**Name Muhammad Ali Khan**

**Reg No 16550**

| |
|---|
| **Course Name** |
| Software Design & Architecture |
| **Instructor** |
| Aasma Khan |

**Question No: 01**

a) **What is Software Architecture? Why is software architecture design so important?**

**Software architecture** refers to the fundamental structures of a software system and the discipline of creating such structures and systems. Each structure comprises software elements, relations among them, and properties of both elements and relations.The *architecture* of a software system is a metaphor, analogous to the architecture of a building.It functions as a blueprint for the system and the developing project, laying out the tasks necessary to be executed by the design teams.

Software architecture is about making fundamental structural choices that are costly to change once implemented. Software architecture choices include specific structural options from possibilities in the design of the software. For example, the systems that controlled the Space Shuttle launch vehicle had the requirement of being very fast and very reliable. Therefore, an appropriate real-time computing language would need to be chosen. Additionally, to satisfy the need for reliability the choice could be made to have multiple redundant and independently produced copies of the program, and to run these copies on independent hardware while cross-checking results.

===========================================================================

**Why is software architecture design so important?**

A software architecture is the foundation of a software system. The design of the architecture is significant to the quality and long-term success of the software. A proper design determines whether the requirements and quality attributes can be satisfied.

There are a number of reasons why a good software architecture design is critical to building useful software.

- Software architecture design is when key decisions are made regarding the architecture.
- Avoiding design decisions can incur technical debt.
- A software architecture design communicates the architecture to others.
- The design provides guidance to the developers.
- The impact of the software architecture design is not limited to technical concerns. It also influences the non-technical parts of the project.

**Some of the other goals are as follows −**

Expose the structure of the system, but hide its implementation details.

Realize all the use-cases and scenarios.

Try to address the requirements of various stakeholders.

Handle both functional and quality requirements.

Reduce the goal of ownership and improve the organization’s market position.

Improve quality and functionality offered by the system.

Improve external confidence in either the organization or system.

=================================================================

**b) Explain any four tasks of architect.**

**Role of Software Architect**

A Software Architect provides a solution that the technical team can create and design for the entire application. A software architect should have expertise in the following areas −

**Design Expertise**

Expert in software design, including diverse methods and approaches such as object-oriented design, event-driven design, etc.

Lead the development team and coordinate the development efforts for the integrity of the design.

Should be able to review design proposals and tradeoff among themselves.

**Domain Expertise**

Expert on the system being developed and plan for software evolution.

Assist in the requirement investigation process, assuring completeness and consistency.

Coordinate the definition of domain model for the system being developed.

**Technology Expertise**

Expert on available technologies that helps in the implementation of the system.

Coordinate the selection of programming language, framework, platforms, databases, etc.

