ZQMS-ARC-REC-002

ZIMBABWE OPEN UNIVERSITY
"Empowerment Through Open Learning" ®

ASSIGNMENT COVER

REGION:
____MDLANDS_____
_

PROGRAMME: __BSSE 141_____
INTAKE:__1__

FULL NAME OF STUDENT: _PRIVILEDGE
KURURA_____ PIN:_P244379d_____

MAILING ADDRESS:
_p244379d@students.zou.ac.zw_____
_____

CONTACT TELEPHONE/CELL: _____ ID. NO.: 63-2368569-
p44_____

COURSE NAME: _Introduction to software
engineering_____ COURSE CODE: _____

ASSIGNMENT NO. e.g. 1 or 2: _____1_____ DUE DATE: _____

ASSIGNMENT TITLE: _____

_____

_____

MARKER'S COMMENTS: _____

_____

_____

OVERALL MARK: _____ MARKER'S NAME: _____

MARKER'S SIGNATURE:_____ DATE: _____

Issue Date: 3 October 2013 Revision 0

QUESTION 1

(a) Requirements analysis
Objective: Recognize the needs of the clients or users.
Important Tasks:
To find out what consumers or clients anticipate, speak with them.
What the software should and shouldn't do should be noted down.
Verify if the program can be developed on schedule and within the allocated budget.
Determine which demands are most crucial.
Result: A comprehensive list of the functions of the software.

(b) Design: Produce a software blueprint or plan.
Important Tasks: Create the system's structure (modules, components).
Consider specifics such as databases, data processing, and algorithms.
Select the appropriate technologies and tools for programming.
To test the design concepts, construct models.
Outcome: A comprehensive design blueprint or record.

(c) Coding.
Important Tasks: Write the code using the design plan as a guide.
To maintain code versions, use tools.
To make sure they function, test the code's smaller sections.
Examine the code to make it better.
Result: The software's operational code.

(d) Testing : Verify that the program operates as intended.
Important Tasks:
Make a strategy for the software's testing.
Check individual components, then the entire system.
Make sure everything is user-friendly and functions as a unit.

Resolve any issues or glitches.
Result: A stable, well-tested product.

(e) Delivery: Provide users with the finished product.
Important Tasks: Arrange for the software's installation and release.
Users should receive education and training.
To use the software, install it.
After release, offer assistance with any problems that may arise.
Result: The program has been delivered and is operational.

QUESTION 2

2 (a)

According to Sommerville, 2011,The software development method known as the Spiral Model involves a cycle of planning, designing, constructing, and testing. These processes are repeated repeatedly (referred to as "iterations" or "spirals") in order to progressively enhance the software till completion.

The Spiral Model's Steps:
Planning: Determine hazards, establish objectives, and decide on the best course of action.
Analyze potential risks by considering what could go wrong and how to prevent them.
Design and Build: Produce a prototype or a tiny portion of the software.
Testing and Review: Put the component to the test, gather input, and refine it.

Up until the complete software is finished, the process restarts with fresh objectives, risks, and advancements after every cycle (or spiral).

Positive aspects (benefits) of the spiral model
Risk management is beneficial for large and complicated projects since it focuses on identifying and addressing risks early on.
Flexibility: When needs are uncertain or likely to change, the ability to make changes after each cycle is beneficial.
Customer Input: Following each iteration, customers can provide input to help guarantee their demands are met in the final product.
Gradual Progress: Better management and tracking are made possible by the project's piecemeal construction.
Drawbacks (negative aspects) of the spiral model
Expensive and Time-Granting: The procedure may become more time-consuming

and costly due to the ongoing planning and risk analysis involved in each cycle.
Complex: Handling several iterations can be challenging; competent teams are needed to handle the intricacy.
Not Suitable for Small tasks: For smaller, simpler tasks, it is frequently too detailed and expensive.
Uncertain End Date: It can be challenging to pinpoint the precise completion date of a project because the cycles continue until the software is finished.

2 (b)
Cohesion and coupling are fundamental ideas in software engineering that aid in creating systems that are simple to comprehend, maintain, and adapt. Here is a brief description of each:

Cohesion: Definition: The degree of relationship between the functions in a single module (or class) is known as cohesion. A cohesive module consists of parts that cooperate to complete a single task.
High Cohesion: When a module concentrates on mastering a single task. This is regarded as beneficial since it simplifies the system's understanding, testing, and maintenance.
Low Cohesion: A system that is disjointed and difficult to administer as a result of a module attempting to accomplish numerous unrelated tasks.
For instance:

High Cohesion: An invoice-only printing class named InvoicePrinter. It has a single function, and all of its techniques are connected to printing.
Low Cohesion: A class named Utility that performs a variety of duties, such as emailing, managing users, and printing bills. The lack of close relationship between these activities makes managing the code more difficult.
Definition of coupling: coupling is the degree of interdependence among many classes or modules. Strongly connected modules are independent, whereas low-coupling modules are loosely related.
Modules with low coupling have the least amount of interdependence possible. This is beneficial since it makes the system more flexible and allows for adjustment without jeopardizing other elements.
High coupling raises the fragility of the system since it results in a high degree of intermodular dependency. If one piece changes, many others may break.
For instance:

Low coupling: When an InvoicePrinter class solely communicates with another Invoice type via well-defined interfaces, modifications to the Invoice class have no

impact on the InvoicePrinter class.
High coupling occurs when an InvoicePrinter class directly reads and changes an Invoice class's data, making it challenging to make changes to one class without also affecting the other.
How These Ideas Support Excellent Design: Sustainability

A system with high cohesiveness is more easily able to identify and resolve problems because every component is dedicated to a single mission.
Low coupling makes maintenance simpler by ensuring that modifications to one component of the system do not damage other components.
Adaptability

One module can be changed or replaced with low coupling without impacting the others. As a result, updating or altering the system over time is made simpler.
Readability and Ease of Use:

Code with high cohesiveness is easy to read because each module performs a single task. To find out what a module performs, you don't have to search around.
An illustration of both:
Assume you are developing an internet-based shop.

High Cohesion: The only thing your class ShoppingCart does is add, remove, and display items in the cart.
Low Coupling: ShoppingCart uses interfaces to call the methods of the Inventory and Payment classes rather than speaking with them directly. This manner, the operation of the shopping cart will be unaffected in the event that the payment system is changed.
You can write software that is modular, simple to maintain, and less likely to break when changes are made by creating systems with low coupling and high cohesiveness.

QUESTION 3

Many factors can lead to the failure of software projects, and inadequate planning, communication, or execution are among the most common ones. These are the following five typical causes:

1. Unclear or Changing Requirements:

It can cause confusion among developers and stakeholders when project requirements are not well-defined from the outset. Project development can also be derailed by constant changes, as teams may need to redo work or alter course regularly.

2. Inadequate Planning:
One of the main reasons projects fail is inadequate planning, which includes erroneous budgets, schedules, and resource allocation. Without a good strategy, teams frequently go over budget or behind schedule, which results in software that is either incomplete or of poor quality.
3. Inadequate Communication: Misunderstandings regarding project objectives and expectations may arise from inadequate communication among the development team, stakeholders, and clients. Ineffective teamwork might cause problems to go undiscovered until it's too late to address them.

4. Unrealistic Technology Expectations or Technical Challenges:
Unexpected technical problems that the team is unable to resolve or technologies that are ill-suited for the job are two reasons why projects can go wrong. Teams who attempt to use novel or unproven technology without fully comprehending their limitations may experience this.
5. Lack of Skilled Resources:
The project team may find it difficult to achieve the technical requirements if they lack the requisite knowledge or expertise. Delays, subpar coding, and eventually project failure may arise from this.

Early resolution of these problems can greatly increase the likelihood of software development success.

QUESTION 4

4 (a)
Throw-away prototypes are frequently advised in big system development because they are quickly constructed to test concepts, get input, and elucidate needs without planning to incorporate them into the finished product. For big systems, disposable prototypes are recommended for the following reasons:

1. Exploration of Requirements:
At first, requirements for large systems may be ambiguous and complex. An interactive, rough version of the system is what a throw-away prototype offers to

help people understand what they need. The prototype is destroyed when input is received, and the final system is constructed with a better knowledge of the requirements (Sommerville, 2011).
2. Steer clear of badly written code:
Throw-away prototypes are usually developed rapidly, with little regard for long-term organization or high-quality code. If this type of code is used in the final system, it may cause future maintenance, scalability, and reliability issues. According to Pressman (2014), discarding the prototype guarantees that the final system is constructed with appropriate design and quality from the beginning.

3. Preventing System Bloat:
Prototypes are typically made to test particular features rather than being optimized or built to meet the performance requirements of the entire system. Should a prototype be retained, extraneous or ineffective code may find its way into the finished product, creating a bloated and ineffective system (Brooks, 1987).
4. Quicker Iteration:
Developers can produce disposable prototypes fast without worrying about long-term issues like maintainability because they are intended to be transitory. Faster feedback loops made possible by this enable the team to iterate on the idea more rapidly and successfully (Boehm, 1988).

In conclusion, adopting throw-away prototypes for large systems guarantees correct system design, helps prevent low-quality code in the finished product, and facilitates faster, more efficient requirement discovery.

4 (b)

In large system development, **throw-away prototypes** are often recommended because they are built quickly to explore ideas, gather feedback, and clarify requirements without the intention of integrating them into the final system. Here are some reasons why throw-away prototypes are preferred for large systems:

1. Exploration of Requirements:

In large systems, requirements can be complex and unclear at the start. A throw-away prototype helps clarify what the users need by providing a rough version of the system that users can interact with. Once feedback is gathered, the prototype is

discarded, and the final system is built with a clearer understanding of requirements (Sommerville, 2011).

