QUALITY ASSURANCE

AIN SHAMS UNIVERSITY

FACULTY OF ENGINEERING

CREDIT HOURS ENGINEERING PROGRAMS

Major Task

Software Quality Assurance (CSE429)

CESS - Spring 2024

Submitted to:

Dr. Islam El Maddah

Submitted by:

Salaheldeen Elhassan Salaheldeen Elsayed	20P2055
Mohamed Amr Abdelkhaleq	20P1643
Mohamed Mahmoud Hussein Ahmed	19P7975
Judy Khaled Mohamed Abdelmonem	20p6630

Table of Contents

Business case:

A software company consisting of five young people develop webpages for different clients.

The webpages are mainly related to educational games and based on servers. For each request two of the young team meet the client and then one of them joins one idle person to develop and script the needed webpage. The next job when arrives, they switch turns just to make sure there will be at least two teams to work in parallel and one of the developers is well aware of the client of the client requirements.

As the job markets for automating manual work increase as well as the clients look forward for more advanced services, the requests have increased dramatically on these five people. It is required to suggest and analyze the possible solutions for the company to survive with high quality output (doing jobs quickly and with minimum errors or deviation from requirements agreed with the client).

Current Operational Model:

Team Composition: The company is small, consisting of five members focused on developing webpages primarily for educational games.

Client Interaction: For each project, two team members initially meet the client to understand requirements.

Development Process: Post-client meeting, one of the members who attended the meeting pairs with another idle team member to develop the requested webpage.

Rotation and Work Distribution: The team members rotate roles to ensure continuous development capacity and familiarity with client requirements across the team.

1. Quality Management System:

A Quality Management System (QMS) is a structured system of procedures, processes, and practices that organizations use to ensure that they consistently meet customer requirements and enhance their satisfaction. It's centered around the principle of continual improvement and compliance with established standards and regulations.

Creating a Quality Management System (QMS) tailored for the software company involves establishing clear objectives, identifying key performance metrics, and outlining the specific data to be collected to monitor and improve quality consistently. For a QMS to be successful we will need to set up comprehensive processes that ensure every aspect of operations contributes to delivering high-quality products and services.

1.1. How to develop a QMS?

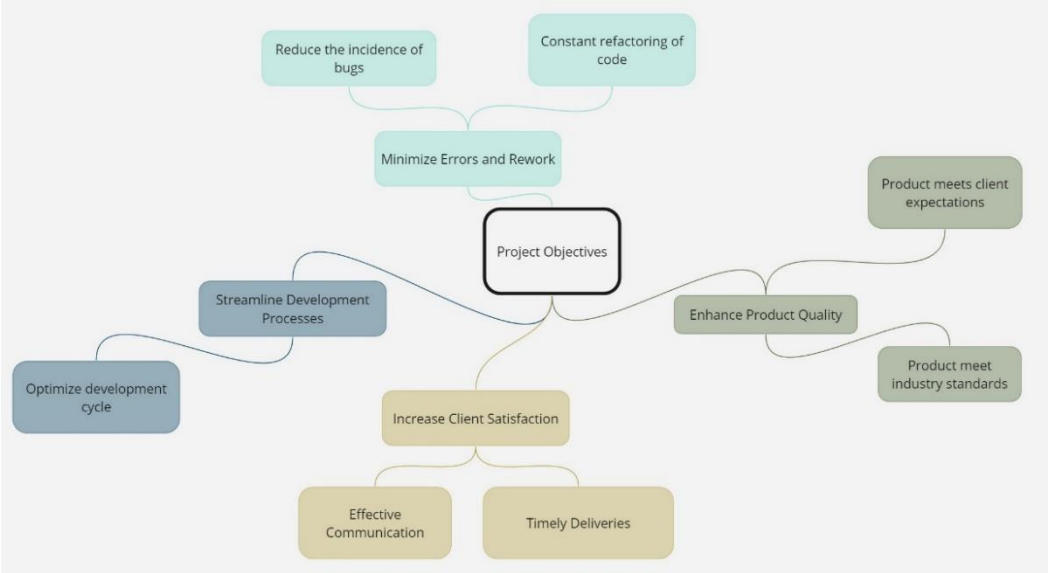
Developing a QMS for the company will involve a few steps and practices that the company will need to follow and integrate into the existing workflow, so it doesn't alter the current progress and add the concept of 'Quality' and 'Feedback' to the work being done, below are some steps and how to develop a QMS:

Step 1: Define Objectives

Identify Objectives: Establish clear, measurable objectives based on client needs, regulatory standards, and internal expectations.

Objectives for the company:

1. **Enhance Product Quality:** Ensure all delivered products meet client expectations and industry standards.
2. **Increase Client Satisfaction:** Achieve high client satisfaction through timely deliveries and effective communication.
3. **Streamline Development Processes:** Optimize development cycles to reduce time-to-market while maintaining high quality.
4. **Minimize Errors and Rework:** Reduce the incidence of bugs and errors in the final product deliveries.



Step 2: Collect Data:

Collecting relevant data throughout the project lifecycle is essential for monitoring progress, ensuring quality, and making informed decisions. Here are key data points to collect for each phase of the educational game webpage development project.

Project Data: Start and end dates, milestones, hours worked, number of revisions.

Quality Data: Number of defects found, type of defects, resolution time for defects.

Client Data: Client feedback scores, nature of complaints, repeat business instances.

Team Data: Hours worked per team member, task allocation, training sessions attended.

Step 3: Document Processes

Map Existing Processes: Document all current processes, including client meetings, development workflows, testing, and deployment.

Identify Gaps: Analyze these processes to identify any gaps or inefficiencies where quality could be compromised.

Step 4: Develop Quality Standards

Set Standards: Based on the documented processes, establish clear quality standards for each phase of the project lifecycle.

Adopt Best Practices: Consider industry standards like ISO 9001 for quality management systems and tailor them to the specific needs of web development.

Step 5: Implement Process Controls and Checkpoints

Quality Assurance (QA) Controls: Introduce checkpoints at critical stages of the development process to ensure adherence to quality standards.

Feedback Loops: Set up mechanisms to gather and analyze feedback from both clients and internal teams to continuously improve processes.

Step 6: Train and Engage Employees

Training Programs: Conduct training sessions to ensure all team members understand their roles in the QMS and how to perform their duties according to the new standards.

Engagement: Foster a quality-driven culture by involving employees in the development and refinement of quality processes.

Step 7: Monitor and Audit

Regular Audits: Schedule regular audits to check compliance with the QMS and identify areas for improvement.

Performance Metrics: Utilize metrics related to quality, such as client satisfaction scores, error rates, and adherence to project timelines, to monitor effectiveness.

Metrics to be calculated:

1. **Client Satisfaction Score (CSS):** Regular surveys to ensure client satisfaction after each project delivery.
2. **Defect Density:** Number of defects found per unit of software size (e.g., per thousand lines of code).
3. **Feature Completion Rate:** Tracks the number of features or user stories completed within each sprint or development cycle, showing how well the team is keeping up with increased demands.
4. **Cycle Time:** Time taken from the start to the end of a development phase, including time spent in different stages like planning, development, and testing.
5. **On-Time Delivery Rate:** Percentage of projects delivered by the agreed-upon deadline.
6. **Rework Level:** Percentage of completed work that requires rework due to errors or missed requirement

Step 8: Review and Improve

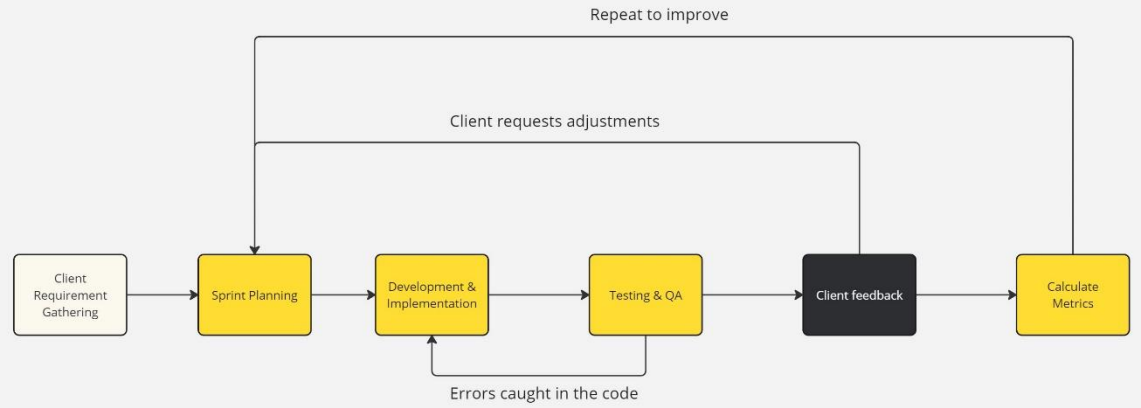
Management Reviews: Hold regular review meetings with management to discuss the outcomes of audits, the status of corrective actions, and potential improvements.

