USN

Internal Assessment Test 3 – July 2021

Sub:	Software Testing				Sub Code:	18CS62/17 CS62	Branch:	ISE	
Date:	04/08/2021	Duration:	90 min's	Max Marks: 50	Sem/Sec:	VI A, B&C			OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1a)	What are self check oracles? Compare self check oracles with comparison based oracles	[2+4]	CO4	L1/L2
1b)	Write short notes on Capture and Replay technique used in test execution	[4]	CO4	L2

1a) What are self check oracles? Compare self check oracles with comparison based oracles

Definition of Self Check Oracles [2 marks]

- **Software that applies a pass/fail criterion to a program execution is called a test oracle, often shortened to oracle**, The oracle can be incorporated into the program under test, so that it checks its own work.
- Typically these self-checks are in the form of assertions, similar to assertions used in symbolic execution and program verification, but designed to be checked during execution.
- **[4 marks] for comparison**
- A test case with a comparison-based oracle relies on predicted output that is either precomputed as part oracle of the test case specification or can be derived in some way independent of the program under test.
- Pre-computing expected test results is reasonable for a small number of relatively simple test cases, and is still preferable to manual inspection of program results because the expense of producing (and debugging) predicted results is incurred once and amortized over many executions of the test case.
- Support for comparison-based test oracles is often included in a test harness program or testing framework.
- A harness typically takes two inputs: (1) the input to the program under test (or can be mechanically transformed to a well-formed input), and (2) the predicted output.
- Frameworks for writing test cases as program code likewise provide support for comparison-based oracles.

1b) Write short notes on Capture and Replay technique used in test execution

Minimum 8 points = 4 marks

- The first time such a test case is executed, the oracle function is carried out by a human, and the interaction sequence is captured. Provided the execution was judged (by the human tester) to be correct, the captured log now forms an (input, predicted output) pair for subsequent automated retesting.
- The savings from automated retesting with a captured log depends on how many build-and-test cycles we can continue to use it in, before it is invalidated by some change to the program.
- Distinguishing between significant and insignificant variations from predicted behavior, in order to prolong the effective lifetime of a captured log, is a major challenge for capture/replay testing.
- Capturing events at a more abstract level suppresses insignificant changes. For example, if we log only the actual pixels of windows and menus, then changing even a typeface or background color can invalidate an entire suite of execution logs.
- Mapping from concrete state to an abstract model of interaction sequences

is sometimes possible but is generally quite limited.

- A more fruitful approach is capturing input and output behavior at multiple levels of abstraction within the implementation.
- We have noted the usefulness of a layer in which abstract input events (e.g., selection of an object) are captured in place of concrete events (left mouse button depressed with mouse positioned at 235, 718). Typically, there is a similar abstract layer in graphical output, and much of the capture/replay testing can work at this level.

2a)	Explain about the following basic principles of Testing Process Framework. i) Sensitivity ii) Restriction	[5]	CO5	L2

Minimum 10 points (5 points in each)= 5 marks

- Human developers make errors, producing faults in software. Faults may lead to failures, but **faulty software may not fail on every execution.**
- **The sensitivity principle states that it is better to fail every time than sometimes**. Consider the cost of detecting and repairing a software fault. If it is detected immediately(e.g., by an on-the-fly syntactic check in a design editor), then the **cost of correction isverysmall,andinfactthelinebetweenfaultpreventionandfaultdetectionisblurred**.
- If a fault is detected in **inspection or unit testing**, the cost is still relatively small. If afault survives initial detection efforts at the unit level, but triggers a **failure detected in integration testing, the cost of correction is much greater.** If the first failure is detected in system or acceptance testing, the cost is very high indeed, and the most costly faults are those detected by customers in the field.
- **A fault that triggers a failure on every execution is unlikely to survive past unit testing. A characteristic of faults that escape detection until much later is that they trigger failures only rarely, or in combination with circumstances that seem unrelated or are difficult to control.**
- For example, a **fault that results in a failure only for some unusual configurations of customer equipment may be difficult and expensive to detect.**
- A fault that results in a failure randomly but very rarely - for example, a race condition that only occasionally causes data corruption - may likewise escape detection until the software is in use by thousands of customers, and even then be difficult to diagnose and correct.
- Run-time array bounds checking in many programming languages (including Java but not Cor C++)is an example of **the sensitivity principle applied at the language level.**
- **A variety of tools and replacements for the standard memory management library are available to enhance sensitivity to memory allocation and reference faults in CandC++. The fail-fast property of Java iterators is another application of the sensitivity principle.**

