**CMP 403- SOFTWARE ENGINERING**

**SUMMARY OF CHAPTER 8 & 9**

**GROUP MEMBERS**

1. AIYEWUMI DAVID OLUWOLE BHU/20/04/09/0011
2. ANOINTING EBUBE DAUDA BHU/20/04/05/0032
3. ORJI CHINONYEREM PEACE BHU/20/04/09/0007
4. CHUKWU DANIEL NONSO BHU/20/04/05/0010
5. AROMOLARAN ADENIKE ELIZABETH BHU/20/04/05/0136
6. JOLLY D JOSEPH BHU/20/04/05/0079
7. ABRAHAM BENJAMIN ORUME BHU/20/04/05/0145
8. AMBINKANME MBOSIRE MUSA BHU/20/04/05/0040
9. MAIGIDA GYENOM TALATU BHU/20/04/05/0013
10. DANJUMA KUYET DIVINE BHU/20/04/05/0011
11. BENCALVIN ISAAC AMBI BHU/20/04/05/0100

**CHAPTER 8**

**Program Testing:**

Definition:

* Intended to demonstrate that a program does what it's supposed to and uncover defects before use.
* Involves executing a program with artificial data and checking for errors or anomalies.

Purpose:

* To reveal the presence of errors, not their absence.
* Part of the broader verification and validation process, including static validation techniques.

**Program Testing Goals:**

1. Demonstrate Compliance:

* For custom software, a test for every requirement in the document.
* For generic software, tests for all system features and combinations to be incorporated.

2. Discover Incorrect Behavior:

* Defect testing focused on exposing undesirable behaviors like crashes, unwanted interactions, and data corruption.

**Validation and Defect Testing:**

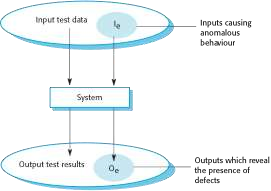
Validation Testing:

* Demonstrates correct operation under expected use conditions.
* A successful test confirms the system operates as intended.

Defect Testing:

* Designed to expose defects, even with obscure test cases.
* A successful test reveals incorrect system behavior, exposing defects.

Figure 1.1: *Input-Output Model of Program Testing*



**Verification vs. Validation**

Verification:

* "Are we building the product right?"
* Software conforms to its specification.

Validation:

* "Are we building the right product?"
* Software does what the user requires.

**V & V Confidence:**

* Aims to establish confidence that the system is 'fit for purpose.'
* Depends on software purpose, user expectations, and marketing environment.

**Inspections and Testing:**

Software Inspections:

* Concerned with static system representation analysis (static verification).
* May use tool-based document and code analysis.

Software Testing:

* Concerned with exercising and observing product behavior (dynamic verification).
* Involves executing the system with test data.

**Advantages of Inspections**

* Errors don't mask each other as in testing.
* Incomplete versions can be inspected without extra costs.
* Broader quality attributes can be considered.

Figure 1.2: *A Model of the Software Testing Process*



**Stages of Testing**

**1. Development Testing:**

* System tested during development to find bugs and defects.
* Includes unit testing, component testing, and system testing.
* Release Testing: Separate testing team tests a complete version before release.
* User Testing: Users test the system in their own environment.

**2. Unit Testing**

* Testing individual components in isolation.
* A defect testing process.

**3. Object Class Testing**

* Complete test coverage involves testing all operations, setting/interrogating attributes, and exercising the object in all states.
* Inheritance complicates object class testing.

**4. Weather Station Testing**

* Define test cases using a state model and identify sequences of state transitions to be tested.

**5. Automated Testing**

* Automate unit testing whenever possible for running and checking tests without manual intervention.
* Use a test automation framework like JUnit.

**Automated test components**

* A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.
* A call part, where you call the object or method to be tested.
* An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

**Unit test effectiveness**

* The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
* If there are defects in the component, these should be revealed by test cases.
* This leads to 2 types of unit test case:

i. The first of these should reflect normal operation of a program and should show that the component works as expected.

ii. The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

**Testing strategies**

* Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.

i. You should choose tests from within each of these groups.

* Guideline-based testing, where you use testing guidelines to choose test cases.

i. These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

**Partition testing**

* Input data and output results often fall into different classes where all members of a class are related.
* Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
* Test cases should be chosen from each partition.

Figure 1.3: *Equivalence partitioning*

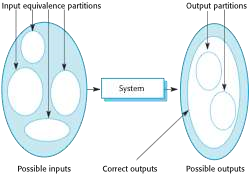
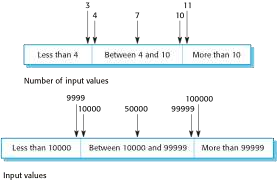


Figure 1.4: *Equivalence partitions*



**Testing guidelines (sequences)**

* Test software with sequences which have only a single value.
* Use sequences of different sizes in different tests.
* Derive tests so that the first, middle and last elements of the sequence are accessed.
* Test with sequences of zero length.