Continuous Improvement: Implement a continuous improvement plan that uses data from audits and reviews to make informed decisions about changes to processes or standards.we will see how to develop a QMS for the company:

1.2 Example / Sample:

Modified Workflow:

We will now discuss the stages of a new workflow that takes into consideration quality practices and helps the company implement the QMS.



1. Client Requirement Gathering

Objective: Understand the client's vision for the educational game, including specific educational goals, target audience, desired features, and technical requirements.

Actions:

Initial Meeting: Two team members (Team A) meet with the client to gather detailed information about the educational game's objectives, target age group, subject matter, interactivity level, and any specific game mechanics or features.

Requirement Documentation: Document the client's requirements, including detailed descriptions of game levels, educational content, user interface design, and any multimedia elements (e.g., animations, sounds).

Clarify Expectations: Ensure that the client's expectations regarding functionality, aesthetics, and user engagement are well understood and documented.

2. Sprint Planning

Objective: Plan the development process in manageable sprints to ensure efficient and organized progress.

Actions:

Define Sprint Goals: Based on the gathered requirements, define the specific goals and deliverables for the upcoming sprint. Prioritize features that are critical to the educational objectives.

Task Allocation: Allocate tasks among team members, ensuring a balanced workload. One developer from Team A and one idle team member form the development team for the sprint.

Set Deadlines: Establish clear targets and deadlines for each task to ensure timely completion of the sprint.

3. Development & Implementation

Objective: Build the educational game webpage according to the defined requirements and sprint plan.

Actions:

Coding and Development: The assigned development team works on coding the game's features and functionalities, focusing on creating engaging and educational content.

Regular Check-ins: Hold daily stand-up meetings to discuss progress, address any blockers, and ensure alignment with sprint goals.

Version Control: Use version control systems (e.g., Git) to manage code changes and collaborate effectively.

4. Testing & Quality Assurance

Objective: Ensure the game is free of bugs and meets quality standards, providing a reliable and engaging user experience.

Actions:

Functional Testing: Conduct thorough testing of the game's features to ensure they work as intended. This includes gameplay mechanics, educational content delivery, and user interface interactions.

Usability Testing: Evaluate the game for ease of use, ensuring it is intuitive for the target age group and effectively conveys the educational material.

Bug Fixing: Identify and resolve any issues or bugs found during testing.

5. Feedback from Client

Objective: Gather the client's input on the developed features to ensure alignment with their vision and expectations.

Actions:

Client Demo: Present the current version of the game to the client, showcasing the implemented features and educational content.

Feedback Collection: Collect detailed feedback from the client regarding the functionality, design, and educational effectiveness of the game.

Document Revisions: Document any requested changes or additional features based on the client's feedback.

6. Calculate Metrics

Objective: Assess the quality and success of the project using specific metrics to identify areas for improvement.

Actions:

Client Satisfaction Score: Measure the client's satisfaction based on their feedback and overall experience with the development process.

Defect Density: Calculate the number of defects per unit of code to assess the quality of the development.

Usability Metrics: Evaluate the ease of use and educational impact of the game based on user testing results.

Performance Metrics: Assess the game's performance, including loading times and responsiveness.

7. Loop Back to Sprint Planning

Objective: Use feedback and metrics to refine the development process and plan for the next sprint.

Actions:

Review Feedback and Metrics: Analyze the feedback and metrics to identify areas that need improvement or additional features that should be added.

Refine Sprint Goals: Adjust the goals for the next sprint based on the analysis, ensuring the game aligns more closely with client needs and quality benchmarks.

Plan Next Sprint: Define tasks and allocate resources for the next sprint, continuing the iterative cycle to progressively enhance the game.

1.3 Costs & Efforts:

Implement Quality Management System in the company will be accompanied by some efforts and sacrifices by the team in terms of effort, money, and time. Given that the company only consists of 5 people training them won't be difficult.

1. Time

Training and Setup: Approximately 2-3 full days of training on QMS processes and tools, including some self-study and online courses. This will get the team member familiar with how to follow Quality practices and make the most out of the QMS.

Process Adaptation: Small teams can also adapt more quickly due to fewer bureaucratic hurdles. This might take a few weeks of intermittent effort as you refine processes.

2. Efforts

Quality Assurance activities: Incorporating additional QA activities into the current workflows will increase the effort by approximately 10-15% per project. This increment accounts for the additional steps of testing and reviews needed to comply with the QMS standards without significantly burdening the team.

Monitoring and Feedback: Setting up a lightweight system for ongoing monitoring and feedback, which includes brief weekly review meetings and client feedback loops. This approach adds time and effort to each project cycle but provides continuous insights into quality improvements.

3. Money

Training Costs: Online courses or hiring a local consultant for a few sessions might cost between $500 to $2,000, depending on the chosen quality standards and the depth of training required.

Software and Tools: Subscription costs for project management and QA tools can vary, but expect to pay about $10 to $100 per user per month, depending on the sophistication and capabilities of the software. For five people, this could range from $50 to $500 per month.

Certification (Optional): For a certification like ISO 9001, the costs can include audit fees, registration fees, and potential consultancy fees. This might total anywhere from $3,000 to $10,000, but considering the small size of the company, it might be on the lower end unless.

1.4 Benefits:

1. Improved Product Quality

Consistency: A QMS helps ensure that every project is delivered with consistent quality, adhering to predefined standards. This is crucial for maintaining a strong reputation in a competitive market.

Reduction in Defects: Systematic quality checks and processes reduce the occurrence of bugs and defects in the final product, enhancing the overall user experience.

2. Increased Customer Satisfaction

Meeting Expectations: By systematically gathering and incorporating client feedback, the company can more effectively meet client expectations, leading to higher satisfaction rates.

Professional Service: The structured approach to handling projects can improve the client's perception of professionalism, potentially leading to repeat business and referrals.

3. Enhanced Efficiency and Reduced Waste

Process Optimization: A QMS streamlines processes, eliminating unnecessary steps and focusing resources on value-adding activities, thus saving time and reducing costs.

Predictability: Improved process control and project management lead to more predictable outcomes, enabling better planning and resource allocation.

2. House of Quality:

House of Quality is a structured method for translating customer requirements into appropriate technical specifications for each stage of product development and production. The HoQ is visually represented as a matrix.

House of Quality (HoQ) is vital as it enables the company to systematically address the increasing complexity and volume of customer demands for educational game webpages. By mapping out customer requirements directly against technical capabilities, the HoQ ensures that every feature developed aligns with what clients' value most. This method helps the team prioritize tasks, efficiently allocate limited resources, and maintain high quality even as project demands escalate.

2.1 How to develop a HoQ:

To develop a HoQ matrix that correctly identifies the relationship between the How and the Whats and determine the importance of each customer need we will need 4 key components:

1. **Identify Customer Requirements (What):**
   At the top left of the matrix, customer needs or requirements (often referred to as "Whats") are listed. These are gathered from customer feedback, market research, and other sources to ensure the product meets or exceeds customer expectations.
2. **Determine Technical Specifications(How):**
   Along the top, running horizontally, are the technical descriptors or specifications (referred to as "Hows"), which represent how the team will meet the customer's needs. These are typically engineering or design parameters that the product must adhere to.
3. **Construct Relationship Matrix:**
   At the center of the HoQ, a matrix (usually filled with numbers) shows the relationship between the customer requirements and the technical specifications. This helps to visually identify how strongly each technical specification affects each customer requirement.
4. **Prioritize Requirements:**
   On the right side of the matrix, there is often a section that ranks the importance of each customer requirement, helping to prioritize which aspects of the product development should receive the most focus and resources.

2.2 Example / Sample:

First, we will define the ("Whats"), These are the needs and wants of the customer, which should be gathered through direct communication, feedback, and market research. Here are some needs that the customer might request:

1 Interactive and engaging user interface.
2 Adaptive learning features.
3 High performance with no latency.
4 Accessibility on multiple devices (responsive design).
5 Secure user data handling.
6 Easy navigation.
7 Educational value and alignment with curriculum standards.