Restriction

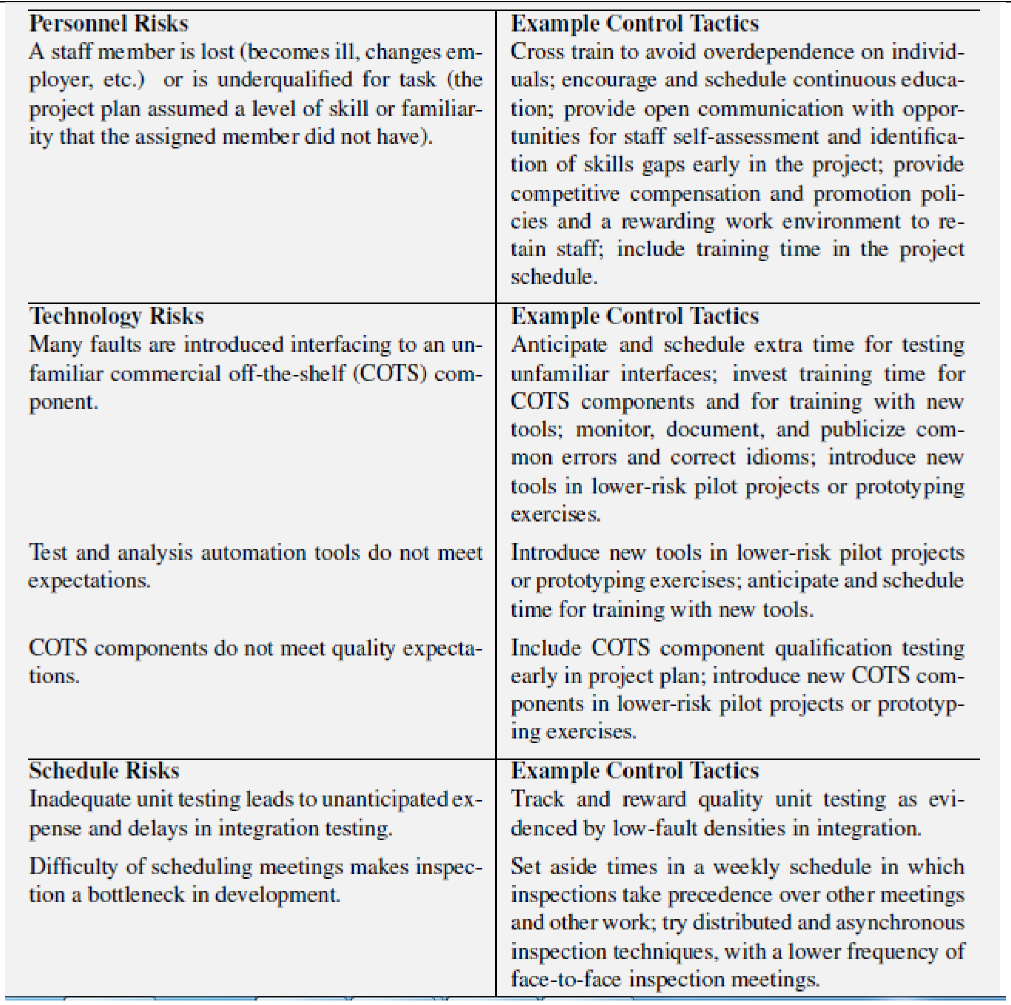
- When there are no acceptably cheap and effective ways to check a property, sometimes one **can change the problem by checking a different, more restrictive property or by limiting the check to a**

	smaller, more restrictive class of programs. Consider the problem of ensuring that each variable is initialized before it is used, one very execution. Simple as the property is, it is not possible for a compiler or **analysis tool to precisely determine whether it holds.**. Additional restrictions may be imposed in the form of programming standards (e.g., restricting the use of type casts or pointer arithmetic in C), or by tools in a developmentenvironment.Otherformsof**restrictioncanapplytoarchitecturalanddetaileddesign**. Consider, for example, the problem of ensuring that a transaction consisting of a sequence of accesses to a complex data structure by one process appears to the outside world as if it had occurred atomically, rather than inter leaved with transactions of other processes. This property is **called serializability**: The end result of a set of such transactions should appear as if they were applied in some serial order, even if they didn't.			
2b)	What is Scaffolding? What are the components of scaffolding? Differentiate Generic versus Specific scaffolding **Scaffolding[1 mark]** • **Code developed to facilitate testing is called scaffolding, by analogy to the temporary structures erected around a building during construction or maintenance.** **components of scaffolding [2 mark]** • Scaffolding may include test drivers (substituting for a main or calling program), test harnesses (substituting for parts of the deployment environment), and stubs (substituting for functionality called or used by the software under test), in addition to program instrumentation and support for recording and managing test execution. **Generic versus Specific scaffolding [2 mark]**	[1+2+2]	CO4	L1/L2