**Methodological Expertise**

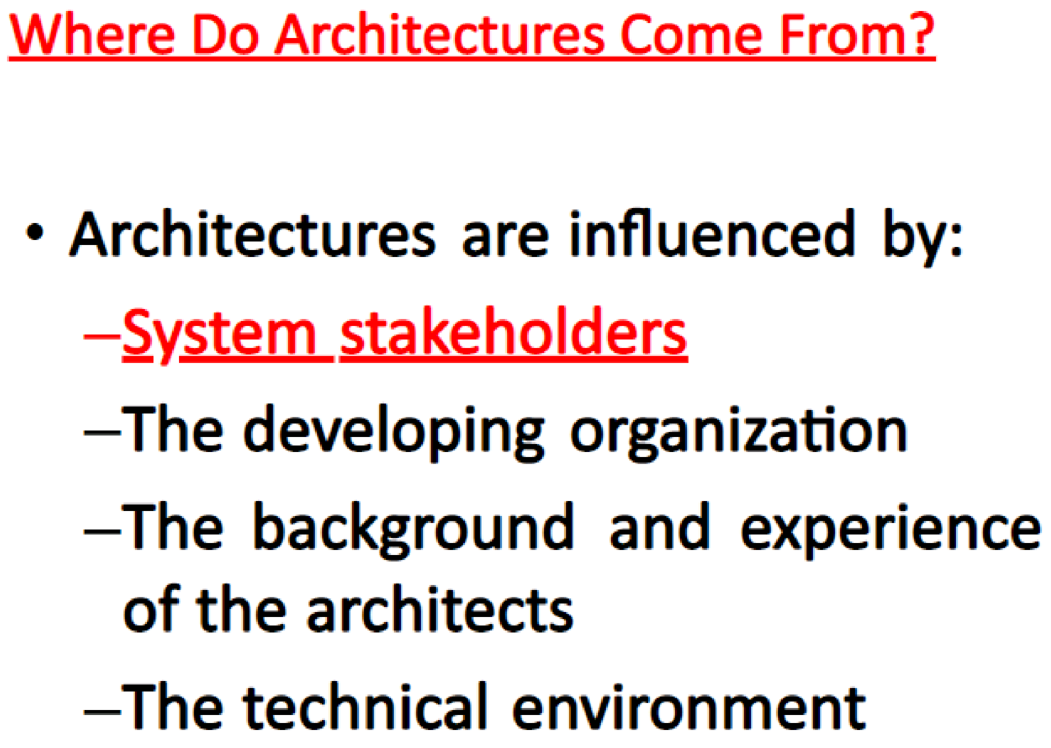
Expert on software development methodologies that may be adopted during SDLC (Software Development Life Cycle).

**Question No: 02**

Explain Architecture Business Cycle (ABC) in detail with figure.

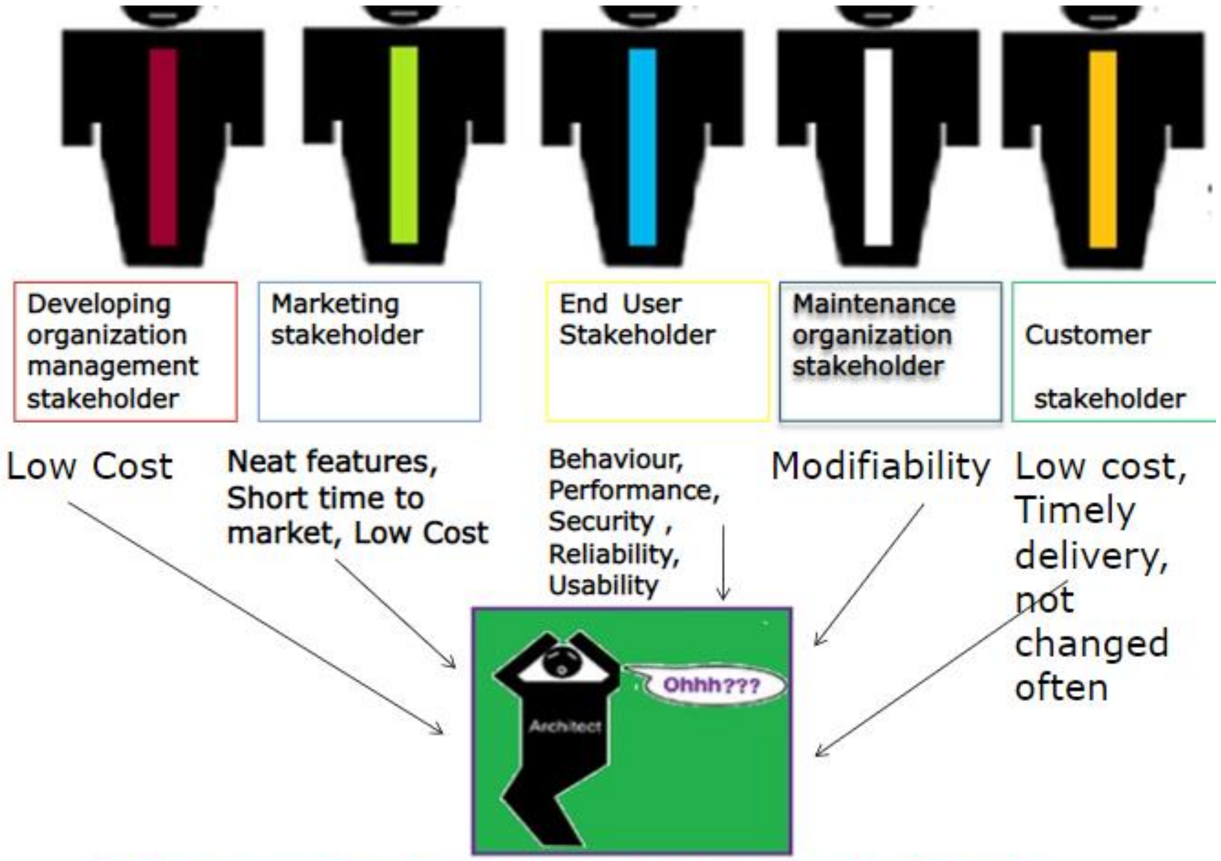
Architecture Business Cycle (ABC)