Next, we will define the ("Hows") These are the specific engineering or technical responses to the customer requirements. They explain how the team will meet the customer's needs. Here are some examples:

1 Use of JavaScript frameworks like React or Angular for interactivity.
2 Implementation of AI algorithms for adaptive learning.
3 Optimization of server responses and load balancing.
4 Responsive web design using CSS frameworks like Bootstrap.
5 Implementation of HTTPS and secure authentication methods.
6 User-friendly UI/UX design with clear navigation.
7 Content development in alignment with educational experts and standards.

Relationship Strengths: We will use numbers to indicate the strength of the relationship between customer requirements and technical descriptors:

- 9 for a strong relationship
- 3 for a medium relationship
- 1 for a weak relationship
- 0 if there is no relationship.

Importance Ratings: We'll calculate the relative importance of each customer requirement based on the sum of the products of relationship strengths and a base importance score given to each requirement. This base score represents the initial assessment of importance before considering technical difficulties and market demands.

Customer Requirements	Base Importance	JS Frameworks	AI Algorithms	Server Optimization	Responsive Design	Secure Data Handling	User-Friendly Design	Educational Content	Total Score	Relative Importance (%)
Interactive UI	10	9	0	0	9	0	9	0	120	12.99%
Adaptive Learning	8	0	9	0	0	0	0	9	99	10.71%
High Performance	10	3	0	9	0	0	0	0	90	9.74%
Accessibility	8	0	0	0	9	0	0	0	192	20.78%
Secure User Data	10	0	0	0	0	9	0	0	90	9.74%
Easy Navigation	10	0	0	0	3	0	9	0	180	19.48%
Educational Value	9	0	3	0	0	0	0	9	153	16.56%

2.3 Costs & Efforts:

1. Effort:

Training and Learning: If the team is not familiar with Quality Function Deployment (QFD) and the HoQ methodology, there will be a learning curve. This includes training sessions or workshops to understand how to effectively use the HoQ for project planning.

Development of HoQ: The effort to gather comprehensive customer requirements, translate them into engineering characteristics, and map out the relationship matrix is substantial. This requires detailed discussions with clients, extensive internal meetings to translate these into technical specifications, and then iterative reviews to refine the HoQ.

2. Money:

Resources and Tools: While basic HoQ can be developed using common tools like Excel or specialized QFD software, there may be costs associated with acquiring more advanced project management or QFD-specific software to facilitate the process.

Consultant Fees: If the team lacks expertise, hiring a consultant to train the team or guide the initial deployment of the HoQ process could be necessary, which incurs additional costs.

3. Time:

Development Time: Creating a detailed HoQ is time-consuming. It involves multiple stages—from initial customer interviews to multiple validation and adjustment phases to ensure that all elements are correctly aligned and prioritized.

Integration into Workflow: Integrating this new tool into the company's existing project management workflow might initially slow down project initiation phases as team members adapt.

2.4 Benefits:

1. Improved Project Alignment with Client Needs:

By using HoQ, the team can ensure a better understanding and alignment of the product's features with the client's needs and expectations, potentially leading to higher customer satisfaction and fewer revisions or misunderstandings.

2. Enhanced Communication and Documentation:

The HoQ process requires and fosters improved communication both internally among team members and externally with clients. It also serves as a valuable documentation tool that clearly outlines how customer needs are translated into specific product features.

3. Quality Improvement:

Systematic attention to client requirements from the outset can lead to higher-quality outputs. The structured approach of the HoQ helps in prioritizing features based on their impact on customer satisfaction and technical feasibility, leading to better resource allocation and focus.

4. Strategic Planning and Risk Management:

HoQ can help in identifying potential risks early in the development process by highlighting areas where customer requirements are difficult to meet or where the development team lacks expertise. This early identification helps in better planning and can mitigate risks more effectively.

3. Failure Mode Error Analysis (FMEA):

Failure Modes and Effects Analysis (FMEA) is a systematic, proactive method for evaluating a process to identify where and how it might fail and to assess the relative impact of different failures, to identify the parts of the process that are most in need of change. FMEA helps teams identify potential problems before they occur and allows for the implementation of corrective actions to mitigate risks.

3.1 How to develop FMEA:

Step 1: Select the System or Process to Analyze

Identify the system or process you want to analyze. This could be a product, service, manufacturing process, or any other process within your organization.

Step 2: Assemble a Cross-Functional Team

Gather a team of individuals with diverse expertise related to the system or process being analyzed. This may include engineers, designers, quality assurance specialists, and other relevant stakeholders.

Step 3: Identify Potential Failure Modes

Brainstorm and list all possible failure modes that could occur within the system or process. Consider different aspects such as design, materials, manufacturing, human error, and environmental factors.

Step 4: Determine Potential Effects of Each Failure Mode

For each identified failure mode, assess the potential effects it could have on the system, process, or end-users. Consider the severity of the consequences if the failure were to occur.

Step 5: Evaluate Likelihood of Occurrence

Determine how likely each failure mode is to occur. Consider historical data, expert judgment, and other relevant information to assess the probability of occurrence.

Step 6: Assess Detection Methods

Evaluate the effectiveness of existing controls or detection methods in identifying or preventing the occurrence of each failure mode. Determine how likely it is for the failure mode to be detected before it reaches the customer or end-user.

Step 7: Calculate Risk Priority Number (RPN)

Calculate the Risk Priority Number (RPN) for each failure mode by multiplying the severity, occurrence, and detection ratings assigned in the previous steps. RPN = Severity × Occurrence × Detection.

Step 8: Prioritize Failure Modes

Prioritize the identified failure modes based on their RPN values. Focus on addressing the failure modes with the highest RPN scores, as they represent the most significant risks to the system or process.

Step 9: Develop and Implement Preventive Measures

For each high-priority failure mode, brainstorm and implement preventive measures to mitigate the associated risks. These measures could include design changes, process improvements, additional controls, or training initiatives.

Step 10: Monitor and Review

Regularly monitor the effectiveness of the preventive measures implemented and review the FMEA process periodically to identify any new failure modes or changes in risk levels.

3.2 Example / Sample

Failure Mode Error Analysis (FMEA) table:

Below we see an example of an FMEA table tailored for our company and a potential project that it could be working on. We can see the process and what failure mode can occur and what is its effect & cause, and finally how to detect it

x

PROCESS	FAILURE MODE	EFFECT	SEVERITY	CAUSE	OCCURENCE	DETECTION METHOD	DETECTION	RPN	ACTION
Game Design	Poor gameplay mechanics	Game is not engaging or educational	9	Inadequate market research	5	User testing, educator feedback	3	135	Involve educators in design phase
Interactive Content	Lack of educational value	Content does not meet educational objectives	8	Misalignment with curriculum standards	6	Educational consultant review	4	192	Regular reviews by educational consultants
User Interface Design	Inconsistent UI/UX	Users find interface confusing	7	Inexperienced design team	4	Usability testing	3	84	Usability testing sessions
Development	Bugs in game features	Gameplay interrupted, features don't work	8	Rushed development, complex code	5	Code reviews, automated testing	2	80	Implement thorough testing and code reviews
Server Integration	Server downtime or slow response times	Game is slow or inaccessible	9	Inadequate server capacity	4	Performance monitoring	3	`108	Optimize server resources
Accessibility Features	Lack of accessibility options	Game is not usable by all potential users	7	Overlooked during design	3	Accessibility testing	4	84	Include accessibility features in initial design
Educational Feedback	Ineffective learning feedback mechanisms	Learners do not receive useful progress updates	8	Poor feedback system design	5	Educator and student reviews	3	120	Involve educational experts in feedback system design
Multiplayer Functionality	Multiplayer issues (sync/errors)	Poor user experience in multiplayer mode	7	Network issues, coding errors	4	Multiplayer testing sessions	3	84	Frequent multiplayer testing and network optimization
Content Updates	Delayed content updates	Content becomes outdated, losing engagement	6	Resource constraints	3	Client and user feedback	5	90	Establish a scheduled content update plan

Recalculated FMEA:

PROCESS	ACTION	NEW SEV	NEW OCC	NEW DET	NEW RPN
Game Design	Involve educators in design phase	6	3	2	36
Interactive Content	Regular reviews by educational consultants	7	4	3	84
User Interface design	Usability testing sessions	5	3	2	30
Development	Implementing testing & code reviews	6	3	2	36
Server Integration	Optimize server resources	7	3	2	42
Accessibility features	Include accessibility features in initial design	5	2	3	30
Educational Feedback	Include educational experts	6	3	2	36
Multiplayer functionality	Frequent multiplayer testing	5	3	2	30
Content Updates	Delayed Content Updates	4	2	3	24

3.3 Costs & Efforts:

1. Effort:

Comprehensive Process Analysis: The team needs to thoroughly understand and document each step of the software development and client interaction processes to identify potential failure modes effectively.

