COURSE UNDERSTANDING REPORT

Group member: Trieu Huynh Ba Nguyen (000405980), Pouya Amiri, Eduard-Gabriel Ailincai, Delia Fliscu, Arlis Arto Puidet

Topic 1: Introduction, importance and need of software quality assurance

Software quality encompasses the attainment of explicit functional and performance requisites, meticulous adherence to documented development standards, and the implicit traits expected from professionally crafted software. Quality management transcends the mere reduction of defects; it tackles essential inquiries such as comprehensive testing, software reliability, acceptable performance benchmarks, usability, and strict adherence to development standards.

The necessity for software quality assurance manifests in a multitude of scenarios. Safety-critical systems, which possess the potential to cause harm, loss of life, or environmental devastation, demand software of the utmost quality. Mission-critical systems, wherein failure could result in the collapse of goal-oriented activities, as well as business-critical systems, where failure incurs substantial costs, equally necessitate high-caliber software.

Nevertheless, the pursuit of software quality often confronts formidable challenges. Specifications may suffer from incompleteness or inconsistency, thereby engendering an imbalance between customer quality requirements and developer quality requirements. Moreover, certain quality attributes prove arduous to specify, either directly or indirectly.

To delineate quality, a hierarchical quality model prevails, comprising quality attributes and their respective weights contingent upon the project's contextual nuances. Quality attributes may be categorized as either external, pertaining to the functioning system or ongoing processes, or internal, derived from the product or process description.

Standards serve as an indispensable instrument in safeguarding quality. Product standards demarcate the essential characteristics that all constituents ought to exhibit, while process standards elucidate the proper execution of the software development process. The ISO 9000 series furnishes internationally recognized standards for quality management, which apply to diverse industries, encompassing the realm of software development.

Software quality management concerns itself with attaining the requisite level of excellence in a software product. This entails the establishment of organizational processes and standards, the implementation of specialized quality procedures at the project level, and the formulation of a meticulous quality plan that establishes quality objectives and outlines the processes and standards to be adhered to throughout the project's course.

Activities encompassed within quality management encompass conducting independent audits of the software development process, ensuring that project deliverables conform to organizational standards and objectives, and maintaining objectivity by preserving a clear delineation between the quality team and the development team.

Topic 2: Software Testing Techniques

Software testing is an indispensable facet of software development that encompasses the authentication and substantiation of a system. Verification is focused on ensuring adherence to system specifications, while validation aims to ascertain if the system satisfies user requirements in practical situations. The primary objectives of verification and validation encompass instilling confidence in the software's suitability for its intended purpose, uncovering defects, and evaluating the system's efficacy and user-friendliness.

The level of confidence in the software is contingent upon factors such as its significance to the organization, user expectations, and the prevailing market conditions. Diverse techniques are employed in software testing, including static and dynamic verification. Static verification entails scrutinizing the system's static representation to identify potential issues, often through code inspections and automated static analysis. Conversely, dynamic verification involves executing the system with test data and observing its behavior.

Code inspections are formalized reviews aimed at defect identification, typically conducted by a group of programmers. These inspections primarily focus on logical errors, code compliance, and programming style, but they cannot evaluate non-functional aspects such as performance or usability. Automated static analysis tools assist in identifying potential bugs and complement the inspection process. In instances where a mathematical specification of the system is available, formal verification methods, rooted in mathematical analysis, can be employed.

Dynamic verification, also known as software testing, is imperative for evaluating the behavior of software in real-world scenarios. It represents the sole technique that verifies non-functional requirements. Software testing can be classified into defect testing and validation testing. Defect testing endeavors to uncover flaws within the system, whereas validation testing demonstrates that the software fulfills its requirements.

Testing is conducted throughout the software development process, with different stages including unit testing, module testing, sub-system testing, system testing, and acceptance testing. Test case design involves creating effective tests for both validation and defect testing. Approaches to test case design encompass requirements-based testing (black-box), partition testing based on input/output domains, and structural testing (white-box) based on code analysis.

Testers encounter challenges in determining what to test due to the infeasibility of exhaustive testing. Test cases are grouped into equivalence classes, and proficient test case design is essential to maximize test coverage and error detection. Requirements-based testing involves deriving tests from system requirements or use cases, with a focus on operations, input, and output. Structural testing analyzes the internal structure of the program, striving to execute all program statements.

Integration testing ensures the proper functioning of multiple modules together by testing their interfaces. Different strategies for integration testing include big-bang testing, top-down testing, bottom-up testing, and sandwich testing. Each strategy possesses its own advantages and disadvantages in terms of thoroughness and early detection of flaws.

Test-driven development (TDD) is a development approach that underscores the creation of tests before writing the actual code. TDD aids in achieving comprehensive code coverage, simplifying debugging, and serving as documentation for the system.