- Software architecture is a result of technical, business and social influences.
- These are in turn affected by the software architecture itself.
- This cycle of influences from the environment to the architecture and back to the environment is called the *Architecture Business Cycle (ABC)*.



## Where Do Architectures Come From?

- Architectures are influenced by:
  - System stakeholders
  - The developing organization
  - The background and experience of the architects
  - The technical environment


Developing organization management stakeholder
Marketing stakeholder
End User Stakeholder
Maintenance organization stakeholder
Customer stakeholder
Low Cost
Neat features, Short time to market, Low Cost
Behaviour, Performance, Security , Reliability, Usability
Modifiability
Low cost, Timely delivery, not changed often
Architect
Ohhh???

<u>**Influence of system stakeholders on Architect**</u>

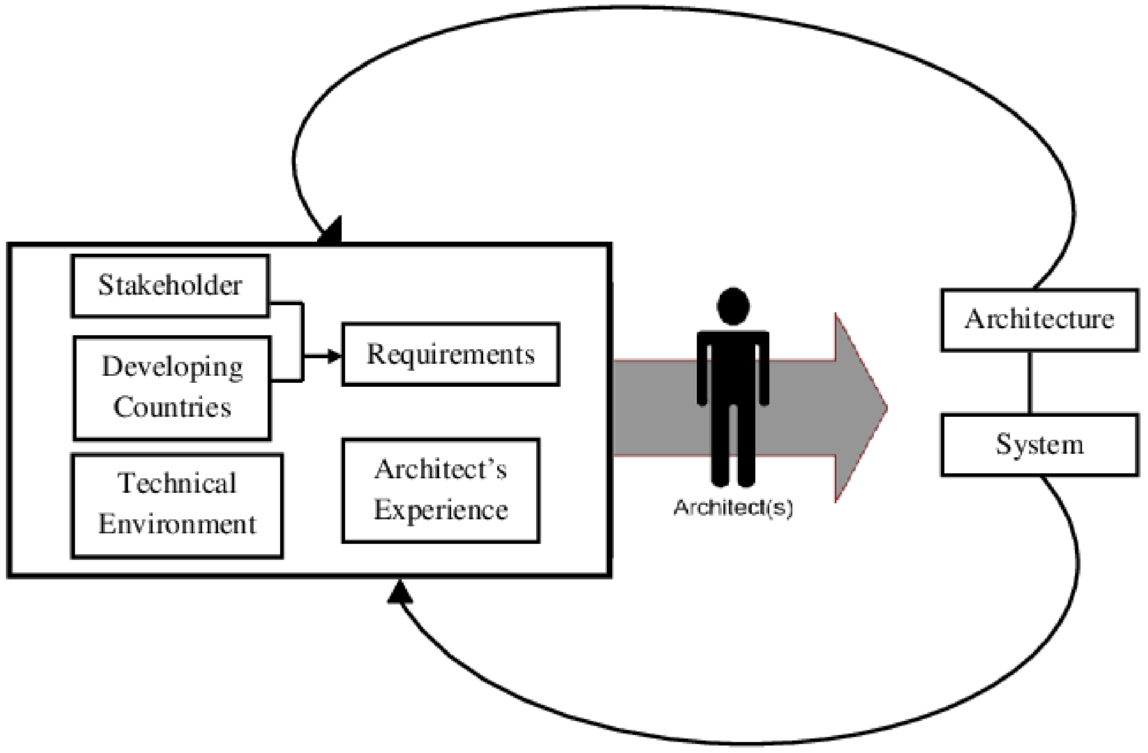
www.bookspar.com | Website for students | VTU NOTES |