Regular Review Meetings: Continuous efforts to review and update the FMEA document as the project progresses, or as new risks are identified, involving all relevant team members.

2. Money:

Software Acquisition: Potential costs for purchasing specialized software to facilitate the FMEA process, which might include features for risk analysis, process mapping, and collaborative tools for team input.

Professional Development: Possible expenses for professional training or workshops to ensure that all team members are proficient in FMEA methodology and tools.

3. Time:

Initial Setup and Training: The time investment to set up the FMEA framework and train the team on its application. This includes the time spent on learning the methodology and adapting existing processes to include FMEA steps.

Ongoing Implementation: Time dedicated to the continuous application of the FMEA throughout the project lifecycle, including regular updates and iterations of the analysis as development progresses.

3.4 Benefits:

Using FMEA (Failure Modes and Effects Analysis) offers several benefits to organizations across various industries. Here are some key advantages:

1. Proactive Risk Management:

FMEA allows organizations to identify potential failure modes and their associated risks before they occur. By addressing these risks proactively, organizations can prevent costly failures, reduce downtime, and improve overall reliability.

2. Improved Product Quality:

By systematically analyzing potential failure modes and their effects, organizations can identify areas for improvement in product design, manufacturing processes, and quality control measures. This leads to higher quality products that meet or exceed customer expectations.

3. Cost Reduction: FMEA helps organizations identify potential failure modes early in the development or production process, reducing the likelihood of costly rework, warranty claims, and product recalls. By addressing potential issues upfront, organizations can save time and resources in the long run.

4.Increased Customer Satisfaction: By proactively identifying and addressing potential failure modes, organizations can deliver products and services that meet or exceed customer expectations in terms of reliability, performance, and safety. This leads to higher levels of customer satisfaction and loyalty.

5. Continuous Improvement: FMEA is not a one-time activity but an ongoing process. By regularly reviewing and updating FMEA analyses, organizations can continuously improve their products, processes, and systems over time, leading to greater efficiency, innovation, and competitiveness.

Part 2: Code

Metrics Calculator

4. Lexer (Tokenizer)

Our team collaborated to develop a Lexer for C++, breaking down code into tokens like keywords and operators. We use this tokenizer to get the **n1** (number of operators), **n2** (number of operands), **N1** (total number of operators), **N2** (total number of operands), **program length** (N1 + N2 = N), and **vocabulary** (n1 + n2 = n).

4.1 Lexer Code

```
struct Token {
    TokenType type;
    std::string lexeme;
};



class Lexer {
public:
    Lexer(const std::string& input);
    Token getNextToken();
    std::vector <int> numbers;
    std::vector <std::string> identifiers;


private:
    const std::string input;
    size_t position;
    std::string currentBuffer;
    std::string nextBuffer;
    const size_t bufferSize;

    void swapBuffers();

    std::string extractLexeme(const std::string& buffer);

    bool isDelimiter(char c);

    void handleToken(Token& token);

    void skipWhiteSpaces();

    void skipComments();

    bool isComment(const std::string& str);

    void skipNewLines();
};

#endif /* lexer_hpp */
```

This header file defines a lexer class and a token structure for lexical analysis in C++ code. The Lexer class has methods for initializing with input, getting the next token, and handling whitespace, comments, and new lines. It uses a buffer to efficiently process the input string. The Token structure holds information about the type and lexeme of a token.

Example Implementations of the functions

Get Next Token Function:

Purpose: Retrieves the next token from the input.

```
Lexer:: Lexer(const std::string& input) : input(input), position(0), bufferSize(4096) {
       // Initialize buffers
       currentBuffer = input.substr(0, std::min(bufferSize, input.length()));
       if (input.length() > bufferSize) {
           nextBuffer = input.substr(bufferSize, std::min(bufferSize, input.length() - bufferSize));
       }
       else {
           nextBuffer = "";
       }
   };

Token Lexer :: getNextToken() {
       Token token;

        // Skip whitespace, comments, and new lines:
        skipWhiteSpaces();
        skipComments();
        skipNewLines();
        skipWhiteSpaces(); // called for the second time to delete spaces after comments or multi-line comments


       // Check if current buffer is exhausted (lexer has consumed all the tokens)
       if (position >= currentBuffer.length() && nextBuffer.empty()) {
           token.lexeme = "";
           token.type = TOKEN_UNKNOWN;
           return token;
       }

       // Extract lexeme from current buffer
    token.lexeme = extractLexeme(currentBuffer.substr(position));

       // Handle tokens and update position
       handleToken(token);
       position += token.lexeme.length();

       // Check if current buffer is empty and there are more characters in the input
       if (position >= currentBuffer.length() && !nextBuffer.empty()) {
           swapBuffers();

           position = 0; // Reset position after swapping buffers
       }

       return token;
   }
```

Explanation:

1. **Skip Whitespace, Comments, and New Lines**:
    - The function starts by skipping over any whitespace, comments, and new lines encountered at the current position in the input buffer. This ensures that the lexer proceeds to the next meaningful token.
2. **Check Buffer Exhaustion**:
    - After skipping over whitespace, comments, and new lines, the function checks if the current buffer is exhausted. If both the current buffer and the next buffer are empty, it means the lexer has consumed all the tokens, and it returns a token with type **TOKEN_UNKNOWN**.

3. **Extract Lexeme**:
    - If the buffer is not exhausted, the function extracts the lexeme from the current buffer starting from the current position. The lexeme represents the text of the token being processed.
4. **Handle Token and Update Position**:
    - Once the lexeme is extracted, the function handles the token by determining its type and updating its properties accordingly. It also updates the position to point to the next character after the extracted lexeme.
5. **Check for Buffer Switch**:
    - After processing the current buffer, the function checks if the current buffer is empty but there are more characters in the input. If so, it swaps the current and next buffers and resets the position to the beginning of the new current buffer. This ensures continuous tokenization of the input text across buffer boundaries.
6. **Return Token**:
    - Finally, the function returns the token representing the extracted lexeme and its type.

Skip Comments Function:

Purpose: This function is responsible for skipping comments in the input text being processed by the lexer.

```
void Lexer::skipComments() {
    // Check for one-line comment
    if (currentBuffer[position] == '/' && position + 1 < currentBuffer.length() &&
currentBuffer[position + 1] == '/') {
        // Skip until the end of line
         while (position < currentBuffer.length() && currentBuffer[position] !=
'\n')
            position++;
    }
    // Check for multi-line comment
   else if (currentBuffer[position] == '/' && position + 1 < currentBuffer.length()
&& currentBuffer[position + 1] == '*') {
        // Skip until the end of the multi-line comment
        while (position < currentBuffer.length() - 1 && !(currentBuffer[position]
== '*' && currentBuffer[position + 1] == '/'))
            position++;
```

```
        // Move past the end of the multi-line comment
        if (position < currentBuffer.length() - 1)
            position += 2;
    }
}
```

1. **One-line Comment Detection**:
    - It checks if the current character in the buffer is '/' and the next character is also '/'.
    - If both conditions are true, it signifies the beginning of a one-line comment.
    - Then, it enters a loop that advances the position until it encounters a newline character ('\n') or reaches the end of the buffer, effectively skipping the entire comment line.
2. **Multi-line Comment Detection**:
    - If the current character is '/' and the next character is '*', it indicates the beginning of a multi-line comment.
    - It enters a loop that continues until it finds the end of the multi-line comment (signaled by '*/').
    - Within this loop, it checks if the current character and the next character form the '*/' sequence, indicating the end of the comment block.
    - If not found, it continues advancing the position until the end of the buffer.
    - After exiting the loop (i.e., after finding the end of the multi-line comment), it moves the position two characters ahead to skip over the '*/' sequence.

