USN | | | | | | | | | | |

Internal Assessment Test 3 – July 2022

Sub:	Software Testing-Scheme and Solutions					Sub Code:	18IS62	Branch:	ISE	
Date:	8/07/2022	Duration:	90 min's	Max Marks:	50	Sem/Sec:	VI C			OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1	Write a short note on i) Quality and Process ii) Test and analysis iii) Risk planning iv) Monitoring the process v) Improving the process.	[10]	CO5	L1

Quality and Process [2 marks]
Explanation:2 marks
Test and analysis [2 marks]
Test:1 mark
Analysis: 1 mark
Risk planning [2 marks]
Explanation:2 marks
Monitoring the process [2 marks]
Explanation:2 marks
Improving the process [2 marks]
Explanation:2 marks

Quality and process

- One can identify particular activities and responsibilities in a software development process that are focused primarily on ensuring adequate dependability of the software product, much as one can identify other activities and responsibilities concerned primarily with project schedule or with product usability.
- It is convenient to group these quality assurance activities under the rubric "quality process," although we must also recognize that quality is intertwined with and inseparable from other facets of the overall process.
- Like other parts of an overall software process, the quality process provides a framework for selecting and arranging activities aimed at a particular goal, while also considering interactions and trade-offs with other important goals.
- All software development activities reflect constraints and trade-offs, and quality activities are no exception.

Test

- Despite the attractiveness of automated static analyses when they are applicable, and despite the usefulness of manual inspections for a variety of documents including but not limited to program source code, **dynamic testing remains a dominant technique.**
- Dynamic testing is really divided into several distinct activities that may occur at different points in a project.

Analysis

- Analysis techniques that do not involve actual execution of program source code play a prominent role in overall software quality processes.

	• Manual inspection techniques and automated analyses can be applied at any development stage. They are particularly well suited at the early stages of specifications and design, where the lack of executability of many intermediate artifacts reduces the efficacy of testing. **Risk planning** • 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. **Monitoring the process** • The quality manager monitors progress of quality activities**, including results as well as schedule, to identify deviations from the quality plan as early as possible and take corrective action.** • Effective monitoring, naturally, depends on a plan that is realistic, well organized, and sufficiently detailed with clear, unambiguous milestones and criteria. We say a process is visible to the extent that it can be effectively monitored. • Successful completion of a planned activity must be distinguished from mere termination, as otherwise it is too tempting to meet an impending deadline by omitting some planned work. **Improving the process** • While the assembly-line, mass production industrial model is inappropriate for software, which is at least partly custom-built, there is almost always some commonality among projects undertaken by an organization over time. • The quality process, as well as the software development process as a whole, can be improved by gathering, analyzing, and acting on data regarding faults and failures. • The goal of quality process improvement is to find cost-effective countermeasures for classes of faults that are expensive because they occur frequently, or because the failures they cause are expensive, or because, once detected, they are expensive to repair.			
2	Explain about quality goals and quality team in detail. **Quality goals explanation: 5Marks** **Quality team explanation: 5Marks** **Quality goals** • **Process visibility requires a clear specification of goals, and in the case of quality process visibility this includes a careful distinction among**	[10]	CO4	L2