Release testing is performed to demonstrate that a specific system release satisfies its designated functionality, performance, and dependability. It typically follows a black-box testing process based on the system specification. User testing is pivotal for incorporating user feedback and validating the system's reliability, performance, usability, and resilience in real-world environments.

Topic 3: Software Metrics

The topic surrounding software metrics underscores the paramount importance of quantifying diverse facets of software development processes, projects, and products. Measurement assumes a critical role in incessantly enhancing software processes, facilitating decision-making, and safeguarding the quality and efficacy of software engineers. This topic elucidates the defining attributes of pivotal terms germane to software metrics, such as measures, metrics, and indicators.

The report accentuates the comprehensive scope of software metrics, encompassing distinct categories, namely process metrics, project metrics, and product metrics. Process metrics entail the quantification of attributes pertaining to the software development process, such as development time, employed methodologies, and other consequential events or incidents. Project metrics serve as indispensable tools for project management, aiding in cost estimation and fostering informed decision-making based on past projects. Conversely, product metrics pertain directly to the software product itself and its associated artifacts, such as code, design documents, and user manuals. These metrics also serve the purpose of evaluating software quality, exemplified by the measurement of module complexity through cyclomatic complexity analysis.

Process metrics are amassed to glean insights into the software process and engender enduring process improvement. By measuring specific process attributes and deriving significant metrics, one can establish indicators that pave the way for strategies aimed at process enhancement. The report also alludes to private metrics, which are tailored to individuals or teams, and public metrics, which are measured at a team level to fortify an organization's process maturity.

Project metrics assume a pivotal role in monitoring and governing software projects. They empower project managers to minimize development time, evaluate product quality, and fine-tune technical approaches to augment quality. Product metrics ascertain various facets of the software product across diverse stages of development, including design intricacy, program size, and documentation volume.

Software measurement entails the utilization of both direct and indirect measures. Direct measures encompass quantifiable entities such as cost, effort, lines of code (LOC), execution speed, memory size, and defects, among others. While these measures furnish concrete data, assessing them in terms of quality, functionality, complexity, reliability, efficiency, and maintainability can prove more challenging. Function-oriented metrics concentrate on the functionality rendered by the software and frequently employ measures like function points, which establish an empirical relationship between software counts and complexity assessments. Conversely, size-oriented metrics derive from the software's size, as evidenced by metrics like errors or defects per KLOC or productivity measured as LOC per person-per-month.

Topic 4: Software Quality Management

Software quality management constitutes a pivotal facet of the software development process, bearing the responsibility of ensuring that the software not only meets explicitly stated requirements and adheres to established standards but also satisfies implicit requirements. The evaluation of quality is predicated on the software's ability to fulfill these criteria and align with user needs. Crucially, software testing assumes a vital role in appraising and enhancing software quality, aiming to ascertain that the software functions as intended and remains dependable and maintainable throughout its life cycle.

Software quality assurance encompasses a range of activities encompassing the improvement of software engineering processes, the design of fault-tolerant software, and comprehensive verification and validation, which includes testing. It is imperative to note that while testing is indicative of quality, it does not inherently confer quality. Quality cannot be imbued into a product solely through testing; it must be incorporated into the development process from its inception.

A sound comprehension of errors, faults, and failures proves indispensable in the realm of software quality management. Failures typically arise as a consequence of system errors stemming from faults within the system. However, not all faults culminate in errors and subsequent failures. Aptly implemented error detection, recovery mechanisms, and inherent protective measures can ameliorate the impact of errors and forestall failures.

The overarching objective of quality assurance lies in guaranteeing that the delivered software system exhibits minimal defects and that any residual defects do not give rise to significant disruptions or damages. Diverse techniques are employed in quality assurance, encompassing defect prevention, defect reduction through inspections and testing, and defect containment through fault tolerance and fault containment measures.

Quality planning represents an integral component of software quality management, entailing the delineation of desired product qualities, the establishment of quality assessment processes, and the identification of the most pivotal quality attributes. Conciseness and a focused approach in quality plans are imperative to ensure readability and adherence.

Software quality extends beyond a mere adherence to specifications. It encompasses non-functional characteristics that exert a substantial influence on the subjective user experience, such as usability, reliability, security, adaptability, and efficiency. Striking a delicate balance among these attributes can prove challenging, as optimizing one may necessitate trade-offs with others. Consequently, the quality plan should accord priority to the most critical attributes for the specific software under development.

The relationship between the software development process and product quality is intricate and not entirely comprehended. The quality of the product is invariably influenced by the quality of the production process. Nonetheless, individual skills, experience, and external factors wield considerable influence over product quality.

Cultivating a culture of quality assumes utmost significance in attaining a high level of product quality. Quality managers play a pivotal role in fostering such a culture by encouraging teams to assume responsibility for their work, supporting quality improvement initiatives, and promoting professional conduct among team members.