2. Avoiding Poorly Designed Code:

Throw-away prototypes are typically built quickly, without a focus on code quality or long-term structure. Using this kind of code in the final system can lead to problems with maintenance, scalability, and reliability later on. Discarding the prototype ensures that the final system is built with proper design and quality from the start (Pressman, 2014).

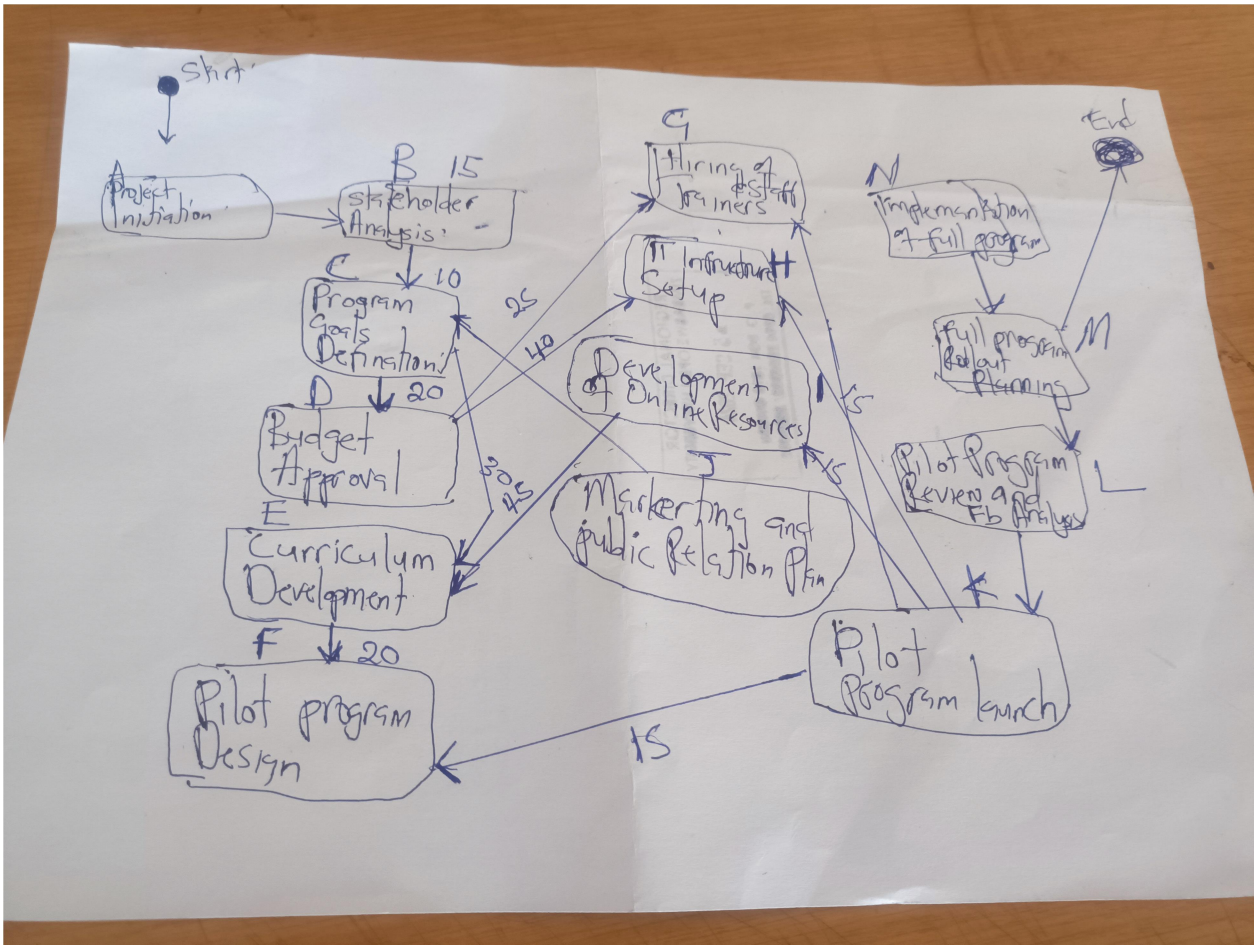
3. Preventing System Bloat:

Prototypes are usually created to test specific functionalities and are not optimized or designed for the full system's performance needs. If a prototype is kept, unnecessary or inefficient code might get integrated into the final product, resulting in a bloated and inefficient system (Brooks, 1987).

4. Faster Iteration:

Because throw-away prototypes are meant to be temporary, developers can build them quickly without worrying about long-term concerns like maintainability. This allows for faster feedback loops, helping the team iterate on the design quickly and effectively (Boehm, 1988).

5(a)



5(b)

Critical path

ABCDHKLM

A-D = 55

15 +50

CRITICAL PATH = 120