**General testing guidelines**

* Choose inputs that force the system to generate all error messages
* Design inputs that cause input buffers to overflow
* Repeat the same input or series of inputs numerous times
* Force invalid outputs to be generated
* Force computation results to be too large or too small.

**Key points**

* Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
* Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers.
* Development testing includes unit testing, in which you test individual objects and methods component testing in which you test related groups of objects and system testing, in which you test partial or complete systems.

**Chapter 8**

**Software Testing-Lecture 2**

**Component testing**

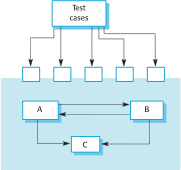
* Software components are often composite components that are made up of several interacting objects.

i. For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.

* You access the functionality of these objects through the defined component interface.
* Testing composite components should therefore focus on showing that the component interface behaves according to its specification.

i. You can assume that unit tests on the individual objects within the component have been completed.

Figure 1.5: *Interface testing*



**Interface testing**

* Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
* Interface types:

i. Parameter interfaces: Data passed from one method or procedure to another.

ii. Shared memory interfaces: Block of memory is shared between procedures or functions.

iii. Procedural interfaces: Sub-system encapsulates a set of procedures to be called by other sub-systems.

iv. Message passing interfaces: Sub-systems request services from other sub-systems

**Interface errors**

* Interface misuse

i. A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.

* Interface misunderstanding

i. A calling component embeds assumptions about the behavior of the called component which are incorrect.

* Timing errors

i. The called and the calling component operate at different speeds and out-of-date information is accessed.

**Interface testing guidelines**

* Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
* Always test pointer parameters with null pointers.
* Design tests which cause the component to fail.

Use stress testing in message passing systems.

* In shared memory systems, vary the order in which components are activated.

**System testing**

* System testing during development involves integrating components to create a version of the system and then testing the integrated system.
* The focus in system testing is testing the interactions between components.
* System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
* System testing tests the emergent behavior of a system.

**System and component testing**

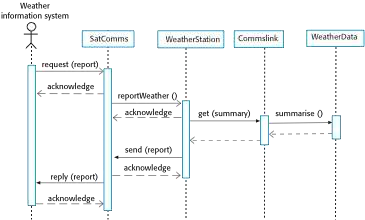
* During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
* Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.

i. In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

**Use-case testing**

* The use-cases developed to identify system interactions can be used as a basis for system testing.
* Each use case usually involves several system components so testing the use case forces these interactions to occur.
* The sequence diagrams associated with the use case documents the components and interactions that are being tested.

Figure 1.6: *Collect weather data sequence chart*



**Testing policies**

* Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
* Examples of testing policies:

i. All system functions that are accessed through menus should be tested.

ii. Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.

iii. Where user input is provided, all functions must be tested with both correct and incorrect input.

**Test-driven development**

* Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
* Tests are written before code and ‘passing’ the tests is the critical driver of development.
* You develop code incrementally, along with a test for that increment. You don’t move on to the next increment until the code that you have developed passes its test.
* TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

**Test-Driven Development (TDD)**

Process Activities:

* Identify a small, implementable increment of functionality.
* Write an automated test for this functionality.
* Execute the test along with all others (initially fails).
* Implement the functionality and rerun the test.
* Move on to the next chunk of functionality.

**Benefits:**

* Code Coverage: Every code segment has at least one associated test.
* Regression Testing: Regression test suite developed incrementally.
* Simplified Debugging: Failed tests pinpoint the problem.
* System Documentation: Tests serve as documentation describing code behavior.

**Regression Testing**

Definition:

* Testing the system to ensure changes haven't broken previously working code.
* Automated testing simplifies rerunning all tests after every change.

**Release Testing**

Purpose:

* Test a release of a system for use outside the development team.
* Convince the supplier it's good enough for use.

Characteristics:

* Focus on functionality, performance, and dependability.
* Black-box testing using system specifications.

**Release Testing vs. System Testing**

Differences:

* Separate team for release testing, not involved in development.
* System testing by the development team focuses on discovering bugs (defect testing).
* Release testing aims to check if the system meets requirements and is ready for external use (validation testing).

**Requirements-Based Testing**

Definition:

* Examining each requirement and developing a test for it.
* Example from MHC-PMS requirements provided.

**Features Tested by Scenario**

* Various features tested, such as authentication, record retrieval, and links with the drugs database.

**Usage Scenario for MHC-PMS**

* Scenario involving a nurse, Kate, using the MHC-PMS for home visits, encryption, record retrieval, and uploading.