Software standards are of paramount importance in quality management as they define the requisite attributes and best practices. They furnish a framework for ensuring consistency, evading past errors, and facilitating comprehension and continuity within an organization. Product and process standards guide the development process and ensure compliance.

Reviews, inspections, and testing stand as pivotal activities in software quality management. Software inspections center around static verification, aiming to unveil issues through meticulous analysis of the system representation. Software testing, conversely, involves dynamic verification by executing the system with test data and observing its behavior. Both reviews and inspections prove effective techniques for identifying errors and enhancing software quality.

Software measurement and analytics represent invaluable tools in quality management, rendering quantitative data about the software and the development process. These data enable insights and inferences about product and process quality. Software analytics, in particular, involves the automated analysis of substantial volumes of data to unveil relationships and patterns that can enlighten project managers and developers.

Topic 5: Software Quality Measurement

Software quality measurement constitutes a fundamental aspect of evaluating and appraising the quality characteristics of software systems. ISO 25010:2011 presents an all-encompassing framework for software quality attributes, which are classified into multiple dimensions.

Functional suitability focuses on the extent to which a software product or system effectively fulfills explicitly stated and implied requirements. This dimension encompasses aspects such as functional completeness, correctness, and appropriateness.

Reliability pertains to the system's capability to perform specified functions within defined conditions. It encompasses factors like maturity, availability, fault tolerance, and recoverability.

Performance efficiency assesses the system's performance in relation to resource usage, including factors such as time behavior, resource utilization, and capacity.

Usability evaluates the efficacy and efficiency with which users can accomplish their objectives using a product or system. It encompasses aspects like appropriateness, recognizability, learnability, operability, user error protection, user interface aesthetics, and accessibility.

Security concentrates on safeguarding information and data from vulnerabilities, incorporating aspects like confidentiality, integrity, non-repudiation, accountability, and authenticity.

Compatibility appraises a product or system's ability to exchange information and execute requisite functions within a shared environment.

Maintainability evaluates the ease with which a product or system can be modified to enhance, rectify, or adapt to changes in the environment or requirements. This dimension includes modularity, reusability, analysability, modifiability, and testability.

Portability assesses the simplicity with which a system, product, or component can be transferred from one environment to another. It comprises adaptability, installability, and replaceability.

While these quality attributes are of utmost importance, conflicts may arise as optimizing one attribute may have an impact on others. Consequently, organizations should define the most vital quality attributes and accordingly develop a quality plan.

ISO 9001 standards furnish a framework for constructing quality management systems that can be employed by organizations involved in software development. The ISO 9001 certification ensures conformity to quality management standards, albeit with a primary focus on adherence to standards rather than user experience.

Software measurement plays a pivotal role in deriving numerical values for software attributes and facilitates objective comparisons between techniques and processes. However, a paucity of established standards in this domain and the lack of systematic employment of software measurement by many organizations present challenges.

Metrics, such as lines of code, complexity, and resource usage, can be employed to evaluate product attributes, identify sub-standard components, and prognosticate product quality or control the software process. Nonetheless, the assumptions and interpretation of metrics can prove arduous, necessitating meticulous analysis within the software context.

Empirical software engineering, founded on data collection and experimentation, forms the basis for research in software measurement and metrics. However, translating research findings into practical software engineering practice presents challenges.

Topic 6: Quality of software requirement engineering, code, testing, and architecture

An essential facet of ensuring software quality involves the discernment and rectification of "bad smell" elements within the development process. These elements serve as indicators that point towards potential issues or deficiencies in various domains. Code smell, architectural smell, test smell, and requirement smell represent diverse types of noxious elements capable of impinging upon the overall quality of the software.

Code smell manifests as specific structures or patterns within the code base, exerting detrimental effects on its maintainability and analyzability. These odors serve as indications of code segments that necessitate refactoring for enhancement. Code smell can be categorized into distinct classes such as bloaters, object-orientation abusers, change preventers, dispensables, and couplers. Each classification signifies particular issues that require redressal, including protracted methods, voluminous classes, excessive coupling, and duplicated code.

Architectural smell pertains to pernicious elements that materialize at a higher level of the system's granularity, revealing underlying deficiencies in its architectural design. Examples of architectural smell includes unstable dependencies, hub-like dependencies, and cyclic dependencies, all of which have adverse ramifications for system quality and exacerbate the challenges associated with maintenance and reusability.

Test smell denotes suboptimal programming practices within the unit test code, suggesting potential design flaws within the test source code. These smell encompass issues such as assertion roulette, redundant assertions, empty tests, and magic number tests. Addressing test smell assumes significance in guaranteeing the efficacy and reliability of test coverage.

Requirement smell functions as indicators of quality violations within requirement artifacts. They serve as red flags for potential complications within the requirements themselves, such as ambiguous phrasing or incomplete sequences. Ensuring the quality of requirements assumes paramount importance, given its direct impact on the quality of the software under development.