Matching Assignment Operators:

```
if (LexerRegex::matchAssignmentOperator(token.lexeme)) {
  if (token.lexeme == "=") {
      token.type = TOKEN_ASSIGNOP;
  }
  else if (token.lexeme == "+=") {
      token.type = TOKEN_ADDASSIGNOP;
  }
  else if (token.lexeme == "-=") {
      token.type = TOKEN_SUBASSIGNOP;
  }
  else if (token.lexeme == "*=") {
      token.type = TOKEN_MULASSIGNOP;
  }
  else if (token.lexeme == "/=") {
      token.type = TOKEN_DIVASSIGNOP;
  }
  else if (token.lexeme == "%=") {
      token.type = TOKEN_MODASSIGNOP;
  }
  else if (token.lexeme == "&=") {
      token.type = TOKEN_ANDASSIGNOP;
  }
  else if (token.lexeme == "|=") {
      token.type = TOKEN_ORASSIGNOP;
  }
  else if (token.lexeme == "^=") {
      token.type = TOKEN_XORASSIGNOP;
  }
  else if (token.lexeme == "<<=") {
      token.type = TOKEN_LEFTASSIGNOP;
  }
  else if (token.lexeme == ">>=") {
      token.type = TOKEN_RIGHTASSIGNOP;
  }
```

Regex:

```
bool LexerRegex::matchAssignmentOperator(const std::string& str) {
    static const std::regex assignmentOpPattern("=|\\+=|\\-=|\\*=|/=|%=|&=|\\|=|\\^=|<<=|>>=");
    return std::regex_match(str, assignmentOpPattern);
}
```

Matching Bitwise Operators:

```
else if (LexerRegex::matchBitwiseOperator(token.lexeme)) {
    if (token.lexeme == "&") {
        token.type = TOKEN_Bitwise_AND;
    }
    else if (token.lexeme == "&") {
        token.type = TOKEN_Bitwise_AND;
    }
    else if (token.lexeme == "|") {
        token.type = TOKEN_Bitwise_OR;
    }
    else if (token.lexeme == "^") {
        token.type = TOKEN_Bitwise_XOR;
    }
    else if (token.lexeme == "~") {
        token.type = TOKEN_Bitwise_NOT;
    }
 }
```

Regex:

```
bool LexerRegex::matchBitwiseOperator(const std::string& str) {
    static const std::regex bitwiseOperatorPattern("&|\\||\\^|~");
    return std::regex_match(str, bitwiseOperatorPattern);
}
```

Matching Unary Operator:

```
else if (LexerRegex::matchUnaryOperator(token.lexeme)) {

    if (token.lexeme == "++") {
        token.type = TOKEN_INCOP;
    }
    else if (token.lexeme == "--") {
        token.type = TOKEN_DECOP;
    }

}
```

Regex:

```
bool LexerRegex::matchUnaryOperator(const std::string& str) {
    // Regular expression pattern for unary operators:
    static const std::regex unaryOperatorPattern("\\+\\+|\\-\\-");
    return std::regex_match(str, unaryOperatorPattern);;
}
```

Matching Binary Operator:

```
 else if (LexerRegex::matchBinaryOperator(token.lexeme)) {
     switch (token.lexeme[0]) {
     case '+':
         token.type = TOKEN_ADDOP;
         break;
     case '-':
         token.type = TOKEN_SUBOP;
         break;
     case '*':
         token.type = TOKEN_MULOP;
         break;
     case '/':
         token.type = TOKEN_DIVOP;
         break;
     case '%':
         token.type = TOKEN_MODOP;
         break;
     }

 }
```

Regex:

```
bool LexerRegex::matchBinaryOperator(const std::string& str) {
    // Regular expression pattern for binary operators:
    static const std::regex binaryoperatorPattern("[\\+\\-\\*%]");
    return std::regex_match(str, binaryoperatorPattern);
}
```

Matching Logical Operator:

```
 else if (LexerRegex::matchlogicalOperator(token.lexeme)) {

     if (token.lexeme == "&&") {
         token.type = TOKEN_ANDOP;
     }
     else if (token.lexeme == "||") {
         token.type = TOKEN_OROP;
     }
     else if (token.lexeme == "!") {
         token.type = TOKEN_NOTOP;
     }

 }
```

Regex:

```
bool LexerRegex::matchlogicalOperator(const std::string& str) {
    static const std::regex logicalOperatorPattern("\\&\\&|\\|\\||!");
    return std::regex_match(str, logicalOperatorPattern);
}
```

Match Relational Operator:

```
else if (LexerRegex::matchRelationalOperator(token.lexeme)) {
          if (token.lexeme == "==") {
              token.type = TOKEN_EQOP;
          }
          else if (token.lexeme == "!=") {
              token.type = TOKEN_NEQOP;
          }
          else if (token.lexeme == ">") {
              token.type = TOKEN_GTOP;
          }
          else if (token.lexeme == "<") {
              token.type = TOKEN_LTOP;
          }
          else if (token.lexeme == ">=") {
              token.type = TOKEN_GTEOP;
```

```
            }
            else if (token.lexeme == "<=") {
                token.type = TOKEN_LTEOP;
            }
        }
```

Regex:

```
bool LexerRegex::matchRelationalOperator(const std::string& str) {
    static const std::regex relationalOpPattern("<=?|>=?|==|!=");
    return std::regex_match(str, relationalOpPattern);
}
```

Match Ternary Operator:

```
else if (LexerRegex::matchTernaryOperator(token.lexeme)) {
            token.type = TOKEN_TERNARYOPERATOR;
        }
```

Regex:

```
bool LexerRegex::matchTernaryOperator(const std::string& str) {
    static const std::regex ternaryOpPattern("\\?\\:");
    return std::regex_match(str, ternaryOpPattern);
}
```

Match Punctuation:

```
else if (LexerRegex::matchPunc(token.lexeme)) {

             if (token.lexeme == "(") {
                token.type = TOKEN_LEFT_PAREN;
            }
            else if (token.lexeme == ")") {
                token.type = TOKEN_RIGHT_PAREN;
            }
            else if (token.lexeme == "{") {
                token.type = TOKEN_LEFT_BRACE;
            }
            else if (token.lexeme == "}") {
                token.type = TOKEN_RIGHT_BRACE;
            }
```

```
        else if (token.lexeme == "[") {
            token.type = TOKEN_LEFT_BRACKET;
        }
        else if (token.lexeme == "]") {
            token.type = TOKEN_RIGHT_BRACKET;
        }
        else if (token.lexeme == "'") {
            token.type = TOKEN_QUOTE;
        }
        else if (token.lexeme == "\"") {
            token.type = TOKEN_DOUBLE_QUOTE;
        }
        else if (token.lexeme == ".") {
            token.type = TOKEN_PERIOD;
        }
        else if (token.lexeme == ",") {
            token.type = TOKEN_COMMA;
        }
        else if (token.lexeme == ";") {
            token.type = TOKEN_SEMICOLON;
        }
        else if (token.lexeme == ":") {
            token.type = TOKEN_COLON;
        }
        else if (token.lexeme == ".") {
            token.type = TOKEN_PERIOD;
        }
        else if (token.lexeme == "?") {
            token.type = TOKEN_QUESTION;
        }
        else if (token.lexeme == "#") {
            token.type = TOKEN_PREPROCESSOR_HASH;
        }
        else if (token.lexeme == "\\") {
            token.type = TOKEN_BACKSLASH;
        }
        else if (token.lexeme == "\n") {
            token.type = TOKEN_NEWLINE;
        }
        else {
        token.type = PUNCTUATION;
        }
    }
```

```
        else if (token.lexeme == " ") {
            token.type = TOKEN_HEX;
        }

        else {
            token.type = TOKEN_UNKNOWN;
        }
```

Regex :

```
bool LexerRegex::matchPunc(const std::string& str) {
    // Regular expression pattern for punctuation:
    // static const std::regex PuncPattern("\\(|\\)|,|;|;\\{|\\}");
                                static              const               std::regex
PuncPattern("\\(|\\)|\\{|\\}|\\[|\\]|,|\\.|;|:|\\?|'|\"|`|\\\\|\\n|#");
    return std::regex_match(str, PuncPattern);}
```

Printing Tokens:

```
void printTokenType(Token token) {

 switch (token.type) {
        case TOKEN_KEYWORD:
            std::cout << "Keyword" << std::endl;
            break;
        case TOKEN_IDENTIFIER:
            std::cout << "Identifier" << std::endl;
            break;
        case TOKEN_NUMBER:
            std::cout << "Number" << std::endl;
            break;
        case TOKEN_FLOAT:
            std::cout << "Float" << std::endl;
            break;
        case TOKEN_BINARY:
            std::cout << "Binary" << std::endl;
            break;
        case TOKEN_HEX:
            std::cout << "Hexadecimal" << std::endl;
            break;
        case TOKEN_OCTAL:
            std::cout << "Octal" << std::endl;
            break;
        case TOKEN_INCOP:
            std::cout << "INCOP" << std::endl;
            break;
        case TOKEN_DECOP:
            std::cout << "DECOP" << std::endl;
            break;
        case TOKEN_ANDOP:
            std::cout << "ANDOP" << std::endl;
            break;
        case TOKEN_OROP:
            std::cout << "OROP" << std::endl;
            break;
        case TOKEN_NOTOP:
            std::cout << "NOTOP" << std::endl;
            break;
        case TOKEN_ADDOP:
            std::cout << "ADDOP" << std::endl;
```

5. Halstead

Halstead metrics, named after Maurice Halstead, are a set of software metrics used to measure various aspects of a program's complexity and size based on the number of unique operators and operands used in the code. These metrics provide valuable insights into the software's complexity, understandability, and maintainability. Halstead metrics include measures such as program length, vocabulary size, volume, difficulty, and effort, which quantify different aspects of a program's complexity.