## ARCHITECTURES INFLUENCED BY SYSTEM STAKEHOLDERS

- Acceptable system involves properties such as performance, reliability, availability, memory utilization, security, modifiability, usability, interoperability with other system.

- The underlying problem, is that each stakeholder has different concerns and goals, some of which may be contradictory.

- The reality is that the architect often has to fill in the blanks and mediate the conflicts.



==================================================================

**Question No: 03**

**Explain ABC Activities?**

## ARCHITECTURE ACTIVITIES

As indicated in the structure of the ABC, architecture activities have comprehensive feedback relationships with each other. We will briefly introduce each activity in the following subsections.

### Creating the Business Case for the System

Creating a business case is broader than simply assessing the market need for a system. It is an important step in creating and constraining any future requirements. How much should the product cost? What is its targeted market? What is its targeted time to market? Will it need to interface with other systems? Are there system limitations that it must work within?

These are all questions that must involve the system's architects. They cannot be decided solely by an architect, but if an architect is not consulted in the creation of the business case, it may be impossible to achieve the business goals.

### Understanding the Requirements

There are a variety of techniques for eliciting requirements from the stakeholders. For example, object-oriented analysis uses scenarios, or "use cases" to embody requirements. Safety-critical systems use more rigorous approaches, such as finite-state-machine models or formal specification languages. In (Understanding Quality Attributes), we introduce a collection of quality attribute scenarios that support the capture of quality requirements for a system.

One fundamental decision with respect to the system being built is the extent to which it is a variation on other systems that have been constructed. Since it is a rare system these days that is not similar to other systems, requirements elicitation techniques extensively involve understanding these prior systems' characteristics. We discuss the architectural implications of product lines in (Software Product Lines: Re-using Architectural Assets).

Another technique that helps us understand requirements is the creation of prototypes. Prototypes may help to model desired behavior, design the user interface, or analyze resource utilization. This helps to make the system "real" in the eyes of its stakeholders and can quickly catalyze decisions on the system's design and the design of its user interface.

Regardless of the technique used to elicit the requirements, the desired qualities of the system to be constructed determine the shape of its architecture. Specific tactics have long been used by architects to achieve particular quality attributes. We discuss many of these tactics in  (Achieving Qualities). An architectural design embodies many tradeoffs, and not all of these tradeoffs are apparent when specifying requirements. It is not until the architecture is created that some tradeoffs among requirements become apparent and force a decision on requirement priorities.

**Creating or Selecting the Architecture**

In the landmark book *The Mythical Man-Month*, Fred Brooks argues forcefully and eloquently that conceptual integrity is the key to sound system design and that conceptual integrity can only be had by a small number of minds coming together to design the system's architecture.  (Achieving Qualities) and  (Designing the Architecture) show how to create an architecture to achieve its behavioral and quality requirements.

**Communicating the Architecture**

For the architecture to be effective as the backbone of the project's design, it must be communicated clearly and unambiguously to all of the stakeholders. Developers must understand the work assignments it requires of them, testers must understand the task structure it imposes on them, management must understand the scheduling implications it suggests, and so forth. Toward this end, the architecture's documentation should be informative, unambiguous, and readable by many people with varied backgrounds. We discuss the documentation of architectures in (Documenting Software Architectures).

## Analyzing or Evaluating the Architecture

In any design process there will be multiple candidate designs considered. Some will be rejected immediately. Others will contend for primacy. Choosing among these competing designs in a rational way is one of the architect's greatest challenges. The chapters in Part Three (Analyzing an Architecture) describe methods for making such choices.