While developers may theoretically view code smell as harmful, the actual impact of code smell on software quality remains inconclusive. Studies have demonstrated that the relationship between code smell and software quality attributes is intricate, with both positive and negative evidence observed. Consequently, blindly refactoring code based on the presence of code smell may not always be prudent, necessitating further research to comprehend the conditions under which code smell adversely affect software quality.

Similarly, architectural smell may not always arise from code smell and warrant particular attention. Researchers should explore their deleteriousness, while practitioners should carefully consider the opportune moments for code refactoring to eliminate architectural smell.

Topic 7: Quality in software construction and coding

The subject matter concerning the quality of software construction and coding delves deeply into the intricate notion of technical debt and its far-reaching implications. Technical debt entails the practice of opting for suboptimal solutions in order to expedite the development process, with full awareness that the repercussions will need to be addressed at a later stage. This phenomenon can be likened to incurring a financial debt, where immediate advantages are gained, but the debt must ultimately be repaid with accumulated interest.

Technical debt exhibits two defining components: the principal, which embodies the cost of rectifying code-related issues after the software release, and the interest, which denotes the ongoing information technology costs arising from violations that result in technical debt. These encompass augmented maintenance expenses and heightened resource consumption.

The accrual of technical debt can be ascribed to several factors, including shoddy architecture, deficient design, hasty coding practices, and a lack of emphasis on quality. The peril lies in deferring the repayment of this debt, as every minute spent working with suboptimal code perpetuates an ever-increasing interest on that debt.

Various tools have been devised to quantify technical debt, such as CAST, Sonargraph, NDepend, SonarQube, Squore, CodeMRI, and Code Inspector. SonarQube, in particular, has garnered substantial adoption by examining code compliance against a predetermined set of rules and estimating the time required for refactoring as part of addressing technical debt. It classifies violations as bugs, code smells, or vulnerabilities, with each being assigned a specific severity level.

Nevertheless, the accuracy and reliability of SonarQube's estimations have been subject to scrutiny. Studies have indicated that code smells identified by SonarQube do not consistently result in a decrease in maintainability, and bugs are not necessarily the fundamental cause of faults. Additionally, the estimated time provided by SonarQube for resolving issues often overestimates the actual effort required for rectification.

Topic 8: Quality and QA activities in software deployment and maintenance

This topic delves into the significance of quality and quality assurance (QA) activities in the deployment and upkeep of software. Maintenance pertains to the alteration of a software product subsequent to its delivery in order to rectify faults, enhance performance, adapt to new requirements, or acclimate to a modified environment. The maintainability of a software system is gauged by its ease of rectification, expansion, or contraction to meet novel demands and restore it to normal operational conditions after maintenance.

Maintenance presents numerous challenges, encompassing inadequately documented programs, lack of structure, absence of standardized data models, and the incapacity to conduct maintenance on systems not designed for such purposes. Maintenance is a multifaceted endeavor due to the intricacies involved in tracing the product or the process that engendered it, insufficient documentation of alterations, instability of changes, and the far-reaching impact caused by modifications. It is often misconceived that maintenance solely occurs post-delivery, thereby disregarding its significance throughout the software lifecycle.

Software is described as an assemblage of interconnected components or modules that exchange data to execute tasks. These components or modules oversee specific activities or functions, such as GUI drawing, event handling, or networking, and are comprised of objects that facilitate information exchange. Objects are established based on defined classes and are fashioned through programming to execute routines and manage messages.

The software lifecycle encompasses a variety of maintenance activities, including change requests from users, customers, or management. These change requests must be meticulously analyzed and implemented, particularly in urgent scenarios such as fault rectification, alterations to the system's environment, or critical business requirements.

Maintenance is pivotal since it consumes a substantial portion of the financial resources allocated to the software lifecycle. It addresses issues that may emerge during software operation and eluded detection during validation and acceptance phases. Furthermore, maintenance encompasses software enhancements to fulfill new or altered user requirements and becomes necessary when upgrading system components or modifying external software interfaces. Reports indicate that up to 93% of software costs are incurred by maintenance and development during the maintenance phase.

ISO/IEC 14764 delineates distinct categories of maintenance. Corrective maintenance centers on rectifying errors, including emergency maintenance to tackle unscheduled issues. Adaptive maintenance entails adapting the system to alterations in the environment, encompassing both hardware and software modifications. Perfective maintenance caters to evolving user requirements and may involve additive maintenance to augment the system's content. Finally, preventive maintenance strives to amplify the system's maintainability.

Grasping the challenges and significance of maintenance is imperative given its substantial impact on software projects and overall costs. Despite this, there is often a dearth of emphasis on comprehending maintenance compared to other phases of software development.