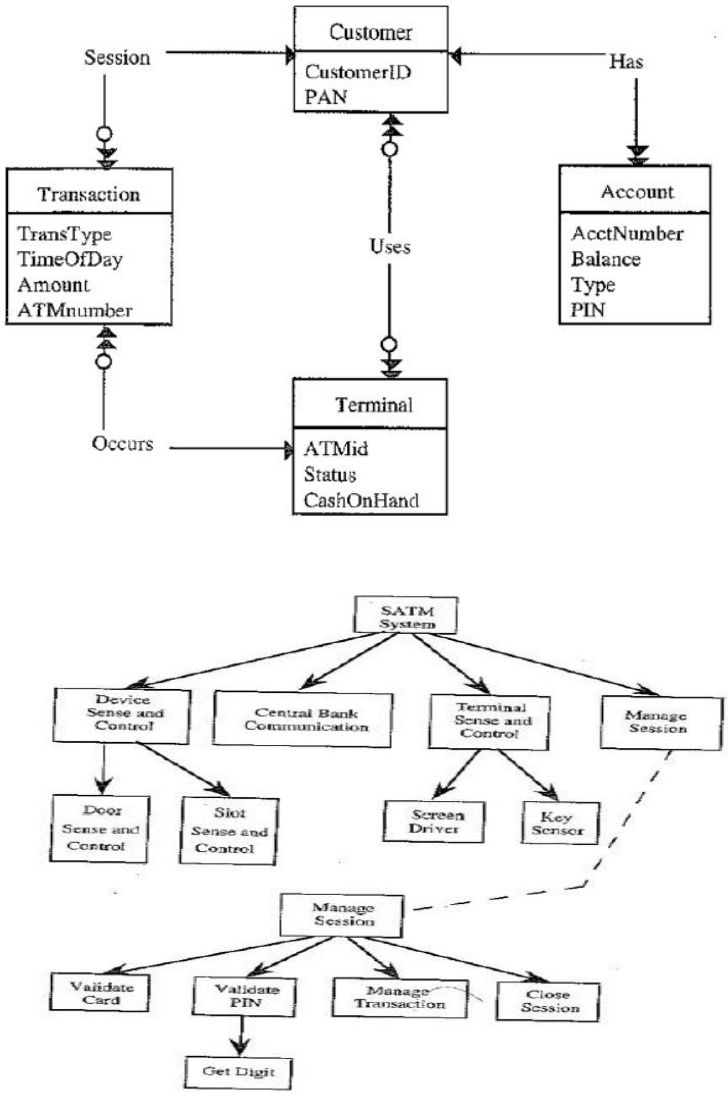
dependability qualities. A team that does not have a clear idea of the difference between reliability and robustness, for example, or of their relative importance in a project, has little chance of attaining either.

- **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.**
- Rather, it is setting no objective at all, and choosing not to make reasoned trade-off decisions or to balance limited resources across various activities. It is, in effect, abrogating responsibility for effective quality planning, and leaving trade-offs among cost, schedule, and quality to an arbitrary, ad hoc decision based on deadline and budget alone.
- 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.**

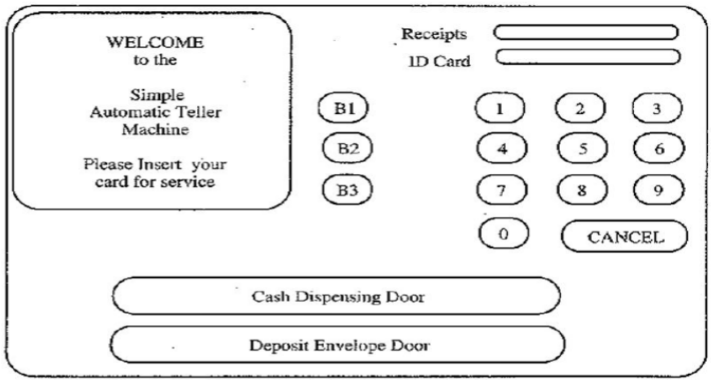
Quality team

- 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 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; encouraging objective judgment of quality and preventing it from being subverted by schedule pressure; fostering shared commitment to quality among all team members; and developing and communicating shared knowledge and values regarding quality.
- Each of the possible organizations of quality roles makes some objectives easier to achieve and some more challenging. Conflict of one kind or another is inevitable, and therefore in organizing the team it is important to recognize the conflicts and take measures to control adverse consequences. If an individual plays two roles in potential conflict (e.g., a developer responsible for delivering a unit on schedule is also responsible for integration testing that could reveal faults that delay delivery), there must be countermeasures to control the risks inherent in that conflict.