Evaluating an architecture for the qualities that it supports is essential to ensuring that the system constructed from that architecture satisfies its stakeholders' needs. Becoming more widespread are analysis techniques to evaluate the quality attributes that an architecture imparts to a system. Scenario-based techniques provide one of the most general and effective approaches for evaluating an architecture. The most mature methodological approach is found in the Architecture Tradeoff Analysis Method (ATAM).

## Implementing Based on the Architecture

This activity is concerned with keeping the developers faithful to the structures and interaction protocols constrained by the architecture. Having an explicit and well-communicated architecture is the first step toward ensuring architectural conformance. Having an environment or infrastructure that actively assists developers in creating and maintaining the architecture (as opposed to just the code) is better.

## Ensuring Conformance to an Architecture

Finally, when an architecture is created and used, it goes into a maintenance phase. Constant vigilance is required to ensure that the actual architecture and its representation remain faithful to each other during this phase. Although work in this area is comparatively immature, there has been intense activity in recent years will present the current state of recovering an architecture from an existing system and ensuring that it conforms to the specified architecture.

**Question No 04:**

Pair programming is an agile software development technique in which two programmers work together at one work station. One types in code while the other reviews each line of code as it is typed in. The person typing is called the driver. The person reviewing the code is called the observer. The two programmers switch roles frequently (possibly every 30 minutes or less).

Suppose that you are asked to build a system that allows Remote Pair Programming. That is, the system should allow the driver and the observer to be in remote locations, but both can view a single desktop in real-time. The driver should be able to edit code and the observer should be able to "point" to objects on the driver's desktop. In addition, there should be a video chat facility to allow the programmers to communicate. The system should allow the programmers to easily swap roles and record rationale in the form of video chats. In addition, the driver should be able to issue the system to backup old work.

- Draw a use case diagram to show all the functionality of the system.
- Describe in detail four non-functional requirements for the system.
- Give a prioritized list of design constraints for the system and justify your list and the ordering.
- Propose a set of classes that could be used in your system and present them in a class diagram