• common driver code into reusable modules.	• hundreds or thousands of such test-specific drivers, on the other hand, may be cumbersome and a disincentive to thorough testing.
• wide of generic scaffolding support can be used across class of applications.	• Used only to particular applications
• typically includes, in addition to a standard interface for executing a set of test cases, basic support for logging test execution and results.	

	• generic scaffolding may suffice for small numbers of hand-written test cases	For large application we go for specific scaffolding			

3(a)	Explain in detail about the Risk management in terms of process and quality management. List out the various risks and their control tactics in both. • Risk is an inevitable part of every project, and so risk planning must be a part of every plan. • Risks cannot be eliminated, but they can be assessed, controlled, and monitored. • The duration of integration, system, and acceptance test execution depends to a **large extent on the quality of software under test**. Software that is sloppily constructed or that **undergoes inadequate analysis and test before commitment to the code base** will slow testing progress. • Even if responsibility for **diagnosing test failures lies with developers** and not with the testing group, a **test execution session that results in many failures and generates many failure reports is inherently more time consuming** than executing a suite of tests with few or no failures. • This **schedule vulnerability is yet another reason to emphasize earlier activities,** in particular those that **provide early indications of quality problems**. **Inspection of design and code (with quality team participation) can help control this risk,** and also serves to communicate quality standards and best practices among the team. **Risk Management in the Quality Plan: Risks Generic to Process Management** • The quality plan must identify potential risks and define appropriate control tactics. Some risks and control tactics are generic to process management, while others are specific to the quality process. • Here we provide a brief overview of some risks generic to process management. Risks specific to the quality process are summarized in the sidebar on page 391.	[8]	CO4	L2