3a)	Discuss about Analysis and Test Plan document in detail. **Analysis [2.5 marks]** **Explanation:2.5 marks** **Test plan [2.5 marks]** **Explanation:2.5 marks** **Analysis** • Analysis techniques that do not involve actual execution of program source code play a prominent role in overall software quality processes. • Manual inspection techniques and automated analyses can be applied at any development stage. They are particularly well suited at the early stages of specifications and design, where the lack of executability of many intermediate artifacts reduces the efficacy of testing. **Test plan** • Inspection, in particular, can be applied to essentially any document including requirements documents, architectural and more detailed design documents, test plans and test cases, and of course program source code. • Inspection may also have secondary benefits, such as spreading good practices and instilling shared standards of quality. • On the other hand, inspection takes a considerable amount of time and requires meetings, which can become a scheduling bottleneck. .	[5]	CO4	L2
3b)	Explain about the following basic principles of Testing Process Framework i) Sensitivity ii) Restriction. **Sensitivity: 2.5 Marks** **Restriction: 2.5 Marks** **Sensitivity** • 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 is very small, and in fact the line between fault prevention and fault detection is blurred**. • If a fault is detected in **inspection or unit testing**, the cost is still relatively small. If a fault 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. **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.**	[5]	CO4	L2

	• Java's solution to this problem is to enforce a stricter, simpler condition: A program is not permitted to **have any syntactic control paths on which an uninitialized reference could occur, regardless of whether those paths could actually be executed.**			
4	Draw the context diagram of the SATM system and explain the same. **Explanation: 5Marks** **Diagram: 5Marks** **Context diagram:**  	[10]	CO5	L1

SATM terminal



Explanation:

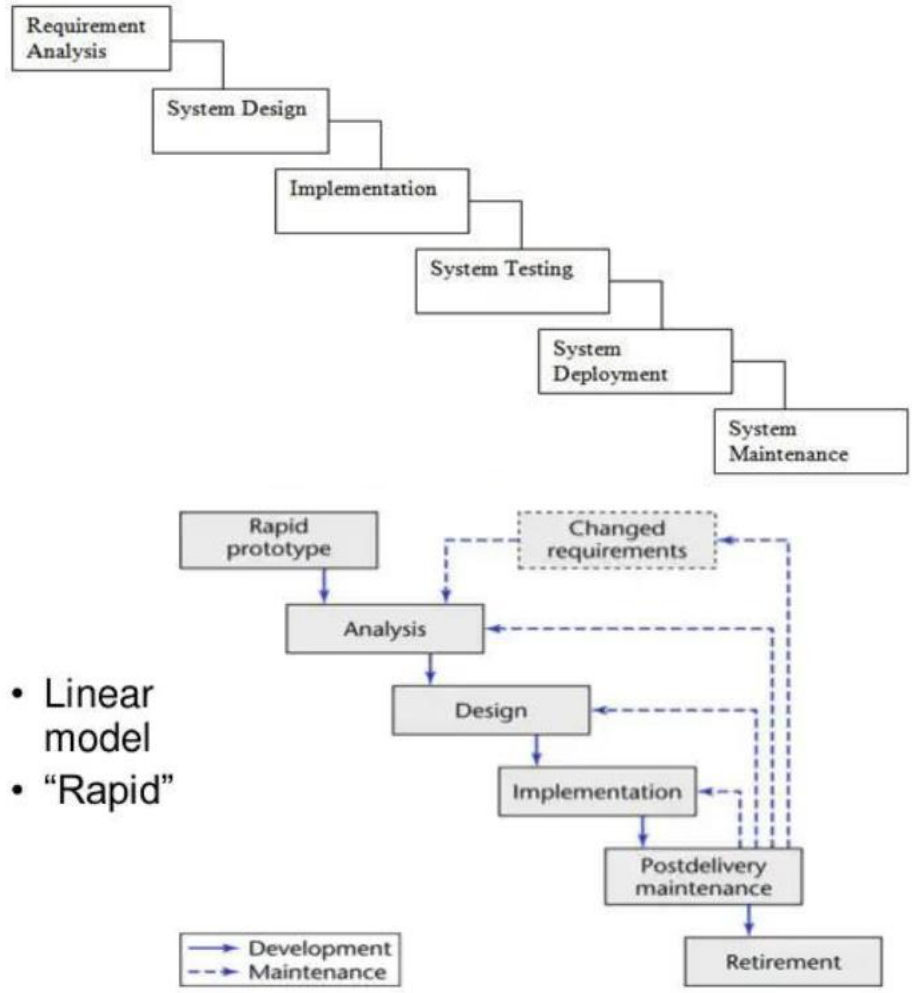
- Entity/Relationship diagram of the major data structures in the SATM system: Customers, Accounts, Terminals, and Transactions. Good data modeling practice dictates postulating an entity for each portion of the system that is described by data that is retained (and used by functional components). Among the data the system would need for each customer are the customer's identification and personal account number (PAN); these are encoded into the magnetic strip on the customer's ATM card. We would also want to know information about a customer's account(s), including the account numbers, the balances, the type of account (savings or checking), and the Personal Identification Number (PIN) of the account.