**Answer:**

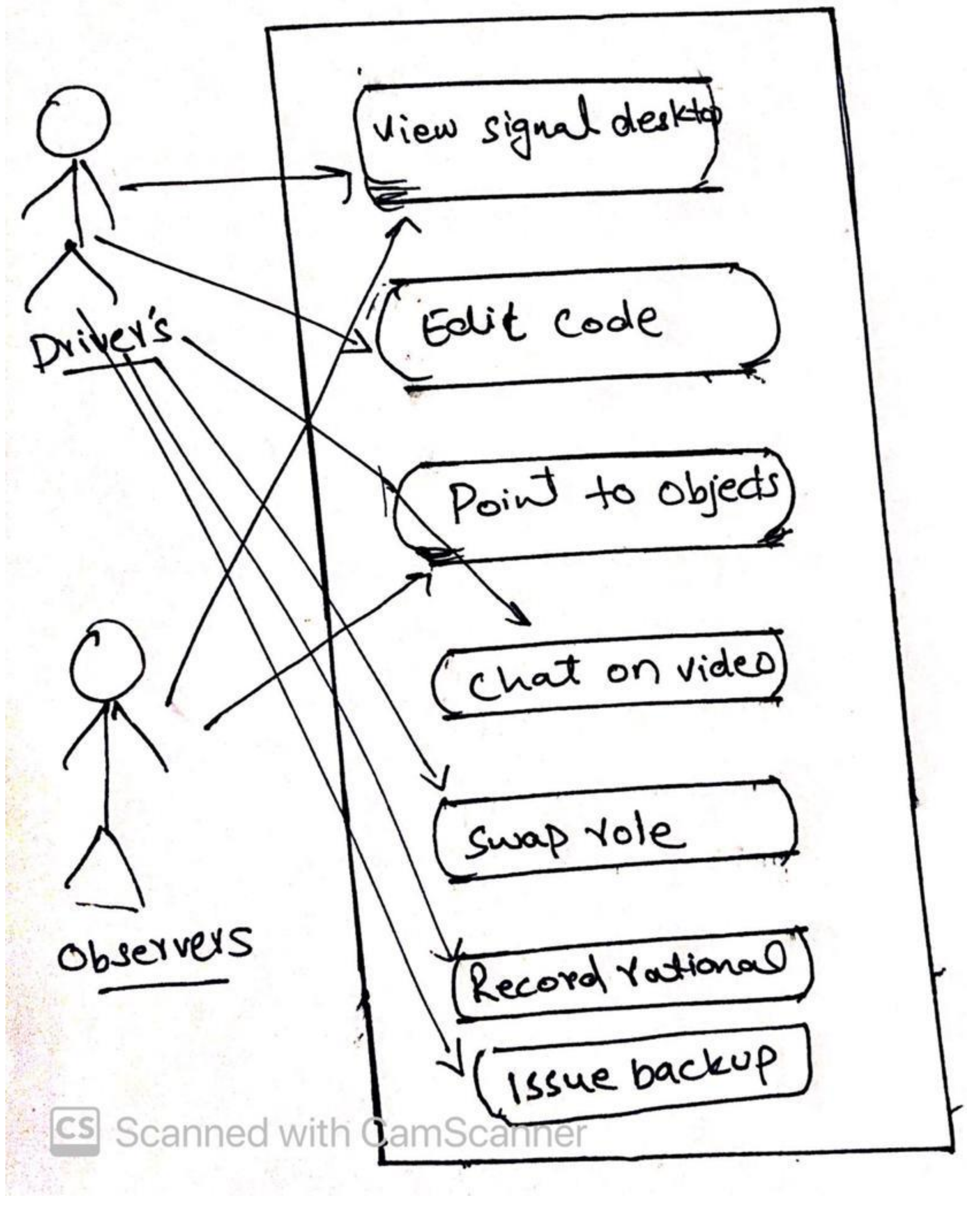
- **Draw a use case diagram to show all the functionality of the system.**

**A: Diagram is Below.**

\

Muhammad Alikhan
Reg# 16550

# Use Case



Scanned with CamScanner

- **Describe in detail four non-functional requirements for the system.**

**A:**

**Ease of use:** A key non-functional requirement is the all-important 'Ease of use'.
It may be easier for a user if they can set up different screen layouts for the different types of work they have to do. So, it is required that the system should allow users to customize the graphical user interface, including menu contents, layout of screens, use of function keys, on-screen colours, fonts and font sizes, and audible alerts. These configuration changes made by the user should be saved in their user-profile.

- **Real-time performance:** The Observer should be able to see the changes made by the Driver immediately without delay; the video chat should be smooth without delay also.

1. Monitor performance in real-time
2. Let customer needs drive collaboration
3. Allocate resources as needed
4. Empower local teams to continuously improve the action plan
5. Structure rewards to encourage collaboration among internal teams
6. Adjust targets to drive optimum performance.

**Availability:**The system should be available to both programmers all the time**.** All the programmers communicate each other any time for the issues so it will be available for 24/7.Availability. And finally, availability describes how likely the system is accessible for a user at a given point in time. While it can be expressed as a probability percentage, you may also define it as a percentage of time the system is accessible for operation during some time period. For instance, the system may be available 98 percent of the time during a month. Availability is perhaps the most business-critical requirement, but to define it, you also must have estimations for reliability and maintainability.

**Portability:**The programmers should be able to use the system regardless of what computer and operating system used by the programmers.***Portability*** defines how a system or its element can be launched on one environment or another. It usually includes hardware, software, or other usage platform specification. Put simply, it establishes how well actions performed via one platform are run on another. Also, it prescribes how well system elements may be accessed and may interact from two different environments. Portability also has an additional aspect called compatibility. ***Compatibility*** defines how a system can co-exist with another system in the same environment. For instance, software installed on an operating system must be compatible with its firewall or antivirus protection. Portability and compatibility are established in terms of operating systems, hardware devices, browsers, software systems, and their versions. For now, a cross-platform, cross-browsing, and mobile-responsive solution is a common standard for web applications.