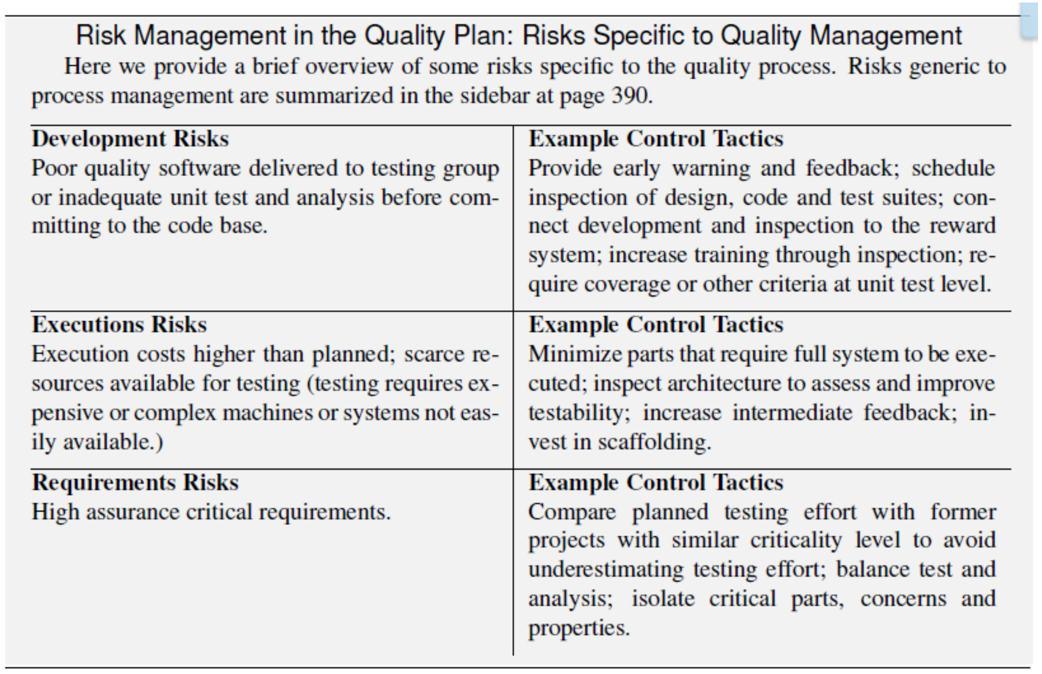


Personnel Risks	Example Control Tactics
A staff member is lost (becomes ill, changes employer, etc.) or is underqualified for task (the project plan assumed a level of skill or familiarity that the assigned member did not have).	Cross train to avoid overdependence on individuals; encourage and schedule continuous education; provide open communication with opportunities for staff self-assessment and identification of skills gaps early in the project; provide competitive compensation and promotion policies and a rewarding work environment to retain staff; include training time in the project schedule.
Technology Risks	**Example Control Tactics**
Many faults are introduced interfacing to an unfamiliar commercial off-the-shelf (COTS) component.	Anticipate and schedule extra time for testing unfamiliar interfaces; invest training time for COTS components and for training with new tools; monitor, document, and publicize common errors and correct idioms; introduce new tools in lower-risk pilot projects or prototyping exercises.
Test and analysis automation tools do not meet expectations.	Introduce new tools in lower-risk pilot projects or prototyping exercises; anticipate and schedule time for training with new tools.
COTS components do not meet quality expectations.	Include COTS component qualification testing early in project plan; introduce new COTS components in lower-risk pilot projects or prototyping exercises.
Schedule Risks	**Example Control Tactics**
Inadequate unit testing leads to unanticipated expense and delays in integration testing.	Track and reward quality unit testing as evidenced by low-fault densities in integration.
Difficulty of scheduling meetings makes inspection a bottleneck in development.	Set aside times in a weekly schedule in which inspections take precedence over other meetings and other work; try distributed and asynchronous inspection techniques, with a lower frequency of face-to-face inspection meetings.

•

Risk Management in the Quality Plan: Risks Specific to Quality Management

- Here we provide a brief overview of some risks specific to the quality process. Risks generic to process management are summarized in the sidebar at page 390.



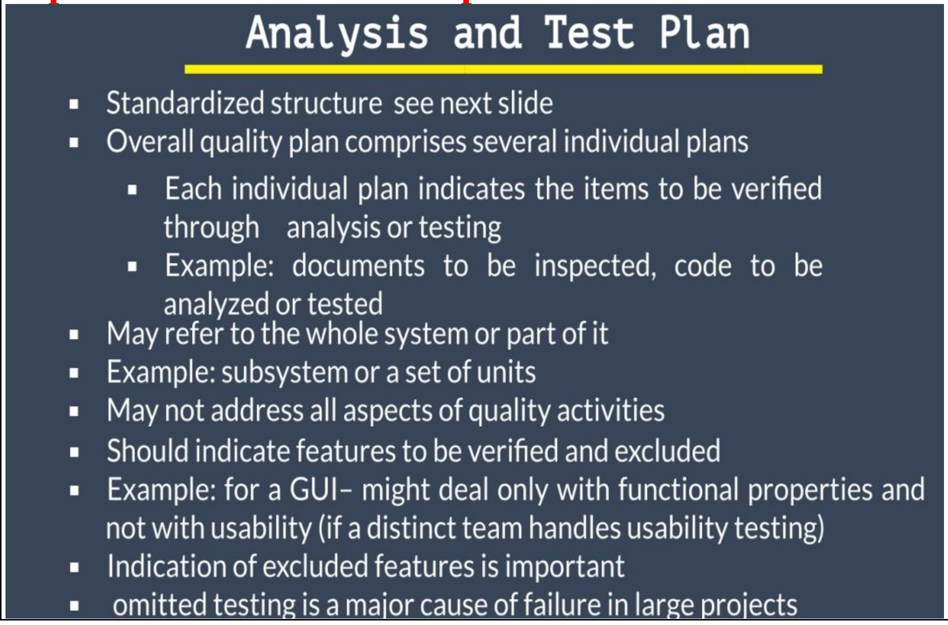
Risk Management in the Quality Plan: Risks Specific to Quality Management