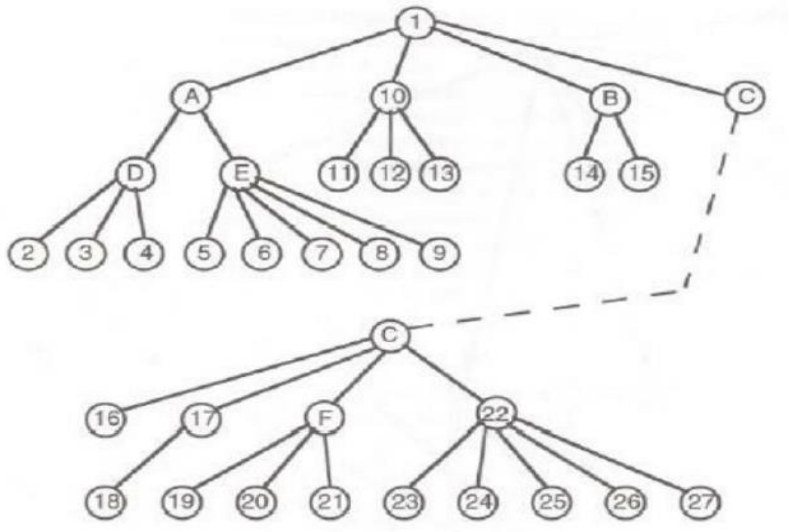
Upper Level SATM Finite State Machine

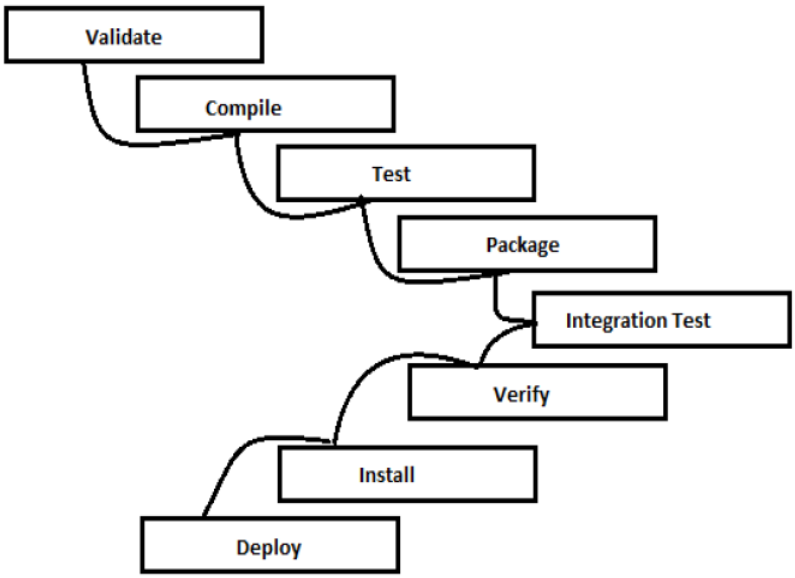
- The upper level finite state machine in Figure divides the system into states that correspond to stages of customer usage. Other choices are possible, for instance, we might choose states to be screens being displayed (this turns out to be a poor choice). Finite state machines can be hierarchically decomposed in much the same way as dataflow diagrams. The decomposition of the Await PIN state is shown in Figure. In both of these figures, state transitions are caused either by events at the ATM terminal (such as a keystroke) or by data conditions (such as the recognition that a PIN is correct). When a transition occurs, a corresponding action may also occur. We choose to use screen displays as such actions; this choice will prove to be very handy when we develop system level test cases.