**Security**:The backup code should be kept securely and be protected from unauthorized access.This non-functional requirement assures that all data inside the system or its part will be protected against malware attacks or unauthorized access. But there's a catch. The lion's share of security non-functional requirements can be translated into concrete functional counterparts. If you want to protect the admin panel from unauthorized access, you would define the login flow and different user roles as system behavior or user actions. So, the non-functional requirements part will set up specific types of threats that functional requirements will address in more detail. But this isn't always the case. If your security relies on specific standards and encryption methods, these standards don't directly describe the behavior of a system.

**Reliability:**The system should be reliable, i.e., it should not crash when the internet speed is slow and when the internet connection is suddenly down the user should be able to resume the session at a later time. This quality attribute specifies how likely the system or its element would run without a failure for a given period of time under predefined conditions. Traditionally, it's expressed as a probability percentage. For instance, if the system has 85 percent reliability for a month, this means that during this month, under normal usage conditions, there's an 85 percent chance that the system won't experience critical failure.

As you may have guessed, it's fairly tricky to define critical failure, time, and normal usage conditions. Another, somewhat simpler approach to that metric is to count the number of critical bugs found in production for some period of time or calculate a mean time to failure. Three ways to measure it are:

- Probability percentage, time;
- The number of critical failures, time; and
- Mean time between failures.

- **Give a prioritized list of design constraints for the system and justify your list and the ordering.**

**A: Design constraints for the system is applied on Nonfunctional Requirement's in Order:**

**Real-time performanceconstraints:**Real-time constraints are restrictions on the timings of events, such that they occur on-time. A system with real-time constraints is called a real-time system. Not merely the performance of such systems, but also their feasibility depends on the satisfaction of real-time constraints. Hard constraints must be satisfied for system correctness, while the violation of soft constraints only degrades a system. Specification of real-time constraints requires either some extensions of programming languages through annotations and logic expressions, or the use of temporal logics or some formal declarative language with temporal constructs. Stringent timing restrictions complicate system design and verification. The time model can be dense or discrete, thus giving different methods for real-time system synthesis and verification. This article surveys the specification, design, and verification of real-time constraints and systems. The most important technique for guaranteeing real-time, namely scheduling, is briefly surveyed. Different system models are presented for handling real-time constraints such as Petri nets, timed automata, process algebra and object-oriented model. Design techniques for real-time hardware systems and for real-time software applications are introduced and discussed. Hardware-software codesign is also introduced. Verification techniques for real-time constraints, such as model checking, are also presented.

**Availabilityconstraints:**You can set up general constraints for availability, a typical use-case is opening hours.Availability-constraints are Time kit's way of defining general rules for availability. As mentioned, opening-hours are a kind of general availability-rule. Timekit defines two different kinds of constraints: "blocking" and "allowing". "Allowing" constraints implicitly blocks everything else, so for instance the constraint "allow day", with the day set to Monday, will block all the other days in the week. This means that you do not need to explicitly block periods that are implicitly being blocked with an "allowing" constraint.

**Resource constraints and general constraints**

Timekit offers two places to define availability-constraints:

On the resource, which could be defining working hours or vacation, constraints that are relevant only for that resource.

On the request to the /availability endpoint (can also be pre-configured through projects), which could be defining a work's opening hours, constraints that are relevant across resources.