5.1 Halstead Functions Code

```cpp
#include "halstead_caluculations.hpp"
#include <iostream>
#include <math.h>
using namespace std;

float getEstimatedLength(int n1, int n2){
    return n1*log(n1) + n2*log(n2);
}

float getPurityRatio(int n1, int n2,int N1, int N2){
    return  getEstimatedLength(n1,n2) /getProgramLength(N1, N2) ;
}

float getVolume(int n1,int n2){
    return getEstimatedLength(n1,n2)*log(getProgramVocabulary(n1,n2));
}

float getDifficulty(int n1,int n2, int N2){
    return (n1/2)*(N2/n2);
}

float getEffort(int n1,int n2,int N2){
    return getDifficulty(n1,n2,N2) * getVolume(n1,n2);
}
float getNumberOfBugs_Volume(int n1, int n2){
    return getVolume(n1,n2)/3000;
}
float getNumberOfBugs_Effort(int n1, int n2,int N2){
   return pow(getEffort(n1,n2,N2),(2/3))/3000;
}
int getProgramLength(int N1, int N2){
    return  N1+N2;
}
int getProgramVocabulary(int n1, int n2){
    return  n1+n2;
}
```

Count number of operators, operands and their total function:

```
} while (token.lexeme != "");

std::set<std::string> listOfDistinctOperators;

   // Insert elements from the vector into the set (duplicates will be automatically removed)
for (int i = 0 ; i < operators.size() ; i++)
{
    listOfDistinctOperators.insert(operators[i]);
}

    std::cout<<"\n\n============================\n";

std::cout<<"printing the list of operators:\n";
printList(operators);
std::cout<<"\n============================\n";



std::cout<<"\nprinting the list of distinct operators:\n";
printSet(listOfDistinctOperators);
std::cout<<"\n============================\n";


n1 = listOfDistinctOperators.size();
N1 = operators.size();


std::cout<<"\n n1 = " <<n1<<std::endl;
std::cout<<" N1 = " <<N1<<std::endl;
```

```
void countNumberOfOperator( std::string input){
    cout<<endl<<"Calculating Halstead complexity matrics...\n";
    std::vector <std::string> operators;

    Lexer lexer(input);
    Token token;

    do {

        token = lexer.getNextToken();
        if (LexerRegex::matchOperator(token.lexeme)) {
            operators.push_back(token.lexeme) ;
        }

        for (int i = 0 ; i < token.lexeme.size(); i++){
            if (isdigit(token.lexeme[i]) == false)
            {
                continue;
            }
            else
            {
                operands.push_back(token.lexeme);
            }
        }

        //check if lexeme is in identifier
        if (((isalpha(token.lexeme[0]))||(token.lexeme[0] == '_'))
            &&
            (( LexerRegex::matchKeyword(token.lexeme)  == false)
            &&
             (LexerRegex::matchDatatype(token.lexeme)  == false))  )
        {
            operands.push_back(token.lexeme);
       }

    } while (token.lexeme != "");
```

```
    std::cout<<"\n n1 = " <<n1<<std::endl;
    std::cout<<" N1 = " <<N1<<std::endl;

    std::cout<<"\n============================\n";

std::cout<<"printing the list of operands:\n\n";
printList(operands);
std::cout<<"\n============================\n";

    for (int i = 0 ; i < operands.size() ; i++)
    {
        distinctOperands.insert(operands[i]);
    }
std::cout<<"\nprinting the list of distinct operands:\n";
printSet(distinctOperands);
std::cout<<"\n============================\n";


n2 = distinctOperands.size();
N2 = operands.size();


std::cout<<"\n n2 = " <<n2<<std::endl;
std::cout<<" N2 = " <<N2<<std::endl;

}
```

Function that calls and prints the Halstead fucntions:

```
void getHalsteadMetric(int n1, int n2,int N1, int N2){
    cout<<endl<<endl;
    cout<<"Halstead complexity matrics:"<<endl;
    cout<<"  - Program length: "<< getProgramLength(N1,N2) <<endl;
    cout<<"  - Program vocabulary: "<< getProgramVocabulary(n1,n2) <<endl;
    cout<<"  - Estimated length: "<< getEstimatedLength(n1,n2) <<endl;
    cout<<"  - Purity Ratio: "    << getPurityRatio (n1, n2, N1, N2) <<endl;
    cout<<"  - Volume: "<< getVolume(n1,n2) <<endl;
    cout<<"  - Difficulty: "<< getDifficulty( n1, n2,  N2) <<endl;
    cout<<"  - Effort: "<< getEffort( n1, n2, N2) <<endl;
    cout<<"  - Number of bugs from volume: "<< getNumberOfBugs_Volume(n1,n2) <<endl;
    cout<<"  - Number of bugs from effort: "<< getNumberOfBugs_Effort(n1,n2, N2) <<endl;
    cout<< endl <<endl;
}
```

6. Cyclomatic Complexity

Cyclomatic complexity measures the number of independent paths through a program's control flow graph. It quantifies code complexity by counting decision points like loops and conditionals. Higher complexity indicates more intricate code, while lower complexity suggests simplicity. Developers use it to identify and refactor complex code sections, aiming to improve readability and maintainability.

6.1 CC Code

Split function:

```cpp
#include "cyclomatic_complexity.hpp"
#include <iostream>
#include <unordered_set>
#include <vector>
#include <string>
#include <sstream>

using namespace std;


// Function to split a string into tokens (words)
vector<string> split(const string& str) {
  vector<string> tokens;
  string token;
  for (char ch : str) {
    if (isalnum(ch) || ch == '_') {
      token += ch;
    } else if (!token.empty()) {
      tokens.push_back(token);
      token.clear();
    }
  }
  if (!token.empty()) {
    tokens.push_back(token);
  }
  return tokens;
}
```

This function **split** takes a string **str** as input and splits it into tokens (words), returning a vector of strings containing the tokens.

Algorithm:

- It iterates through each character **ch** in the input string.
- If **ch** is an alphanumeric character or an underscore, it adds it to the current token.
- When it encounters a non-alphanumeric character (i.e., delimiter), it checks if the current token is not empty. If so, it adds the token to the vector of tokens and clears the token.
- Finally, if there's any remaining token after the loop, it adds it to the vector of tokens.

Return:

- It returns the vector of tokens containing all the words extracted from the input string.

Count decision points function:

```
// Function to count decision points (predicates)
int countDecisionPoints(const string& code, vector<string> ) {
  static const unordered_set<string> keywords = {"if", "else", "switch", "case", "while", "for"};
  int decisionPoints = 0;
  for (const string& line : split(code)) {
    for (const string& keyword : keywords) {
      if (line.find(keyword) != string::npos) {
        decisionPoints++;
        break; // Only count the first occurrence per line
      }
    }
  }
  return decisionPoints;
}
```

This function **countDecisionPoints** takes a string **code** representing a piece of code and a vector of strings as input. It counts the decision points (predicates) in the code, such as if statements, loops, and switch cases.

Algorithm:

- It initializes a set **keywords** containing the keywords representing decision points in programming languages (e.g., if, else, switch, while, for).
- It initializes a counter **decisionPoints** to count the total number of decision points encountered.
- It iterates through each line of the code by splitting the input string **code** into lines using the **split** function.
- For each line, it iterates through each keyword in the **keywords** set.
- If the keyword is found in the line using the **find** function, it increments the **decisionPoints** counter and breaks out of the inner loop to count only the first occurrence of a decision point per line.