PIN Entry Finite State Machine

- If we only use a structure chart to guide integration testing, we miss the fact that some (typically lower level) functions are used in more than one place. Here, for example, the Screen Driver function is used by several other modules, but it only appears once in the functional decomposition. In the next chapter, we will see that a "call graph" is a much better basis for integration test case identification. We can develop the beginnings of such a call graph from a more detailed view of portions of the system. To support this, we need a numbered decomposition, and a more detailed view of two of the components.

5	Explain the traditional view of testing levels of waterfall-life cycle with a neat diagram and rapid prototyping life cycles. **Explanation: 5Marks** **Diagram: 5Marks** **Diagram:**  **Explanation:** • The Waterfall Life Cycle Of the three traditional levels of testing (unit, integration, and system), unit testing is best understood. The testing theory and techniques we worked through in Parts I and II are directly applicable to unit testing. System testing is understood better than integration testing, but both need clarification. The bottom-up approach sheds some insight: test the individual components, and then integrate these into subsystems until the entire system is tested. System testing should be something that the customer (or user) understands, and it often borders on customer acceptance testing. Generally, system testing is functional rather than structural; this is mostly due to the absence of a structural basis for system test cases. In the traditional view, integration testing is what's left over: it's not unit testing, and it's not system testing. Most of the usual discussions on integration testing center on the order in which units are integrated: top-down, bottom-up, or the "big bang" (everything at once). • Rapid prototyping has interesting implications for system testing. To use the	[10]	CO5	L1

	prototyping cycle(s) as information gathering activities, and then produce a requirements specification in a more traditional manner. Another possibility is to capture what the customer does with the prototype(s), define these as scenarios that are important to the customer, and then use these as system test cases. The main contribution of rapid prototyping is that it brings the operational (or behavioural) viewpoint to the requirements specification phase. Usually, requirements specification techniques emphasize the structure of a system, not its behavior. This is unfortunate, because most customers don't care about the structure, and they do care about the behavior. • Executable specifications are an extension of the rapid prototyping concept. With this approach, the requirements are specified in an executable format (such as finite state machines or Petri nets). The customer then executes the specification to observe the intended system behavior and provides feedback as in the rapid prototyping model.			
6a)	Explain the decomposition based integration with an example. **Explanation: 2.5 Marks** **Diagram: 2.5 Marks** **Explanation:** Integration testing only consider integration testing based on the functional decomposition of the system being tested. These approaches are all based on the functional decomposition, expressed either as a tree (Figure) or in textual form. These discussions inevitably center on the order in which modules are to be integrated. There are four choices: from the top of the tree downward (top down), from the bottom of the tree upward (bottom up), some combination of these (sandwich), or most graphically, none of these (the big bang). All of these integration orders presume that the units have been separately tested, thus the goal of decomposition based integration is to test the interfaces among separately tested units. **Diagram:**  	[5]	CO4	L2

6b)	Explain alternative life cycle models. **Explanation: 2.5 Marks** **Diagram: 2.5 Marks** **Explanation:** • Practitioners have devised alternatives in response to shortcomings of the traditional waterfall model of software development. Common to all of these alternatives is the shift away from the functional decomposition to an emphasis on composition. Decomposition is a perfect fit both to the top-down progression of the waterfall model and to the bottom-up testing order. One of the major weaknesses of waterfall development is the over-reliance on this whole paradigm. Functional decomposition can only be well done when the system is completely understood, and it promotes analysis to the near exclusion of synthesis. The result is a very long separation between requirements specification and a completed system, and during this interval, there is no opportunity for feedback from the customer. Composition, on the other hand, is closer the way people work start with something known and understood, then add to it gradually, and maybe remove undesired portions. • There are three mainline derivatives of the waterfall model: incremental development, evolutionary development, and the Spiral model. Each of these involves a series of increments or builds. Within a build, the normal waterfall phases from detailed design through testing occur, with one important difference: system testing is split into two steps, regression, and progression testing **Diagram:**  	[5]	CO4	L2