**Performance Testing**

* Part of release testing.
* Involves testing emergent properties like performance and reliability.
* Tests should reflect the system's usage profile.
* Stress testing intentionally overloads the system to test failure behavior.

**User Testing**

Types:

* Alpha Testing:

Users work with the development team at the developer's site.

* Beta Testing:

Release made available for users to experiment and raise problems.

* Acceptance Testing:

Customers test the system to decide its readiness for deployment.

**Acceptance Testing Process Stages**

1. Define acceptance criteria.

2. Plan acceptance testing.

3. Derive acceptance tests.

4. Run acceptance tests.

5. Negotiate test results.

6. Reject/accept the system.

**Agile Methods and Acceptance Testing**

* Users/customers are part of the development team.
* Users define tests, integrated with other tests, no separate acceptance testing process.

**Key Points**

* Break the software using experience and guidelines for effective test case selection.
* Prefer automated tests whenever possible.
* Test-first development involves writing tests before the code.
* Scenario testing uses typical usage scenarios to derive test cases.
* Acceptance testing determines if software is ready for deployment and use in its operational environment.

**CHAPTER 9**

**Software Change**

* Inevitable due to new requirements, business environment changes, errors, new hardware, and performance improvements.
* Organizations face challenges in implementing and managing change in existing software systems.

**Importance of Evolution**

* Organizations heavily invest in software systems, critical business assets.
* Regular changes and updates are necessary to maintain the value of these assets.
* Majority of software budget in large companies is allocated to evolving existing software rather than developing new.

**Evolution Processes**

Evolution:

* Ongoing changes in a software system's operational use to implement new requirements.

Servicing:

* Changes to keep the software operational (bug fixes and adaptations) with no new functionality.

Phase-out:

* Software still in use but no further changes.

**Evolution Processes (continued)**

* Depend on software type, development processes, and the skills of those involved.
* Change proposals drive system evolution.
* Identification and evolution processes continue throughout the system's lifetime.

**Change Implementation**

* Iterative development process for designing, implementing, and testing revisions.
* Initial stage may involve program understanding, especially if original developers are not involved.

**Urgent Change Requests**

* Implemented without going through all software engineering stages in emergencies (e.g., system faults, unexpected effects from environment changes, urgent business changes).
* Emergency repair process diagram provided.

**Agile Methods and Evolution**

* Agile methods support incremental development, making the transition from development to evolution seamless.
* Automated regression testing is valuable during system changes.
* Changes expressed as additional user stories.

**Handover Problems**

* Issues when the development team uses agile, but the evolution team prefers plan-based (or vice versa).
* Problems may arise due to documentation expectations, testing approaches, and code complexity.

**Program Evolution Dynamics**

* Study of the processes of system change.
* Lehman and Belady proposed 'laws' applicable to large systems developed by large organizations.
* Laws are more like sensible observations, and their applicability to other software systems is uncertain.

**Lehman’s Laws**

* Continuing Change:

Programs in a real-world environment must change to remain useful.

* Increasing Complexity:

Evolving programs tend to become more complex; extra resources are needed for preservation and simplification.

* Large Program Evolution:

System attributes like size, time between releases, and reported errors are approximately invariant for each release.

* Organizational Stability:

Over a program’s lifetime, the rate of development is approximately constant.

* Conservation of Familiarity:

Incremental change in each release is approximately constant.

* Continuing Growth:

Functionality must continually increase to maintain user satisfaction.

* Declining Quality:

Quality declines without modification to reflect changes in the operational environment.

* Feedback System:

Evolution processes incorporate feedback systems for significant product improvement.

**Applicability of Lehman’s Laws**

* Generally applicable to large, tailored systems developed by large organizations.
* Uncertain modification for shrink-wrapped software, systems with COTS components, small organizations, and medium-sized systems.

**Key Points**

* Software development and evolution are integrated, iterative processes represented by a spiral model.
* For custom systems, maintenance costs usually exceed development costs.
* Software evolution is driven by change requests, involving change impact analysis, release planning, and change implementation.
* Lehman’s laws describe insights from long-term studies of system evolution.

**CHAPTER 9**

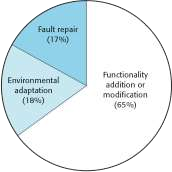
**Software Maintenance- Lecture 2**

* Modifying a program after deployment, typically for custom software.
* Generic software products evolve to create new versions.
* Maintenance involves modifying existing components and adding new ones without major architectural changes.
* Types: fault repair, adaptation to a different environment, and adding/modifying functionality.

**Maintenance Effort Distribution**

- Diagram provided, showing how maintenance efforts are distributed.