Return:

- It returns the total count of decision points found in the code

Calculating CC using McCabe's method:

```
// Function to calculate cyclomatic complexity using McCabe's method (same as before)
int calculateCyclomaticComplexityMcCabe(const string& code) {
  return countDecisionPoints(code) + 1;
}
```

Calculating CC using Edges-Nodes method:

```
// Function to calculate cyclomatic complexity using Edges-Nodes method
int calculateComplexityEdgesNodes(const string& code) {
  // 1. Preprocess code (remove comments, whitespaces)
  string preprocessedCode;
  for (char ch : code) {
    if (isalnum(ch) || ch == '_' || ch == '.' || ch == '{') {
      preprocessedCode += ch;
    }
  }

  // 2. Count basic blocks (sequences without branches)
  int basicBlocks = 0;
  bool inBlock = false;
  for (char ch : preprocessedCode) {
    if (ch == '{') {
      inBlock = true;
    } else if (ch == '}') {
      if (inBlock) {
        basicBlocks++;
      }
      inBlock = false;
    } else if (!isalnum(ch) && !inBlock) {
      basicBlocks++; // Assuming basic block ends with a non-alphanumeric outside curly braces
    }
  }

  // 3. Estimate edges based on control flow keywords and nesting levels
  int edges = 0;
  int nestingLevel = 0;
  for (char ch : preprocessedCode) {
    if (ch == '{') {
      nestingLevel++;
    } else if (ch == '}') {
      nestingLevel--;
    } else if (iscntrl(ch)) { // Control characters (including newline)
      edges += nestingLevel + 2; // Add 2 for potential entry/exit edges
    }
  }

  // 4. Estimate nodes based on basic blocks and nesting levels
  int nodes = basicBlocks + nestingLevel + 1; // Add 1 for starting node

  // 5. Calculate cyclomatic complexity
  return edges - nodes + 2;
}
```

This function calculates the cyclomatic complexity of a piece of code using the Edges-Nodes method. Here's a brief explanation:

1. **Preprocess Code**:
    - Removes comments and whitespaces from the input code, keeping only alphanumeric characters, underscores, periods, and curly braces.
2. **Count Basic Blocks**:
    - Identifies sequences of code without branches (basic blocks) by tracking opening and closing curly braces. Each basic block is counted.
3. **Estimate Edges**:
    - Determines the edges in the control flow graph based on control flow keywords and nesting levels. It considers the nesting level of code blocks and control characters (including newline) to estimate potential entry and exit edges for each block.
4. **Estimate Nodes**:
    - Calculates the nodes in the control flow graph based on the number of basic blocks and nesting levels, adding one for the starting node.
5. **Calculate Cyclomatic Complexity**:
    - Computes the cyclomatic complexity using the formula: Cyclomatic Complexity = Edges - Nodes + 2.
6. **Return**:
    - Returns the calculated cyclomatic complexity of the code

7. Main Function

```
int main() {

    // Sample source code string
    std::string  source_code = readSourceCode(sourse_code_file_name_and_path);
    //or use:
    //std::string input = getHardCoddedSourceCodeExample(3);

    predicate_complexity = calculateCyclomaticComplexityMcCabe(source_code);
    McCabe_complexity = calculateCyclomaticComplexityMcCabe(source_code);
    print_quality_matrics_report();
    Lexer lexer(source_code);
    Token token;
    extract_token_stream_from_input_file(source_code);
    extract_identifiers_from_token_stream(token_stream);
    extract_symbol_table_from_identifiers_list(identifiers);

    countNumberOfOperator(source_code);

    getHalsteadMetric( n1,  n2, N1,  N2);

    print_token_stream(token_stream);
    print_symbol_table();

    export_quality_matrics_report();
    return 0;
}
```

This **main** function orchestrates tasks for code analysis and quality metrics calculation:

1. **Read Source Code**: Reads the source code from a file or uses a hardcoded example.
2. **Calculate Cyclomatic Complexity**: Determines the cyclomatic complexity using the McCabe method.
3. **Print Quality Metrics Report**: Displays a summary of quality metrics.
4. **Lexical Analysis**: Initializes a **Lexer** object for further analysis.
5. **Tokenization**: Extracts a token stream from the source code.
6. **Identifier Extraction**: Collects unique identifiers from the token stream.
7. **Symbol Table Generation**: Constructs a symbol table from the extracted identifiers.
8. **Count Operators**: Calculates the number of operators in the source code.
9. **Halstead Metrics Calculation**: Computes Halstead metrics.
10. **Print Token Stream and Identifiers**: Displays the token stream and unique identifiers.
11. **Export Quality Metrics Report**: Saves the quality metrics report.

8. Sample Runs and Output

Example 1

code_example1.txt

```
int module1 (int x, int y) {
   while (x!=y){
      if(x>y)
         x=x-y,
      else y=y-x;
      }
   return x;
}
```

Token Stream 1

```
==============================================
printing token stream:
Token: int           Type: DataType
Token: module1       Type: Identifier
Token: (             Type: LeftParenthesis
Token: int           Type: DataType
Token: x             Type: Identifier
Token: ,             Type: Comma
Token: int           Type: DataType
Token: y             Type: Identifier
Token: )             Type: RightParenthesis
Token: {             Type: LeftBrace
Token: while         Type: Keyword
Token: (             Type: LeftParenthesis
Token: x             Type: Identifier
Token: !=            Type: NotEqualOperator
Token: y             Type: Identifier
Token: )             Type: RightParenthesis
Token: {             Type: LeftBrace
Token: if            Type: Keyword
Token: (             Type: LeftParenthesis
Token: x             Type: Identifier
Token: >             Type: GreaterThanOperator
Token: y             Type: Identifier
Token: )             Type: RightParenthesis
Token: x             Type: Identifier
Token: =             Type: AssignmentOperator
Token: x             Type: Identifier
Token: -             Type: SubtractionOperator
Token: y             Type: Identifier
Token: ,             Type: Comma
Token: else          Type: Keyword
Token: y             Type: Identifier
```

```
Token: =              Type: AssignmentOperator
Token: y              Type: Identifier
Token: -              Type: SubtractionOperator
Token: x              Type: Identifier
Token: ;              Type: Semicolon
Token: }              Type: RightBrace
Token: return         Type: Keyword
Token: x              Type: Identifier
Token: ;              Type: Semicolon
Token: }              Type: RightBrace
Token:                Type: UnknownToken
```

Identifiers 1

```
================================================
printing list of unique identifiers:
int
module1
int
x
int
y
while
x
y
if
x
y
x
x
y
else
y
y
x
return
x
================================================
```

Halstead metrics 1

```
Calculating Halstead complexity matrics...


============================
printing the list of operators:
!=  if  >   =   -   else    =   -
============================

printing the list of distinct operators:
!= - = > else if
============================

 n1 = 6
 N1 = 8

============================
printing the list of operands:

module1 module1 x   y   x   y   x   y   x   x   y   y   y   x   x
============================

printing the list of distinct operands:
module1 x y
============================

 n2 = 3
 N2 = 15
```

```
Halstead complexity matrics:
  - Program length: 23
  - Program vocabulary: 9
  - Estimated length: 14.0464
  - Purity Ratio: 0.610713
  - Volume: 30.8631
  - Difficulty: 15
  - Effort: 462.946
  - Number of bugs from volume: 0.0102877
  - Number of bugs from effort: 0.000333333
```

Cyclomatic Complexity 1

```
Cyclomatic Complexity matrics:

  - using Predicate Method: 4
  - using McCabe's Method: 4
```

Output Report 1

quality_matrics_report.txt

```
Quality matrics report:


//================================//
Halstead matrics:
Program length: 23
Program vocabulary: 9
Estimated length: 14.0464
Purity Ratio: 0.610713
Volume: 30.8631
Difficulty: 15
Effort: 462.946
Number of bugs from volume: 0.0102877
Number of bugs from effort: 0.000333333

//================================//




//================================//
Cyclomatic Complexity matrics:

  - using Predicate Method: 4
  - using McCabe's Method: 4

//================================//
```

Example 2

quality_matrics_report.txt

```
Quality matrics report:


//==================================//
Halstead matrics:
Program length: 30
Program vocabulary: 18
Estimated length: 41.3915
Purity Ratio: 1.37972
Volume: 119.637
Difficulty: 2
Effort: 239.274
Number of bugs from volume: 0.039879
Number of bugs from effort: 0.000333333

//==================================//




//==================================//
Cyclomatic Complexity matrics:

  - using Predicate Method: 3
  - using McCabe's Method: 3

//==================================//
```