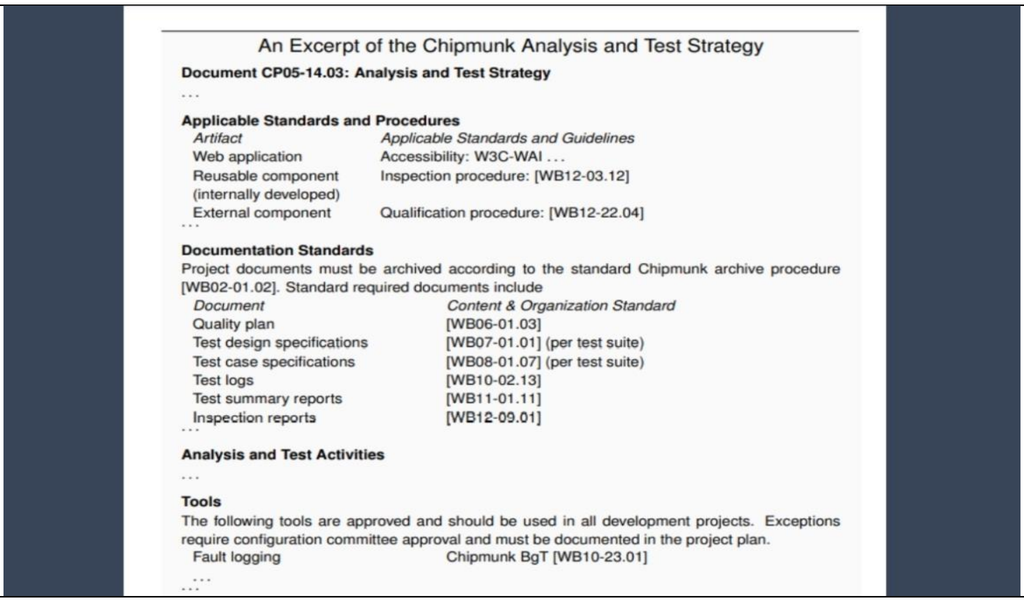
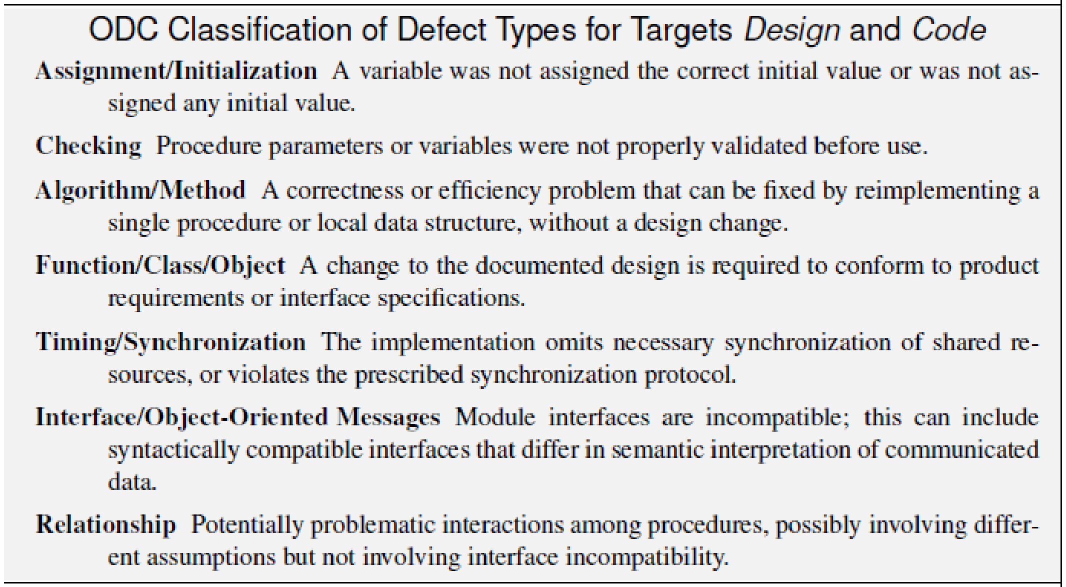
Here we provide a brief overview of some risks specific to the quality process. Risks generic to process management are summarized in the sidebar at page 390.

Development Risks	Example Control Tactics
Poor quality software delivered to testing group or inadequate unit test and analysis before committing to the code base.	Provide early warning and feedback; schedule inspection of design, code and test suites; connect development and inspection to the reward system; increase training through inspection; require coverage or other criteria at unit test level.
Executions Risks	**Example Control Tactics**
Execution costs higher than planned; scarce resources available for testing (testing requires expensive or complex machines or systems not easily available.)	Minimize parts that require full system to be executed; inspect architecture to assess and improve testability; increase intermediate feedback; invest in scaffolding.
Requirements Risks	**Example Control Tactics**
High assurance critical requirements.	Compare planned testing effort with former projects with similar criticality level to avoid underestimating testing effort; balance test and analysis; isolate critical parts, concerns and properties.