Figure 1.1: *Maintenance effort distribution*



**Maintenance Cost**

* Usually greater than development costs (2\* to 100\* depending on the application).
* Affected by both technical and non-technical factors.
* Increases with software maintenance, making further maintenance difficult.
* Ageing software can have high support costs (e.g. old languages, compilers etc.)

**Maintenance Cost Factors**

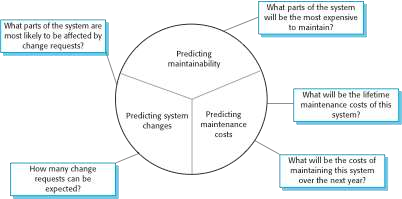
* Team stability, contractual responsibility, staff skills, program age, and structure.
* Maintenance staff often inexperienced and lack domain knowledge.
* As programs age, their structure degrades, making them harder to understand and change.

**Maintenance Prediction**

* Concerned with assessing which parts of the system may cause problems and have high maintenance costs.
* Change acceptance depends on maintainability.
* Implementing changes degrades the system, reducing maintainability.
* Maintenance costs depend on the number of changes, and costs of change depend on maintainability.

**Maintenance Prediction (continued)**

Figure 1.2: *Maintenance Prediction*



**Change Prediction**

* Predicting the number of changes requires an understanding of the system-environment relationship.
* Tightly coupled systems require changes whenever the environment changes.
* Influencing factors: number and complexity of system interfaces, volatile requirements, and business processes.

**Complexity Metrics**

* Predicting maintainability by assessing component complexity.
* Most maintenance effort spent on a small number of components.
* Complexity depends on control structures, data structures, object/method/module size.

**Process Metrics**

* Used to assess maintainability.
* Metrics include requests for corrective maintenance, impact analysis time, implementation time, and outstanding change requests.
* Increasing metrics may indicate a decline in maintainability.

**System Re-engineering**

* Restructuring or rewriting part or all of a legacy system without changing functionality.
* Applicable when some but not all subsystems require frequent maintenance.
* Involves adding effort to make subsystems easier to maintain.

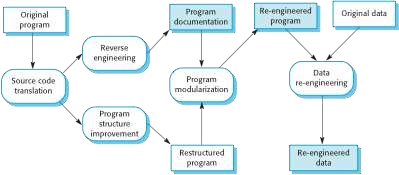
**Advantages of Reengineering**

* Reduced risk compared to new software development.
* Lower cost compared to developing new software.

**Reengineering Process Activities**

* Source code translation, reverse engineering, program structure improvement, program modularization, and data reengineering.

Figure 1.3: *The reengineering process*



**Reengineering Approaches**

* Different approaches based on source code transformation, reverse engineering, and program transformation.

**Reengineering Cost Factors**

* Quality of software to be reengineered, tool support availability, extent of required data conversion, and availability of expert staff.

**Preventative Maintenance by Refactoring**

* Refactoring is making improvements to a program to slow down degradation through change.
* Considered as 'preventative maintenance' to reduce future change problems.
* Involves modifying a program to enhance structure, reduce complexity, and improve understandability.
* Concentrates on program improvement without adding functionality.

**Refactoring and Reengineering**

* Reengineering occurs after a system has been maintained, using automated tools to create a more maintainable system.
* Refactoring is continuous improvement throughout development, avoiding structure and code degradation.

**'Bad Smells' in Program Code**

Identifies issues in program code:

* Duplicate code
* Long methods
* Switch (case) statements
* Data clumping
* Speculative generality
* Strategies to address these issues are provided.

**Legacy System Management**

* Strategies for organizations relying on legacy systems:

1. Scrap the system
2. Continue maintaining
3. Transform through reengineering
4. Replace with a new system

* Strategy depends on system quality and business value.

**Legacy System Categories**

Categories based on quality and business value:

* Low quality, low business value (scrap)
* Low quality, high business value (reengineer/replace)
* High quality, low business value (replace/scrap/maintain)
* High quality, high business value (continue maintenance)

**Business Value Assessment**

* Assessment considers different viewpoints from stakeholders.
* Issues in assessment include system use, supported business processes, system dependability, and system outputs.

**System Quality Assessment**

* Assessment includes business process assessment, environment assessment, and application assessment.
* Business process assessment considers process model, adaptation, relationships, and legacy software support.
* Environmental assessment factors include supplier stability, failure rate, age, performance, support requirements, maintenance costs, interoperability, understandability, documentation, data, and programming language.
* Quantitative data may be collected to assess system quality.

**Key Points**

* Three types of software maintenance: bug fixing, adapting to a new environment, and implementing new/changed requirements.
* Software reengineering involves restructuring and re-documenting to improve understandability.
* Refactoring, a form of preventative maintenance, involves making changes while preserving functionality.
* Assessing the business value and application quality helps decide whether to replace, transform, or maintain a legacy system.