and tested, throwing them all together in one step, and then testing. The structural units and subsystems are then tested to ensure that all the components they comprise of collaborate without any issues. Non-incremental integration can cause a lot of rework and time loss. Regardless of the method selected, integration testing exposes any faults in the interfaces and in the interaction between the integrated components. This level of testing is necessary to ensure that the interaction between components works as designed.

Once integration testing is completed, the system as a whole can be tested. System tests are designed to check the system against customers' specified functional and non-functional requirements. These requirements can be found in specification documentation, Risk analyses, user manuals, and training material. System testing tests all of the integrated components in an environment that represents as closely as possible the client production environment that end users will use. Instead of using test drivers, the software is installed on a complete system that mirrors the hardware, system software, device drivers, networks, and any external system connections required, such as database connections. In addition to testing the system itself, items like system and user documentation, configuration settings, and system configuration optimization are also covered and tested. From this testing, documentation of failures from the inconsistent, incorrect, or incomplete implementation of the requirements should be created. Also included would be any overlooked or undocumented requirements that have been identified during the process.

When a system is not a standalone system and links into external systems, there is an additional stage that comes just before acceptance testing, where these external interfaces are tested. This stage is called system integration testing. Depending on the software being developed, this can be due to hardware, outside networks, custom off-the-shelf applications, and many other reasons. This type of testing takes the shape of requirement-based functional testing, as the testers have no control over the external system.

The acceptance testing level involves the end customer verifying that the system meets the specified requirements and end-user needs and expectations. The previous levels of testing have involved testing the functional and non-functional requirements from the development and testing perspective (Spillner et al., 2014). However, acceptance testing involves focusing on the customer and user experience perspective. This testing involves the customer and end-user. Four typical forms of acceptance testing may be applied at this time. If system testing is completed, but the software is not quite ready to be handed over to the client, internal Dog food testing can be applied. (If you made it, you should try using it yourself first.) Contract acceptance testing takes the requirements outlined in the contract and verifies that the software meets them. User acceptance testing uses business processes and typical usage scenarios (use cases) to verify that the software meets user requirements and expectations. Operational testing, approved by the user's system admins, involves testing backup/restore processes, disaster recovery, user management, and checking for vulnerabilities in security. Alpha and Beta testing involve providing the software to a smaller group of end-users to test usability.

If the system is ready for the client, then this is the point where we get the Go/No-Go. If they accept the system, they sign off on the testing, but if not, then there are more issues to resolve first.

**Figure 1**

***User Registration and Login Activity Diagram***

Diagram

Description automatically generated

**Figure 2**

***Course Enrollment Activity Diagram***

Diagram

Description automatically generated

**Figure 3**

***Class Diagram***

Graphical user interface, website

Description automatically generated

**Figure 4**

***Sequence Diagram***

Text

Description automatically generated

**Figure 5**

***State Diagram***

Diagram

Description automatically generated

**Figure 6**

***Use Case Diagram***

Diagram

Description automatically generated

**References**

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The unified modeling language user guide* (2nd

ed). Addison-Wesley.

Spillner, A., Linz, T., & Schaefer, H. (2011). Software testing foundations: A study guide for the certified tester exam: foundation level, ISTQB compliant (3rd ed). Rocky Nook ; Distributed by O’Reilly Media.

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

Bartlett Learning. Retrieved from <https://platform.virdocs.com/r/s/0/doc/581911/sp/175231953/mi/563314074?cfi=%2F4%2F4>.