**Portabilityconstraint:**

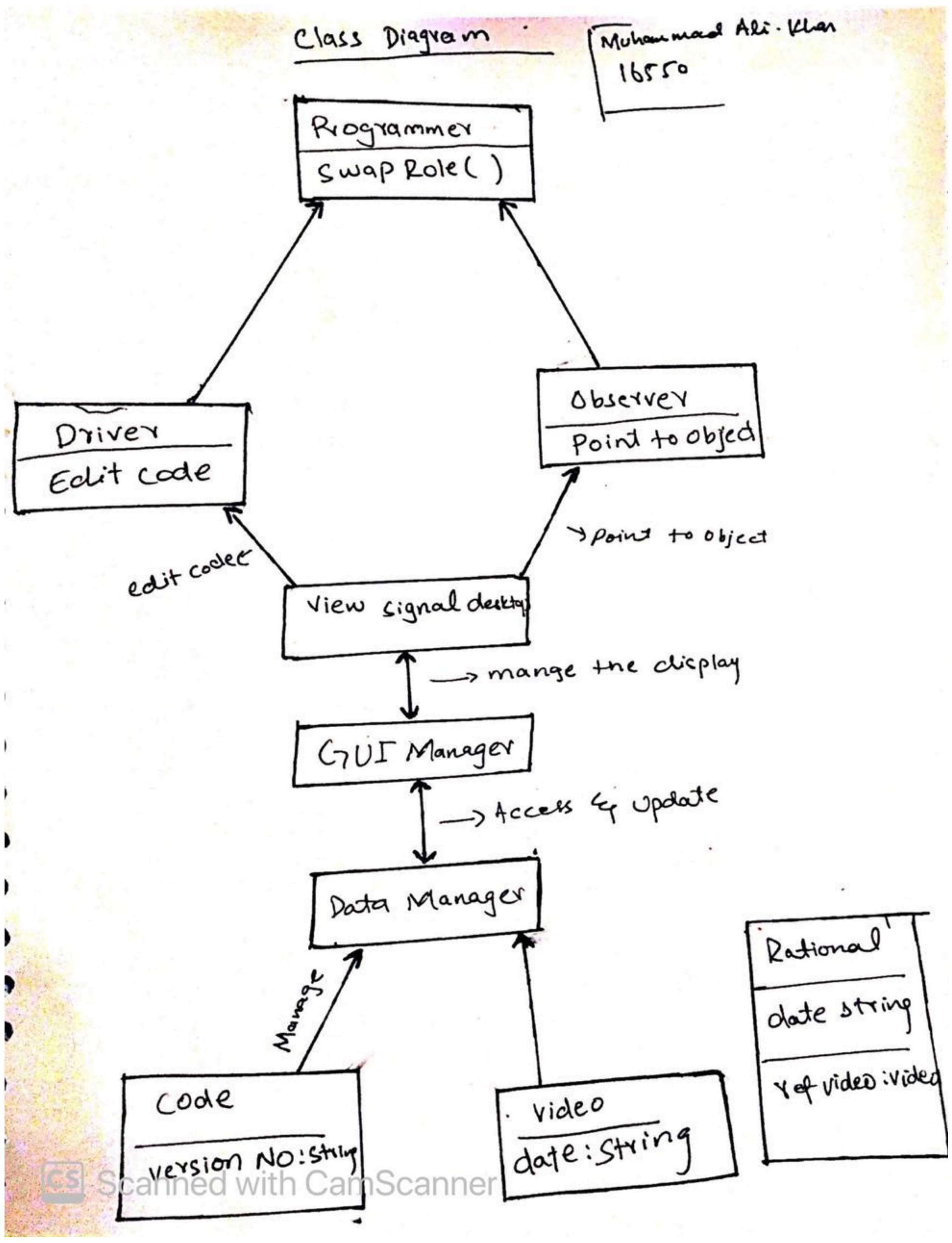
"The system should be portable" is an NFR. This NFR may lead to a constraint on the programming language used for the implementation of the system (e.g., the programming language Java (rather than C and C++) might be preferred in order to meet this NFR.

**Securityconstraints**: The system must be secured" is a NFR. The design constraints could be a user authentication must be in place, the communication protocol must be encrypted, and/or the data must be stored on a server behind firewall.

**Reliabilityconstraints:**The constraints in reliability of the system is only internet if the internet is very slow the reliability of the system will not good and hence the system will not be work properly.

- **Propose a set of classes that could be used in your system and present them in a class diagram**

**A: Class Diagram: diagram is below:**



Scanned with CamScanner

==========================================================