3(b)	Define the software testing principle: **Visibility**	[2]	CO4	L1

	[2 marks] • Visibility means the ability to **measure progress or status against goals**. In software engineering, one encounters the visibility principle mainly in the **form of process visibility, and then mainly in the form of schedule visibility: ability to judge the state of development against a project schedule.** • **Quality process visibility** also applies **to measuring achieved (or predicted) quality against quality goals**. The principle of visibility involves **setting goals that can be assessed as well as devising methods to assess their realization.** • Visibility is closely related to **observability, the ability to extract useful information from a software artifact.**			
4	Explain about quality goals and quality team in detail. **Quality Goals [5marks]** • **Process visibility requires a clear specification of goals, and in the case of quality process visibility this includes a careful distinction among dependability qualities**. A team that does not have a clear idea of the difference between reliability and robustness, for example, or of their relative importance in a project, has little chance of attaining either. • Correctness : The degree to which a system is free from [defects] in its specification, design, and implementation. • Robustness : The degree to which a system continues to function in the presence of invalid inputs or stressful environmental conditions. • Reliability : The ability of a system to perform its requested functions under stated conditions whenever required - having a long mean time between failures. • Goals must be further refined into a clear and reasonable set of objectives. If an **organization claims that nothing less than 100% reliability will suffice, it is not setting an ambitious objective.** • The relative **importance of qualities** and their relation to other project objectives varies. **Time-to-market may be the most important property for a mass market product**, **usability may be more prominent for a Web based application**, and **safety may be the overriding requirement for a life-critical system.** • The **external properties of software** can ultimately be divided into dependability (does the software do what it is intended to do?) and usefulness. There is no precise dependability way to distinguish these, but a rule of thumb is that when software is not dependable, we say it has a fault, or a defect, or (most often) a bug, resulting in an undesirable behavior or failure. It is quite possible to build systems that are very reliable, relatively free from usefulness hazards, and completely useless. **Quality Team[5 marks]** • The quality plan must **assign roles and responsibilities to people**. As with other aspects of planning, assignment of responsibility occurs at a **strategic level and a tactical level**. • **The tactical level**, represented directly in the project plan, assigns responsibility to individuals in accordance with the general strategy. It involves balancing level of effort across time and carefully managing personal	[5+5]	CO4	L2

	interactions. • **The strategic level of organization** is represented not only in the quality strategy document, but in the structure of the organization itself. • The strategy for assigning responsibility may be partly driven by external requirements. For example, independent quality teams may be required by certification agencies or by a client organization. • Additional objectives include ensuring **sufficient accountability that** quality tasks are not easily overlooked; • An **independent and autonomous testing team** lies at one end of the spectrum of possible team organizations. **One can make that team organizationally independent so that,** for example, a project manager with schedule pressures can neither bypass quality activities or standards, nor reallocate people from testing to development, nor postpone quality activities until too late in the project. • Separating **quality roles from development roles minimizes the risk of conflict between roles played by an individual,** and thus makes most sense for roles in which independence is paramount, such as final system and acceptance testing. • **The more development and quality roles are combined and intermixed, the more important it is to build into the plan checks and balances to be certain that quality** activities • **Separate roles do not necessarily imply segregation of quality activities to distinct individuals.** • **Outsourcing test and analysis activities is sometimes motivated by the perception that testing is less technically demanding than development and can be carried out by lower-paid and lower-skilled individuals.** • Outsourcing can be a **reasonable approach when its objectives are not merely minimizing cost, but maximizing independence** • **with mixed roles requires special attention to avoid the conflicts between roles played by an individual**, • **The plan must clearly define milestones and delivery for outsourced activities, as well as checks on the quality of delivery in both directions**			
5(a)	Discuss about Analysis and Test Plan document in detail **Explanation 3 marks example 2 marks**  Analysis and Test Plan ▪ Standardized structure see next slide ▪ Overall quality plan comprises several individual plans ▪ Each individual plan indicates the items to be verified through analysis or testing ▪ Example: documents to be inspected, code to be analyzed or tested ▪ May refer to the whole system or part of it ▪ Example: subsystem or a set of units ▪ May not address all aspects of quality activities ▪ Should indicate features to be verified and excluded ▪ Example: for a GUI– might deal only with functional properties and not with usability (if a distinct team handles usability testing) ▪ Indication of excluded features is important ▪ omitted testing is a major cause of failure in large projects	[5]	CO5	L2

	 An Excerpt of the Chipmunk Analysis and Test Strategy **Document CP05-14.03: Analysis and Test Strategy** . . . **Applicable Standards and Procedures** *Artifact* — *Applicable Standards and Guidelines* Web application — Accessibility: W3C-WAI . . . Reusable component (internally developed) — Inspection procedure: [WB12-03.12] External component — Qualification procedure: [WB12-22.04] . . . **Documentation Standards** Project documents must be archived according to the standard Chipmunk archive procedure [WB02-01.02]. Standard required documents include *Document* — *Content & Organization Standard* Quality plan — [WB06-01.03] Test design specifications — [WB07-01.01] (per test suite) Test case specifications — [WB08-01.07] (per test suite) Test logs — [WB10-02.13] Test summary reports — [WB11-01.11] Inspection reports — [WB12-09.01] . . . **Analysis and Test Activities** . . . **Tools** The following tools are approved and should be used in all development projects. Exceptions require configuration committee approval and must be documented in the project plan. Fault logging — Chipmunk BgT [WB10-23.01] . . .			
5(b)	List and explain design and code defects **At least 5 defects +explanation 5 marks**  ODC Classification of Defect Types for Targets *Design* and *Code* **Assignment/Initialization** A variable was not assigned the correct initial value or was not assigned any initial value. **Checking** Procedure parameters or variables were not properly validated before use. **Algorithm/Method** A correctness or efficiency problem that can be fixed by reimplementing a single procedure or local data structure, without a design change. **Function/Class/Object** A change to the documented design is required to conform to product requirements or interface specifications. **Timing/Synchronization** The implementation omits necessary synchronization of shared resources, or violates the prescribed synchronization protocol. **Interface/Object-Oriented Messages** Module interfaces are incompatible; this can include syntactically compatible interfaces that differ in semantic interpretation of communicated data. **Relationship** Potentially problematic interactions among procedures, possibly involving different assumptions but not involving interface incompatibility.	[5]	CO4	L2
6(a)	Explain about various dependability properties in testing process framework with diagram Correctness, Reliability, availability, Meantime between Failure, Safety, Robustness should be explained properly. **[6 marks definition + 2 marks explanation]** **Correctness:** • A program or system is correct if it is consistent with its specification. By definition, a specification divides all possible system behaviours into two classes, successes (or correct executions) and failures. All of the possible behaviors of a correct system are successes. • **Reliability is a statistical approximation to correctness, in the sense that 100% reliability is indistinguishable from correctness.** Roughly speaking, reliability is a measure of the likelihood of correct function for some "unit" of behavior, which could be a single use or program execution or a period of time.	[8]	CO4	L2

- **Availability** is an appropriate measure when a failure has some duration in time.
- **Mean time between failures (MTBF) is yet another measure of reliability**, also using time as the unit of execution. The hypothetical network switch that typically fails once in a 24-hour period and takes about an hour to recover has a mean time between failures of 23 hours.
- **Software safety** is an extension of the well-established field of system safety into software. Safety is concerned with **preventing certain undesirable behaviors, called hazards**. It is quite **explicitly not concerned with achieving any useful behavior apart from whatever functionality is needed to prevent hazards.**
- Correctness **and reliability are contingent upon normal operating conditions**. It is not reasonable to expect a word processing program to save changes normally when the file does not fit in storage, or to expect a database to continue to operate normally when the computer loses power, or to expect a Web site to provide completely satisfactory service to all visitors when the load is 100 times greater than the maximum for which it was designed.
- Software that fails under these conditions, which violate the premises of its design, may still be "correct" in the strict sense, yet the manner in which the software fails is important.
- It is acceptable that the word processor fails to write the new file that does not fit on disk, but unacceptable to also corrupt the previous version of the file in the attempt.
- It is acceptable for the database system to cease to function when the power is cut, but unacceptable for it to leave the database in a corrupt state. And it is usually preferable for the Web system to **turn away some arriving users rather than becoming too slow for** all, or crashing.
- Software that gracefully degrades or fails "softly" outside its normal operating parameters is robust.
- Software safety is a kind of robustness, but robustness is a more general notion that concerns not only avoidance of hazards (e.g., data corruption) but also partial functionality under unusual situations. Robustness, like safety, begins with explicit consideration of unusual and undesirable situations, and should include augmenting software specifications with appropriate responses to undesirable events

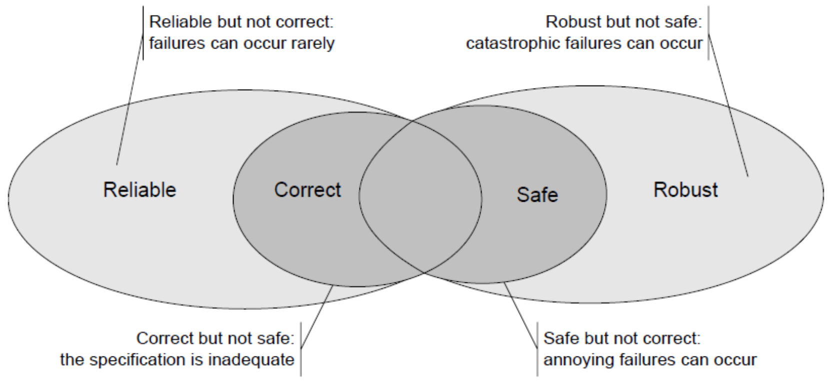
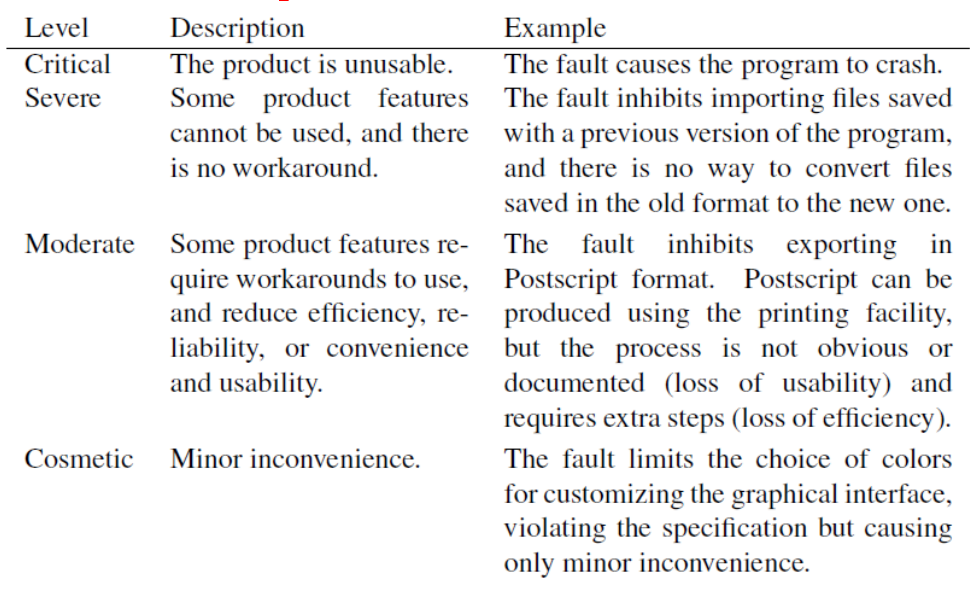

Reliable but not correct:
failures can occur rarely
Robust but not safe:
catastrophic failures can occur
Reliable
Correct
Safe
Robust
Correct but not safe:
the specification is inadequate
Safe but not correct:
annoying failures can occur

Figure 4.1: Relation among dependability properties

6(b)	List the various types of faults with examples **4 faults with example 4 marks**	[2]	CO4	L2



Level	Description	Example
Critical	The product is unusable.	The fault causes the program to crash.
Severe	Some product features cannot be used, and there is no workaround.	The fault inhibits importing files saved with a previous version of the program, and there is no way to convert files saved in the old format to the new one.
Moderate	Some product features require workarounds to use, and reduce efficiency, reliability, or convenience and usability.	The fault inhibits exporting in Postscript format. Postscript can be produced using the printing facility, but the process is not obvious or documented (loss of usability) and requires extra steps (loss of efficiency).
Cosmetic	Minor inconvenience.	The fault limits the choice of colors for customizing the graphical interface, violating the specification but causing only minor inconvenience.

Table 20.1: Standard severity levels for root cause analysis (RCA).