Token Stream 2

```
printing token stream:
Token: int          Type: DataType
Token: main         Type: Identifier
Token: (            Type: LeftParenthesis
Token: )            Type: RightParenthesis
Token: {            Type: LeftBrace
Token: int          Type: DataType
Token: x            Type: Identifier
Token: =            Type: AssignmentOperator
Token: 10           Type: Number
Token: ;            Type: Semicolon
Token: if           Type: Keyword
Token: (            Type: LeftParenthesis
Token: x            Type: Identifier
Token: >            Type: GreaterThanOperator
Token: 5            Type: Number
Token: )            Type: RightParenthesis
Token: {            Type: LeftBrace
Token: cout         Type: Identifier
Token: <            Type: LessThanOperator
Token: <            Type: LessThanOperator
Token: "            Type: SingleQuote
Token: x            Type: Identifier
Token: is           Type: Identifier
Token: greater      Type: Identifier
Token: than         Type: Identifier
Token: 5            Type: Number
Token: "            Type: SingleQuote
Token: ;            Type: Semicolon
Token: }            Type: RightBrace
Token: else         Type: Keyword
Token: {            Type: LeftBrace
Token: cout         Type: Identifier
Token: <            Type: LessThanOperator
```

```
Token: }            Type: RightBrace
Token: else         Type: Keyword
Token: {            Type: LeftBrace
Token: cout         Type: Identifier
Token: <            Type: LessThanOperator
Token: <            Type: LessThanOperator
Token: "            Type: SingleQuote
Token: x            Type: Identifier
Token: is           Type: Identifier
Token: less         Type: Identifier
Token: than         Type: Identifier
Token: or           Type: Identifier
Token: equal        Type: Identifier
Token: to           Type: Identifier
Token: 5            Type: Number
Token: "            Type: SingleQuote
Token: ;            Type: Semicolon
Token: }            Type: RightBrace
Token: return       Type: Keyword
Token: 0            Type: Number
Token: ;            Type: Semicolon
Token: }            Type: RightBrace
Token:              Type: UnknownToken
```

Identifiers 2

```
================================================
printing list of unique identifiers:
int
main
int
x
if
x
cout
x
is
greater
than
else
cout
x
is
less
than
or
equal
to
return
================================================
```

Halstead metrics 2

```
Calculating Halstead complexity matrics...


==========================
printing the list of operators:
=   if  >   <   <   else    <   <
==========================

printing the list of distinct operators:
< = > else if
==========================

 n1 = 5
 N1 = 8

==========================
printing the list of operands:

main    x   10  10  x   5   cout    x   is  greater than    5   cout    x   is  less    than    or
equal   to  5   0
==========================

printing the list of distinct operands:
0 10 5 cout equal greater is less main or than to x
==========================

 n2 = 13
 N2 = 22
```

```
Halstead complexity matrics:
  - Program length: 30
  - Program vocabulary: 18
  - Estimated length: 41.3915
  - Purity Ratio: 1.37972
  - Volume: 119.637
  - Difficulty: 2
  - Effort: 239.274
  - Number of bugs from volume: 0.039879
  - Number of bugs from effort: 0.000333333
```

Cyclomatic Complexity 2

```
Cyclomatic Complexity matrics:

  - using Predicate Method: 3
  - using McCabe's Method: 3
```

Output Report 2

quality_matrics_report.txt

```
Quality matrics report:


//================================//
Halstead matrics:
Program length: 30
Program vocabulary: 18
Estimated length: 41.3915
Purity Ratio: 1.37972
Volume: 119.637
Difficulty: 2
Effort: 239.274
Number of bugs from volume: 0.039879
Number of bugs from effort: 0.000333333

//================================//




//================================//
Cyclomatic Complexity matrics:

  - using Predicate Method: 3
  - using McCabe's Method: 3

//================================//
```

Example 3

code_example3.txt

```
int main() {
    int x = 10;
    if (x > 5) {
        cout << "x is greater than 5";
        if (x > 10) {
            cout << " and greater than 10";
        }
    } else {
        cout << "x is less than or equal to 5";
    }
    return 0;
}
```

Token Stream 3

```
================================================
printing token stream:
Token: int            Type: DataType
Token: main           Type: Identifier
Token: (              Type: LeftParenthesis
Token: )              Type: RightParenthesis
Token: {              Type: LeftBrace
Token: int            Type: DataType
Token: x              Type: Identifier
Token: =              Type: AssignmentOperator
Token: 10             Type: Number
Token: ;              Type: Semicolon
Token: if             Type: Keyword
Token: (              Type: LeftParenthesis
Token: x              Type: Identifier
Token: >              Type: GreaterThanOperator
Token: 5              Type: Number
Token: )              Type: RightParenthesis
Token: {              Type: LeftBrace
Token: cout           Type: Identifier
Token: <              Type: LessThanOperator
Token: <              Type: LessThanOperator
Token: "              Type: SingleQuote
Token: x              Type: Identifier
Token: is             Type: Identifier
Token: greater        Type: Identifier
Token: than           Type: Identifier
Token: 5              Type: Number
Token: "              Type: SingleQuote
Token: ;              Type: Semicolon
Token: if             Type: Keyword
Token: (              Type: LeftParenthesis
Token: x              Type: Identifier
Token: >              Type: GreaterThanOperator
```

```
Token: (              Type: LeftParenthesis
Token: x              Type: Identifier
Token: >              Type: GreaterThanOperator
Token: 10             Type: Number
Token: )              Type: RightParenthesis
Token: {              Type: LeftBrace
Token: cout           Type: Identifier
Token: <              Type: LessThanOperator
Token: <              Type: LessThanOperator
Token: "              Type: SingleQuote
Token: and            Type: Identifier
Token: greater        Type: Identifier
Token: than           Type: Identifier
Token: 10             Type: Number
Token: "              Type: SingleQuote
Token: ;              Type: Semicolon
Token: }              Type: RightBrace
Token: }              Type: RightBrace
Token: else           Type: Keyword
Token: {              Type: LeftBrace
Token: cout           Type: Identifier
Token: <              Type: LessThanOperator
Token: <              Type: LessThanOperator
Token: "              Type: SingleQuote
Token: x              Type: Identifier
Token: is             Type: Identifier
Token: less           Type: Identifier
Token: than           Type: Identifier
Token: or             Type: Identifier
Token: equal          Type: Identifier
Token: to             Type: Identifier
Token: 5              Type: Number
Token: "              Type: SingleQuote
Token: ;              Type: Semicolon
Token: }              Type: RightBrace
```

```
Token: return         Type: Keyword
Token: 0              Type: Number
Token: ;              Type: Semicolon
Token: }              Type: RightBrace
Token:                Type: UnknownToken
```

Identifiers 3

```
================================================
printing list of unique identifiers:
int
main
int
x
if
x
cout
x
is
greater
than
if
x
cout
and
greater
than
else
cout
x
is
less
than
or
equal
to
return
================================================
```

Halstead metrics 3

```
Calculating Halstead complexity matrics...


============================
printing the list of operators:
=   if  >   <   <   if  >   <   <   else    <   <
============================

printing the list of distinct operators:
< = > else if
============================

 n1 = 5
 N1 = 12

============================
printing the list of operands:

main    x   10  10  x   5   cout    x   is  greater than    5   x   10  10
cout    and greater than    10  10  cout    x   is  less    than    or  equal
to  5   0
============================

printing the list of distinct operands:
0 10 5 and cout equal greater is less main or than to x
============================

 n2 = 14
 N2 = 31
```

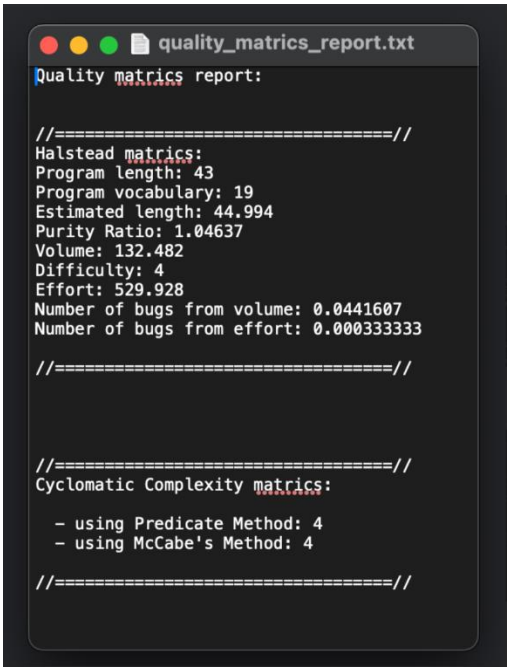
```
Halstead complexity matrics:
  - Program length: 43
  - Program vocabulary: 19
  - Estimated length: 44.994
  - Purity Ratio: 1.04637
  - Volume: 132.482
  - Difficulty: 4
  - Effort: 529.928
  - Number of bugs from volume: 0.0441607
  - Number of bugs from effort: 0.000333333
```

Cyclomatic Complexity 3:

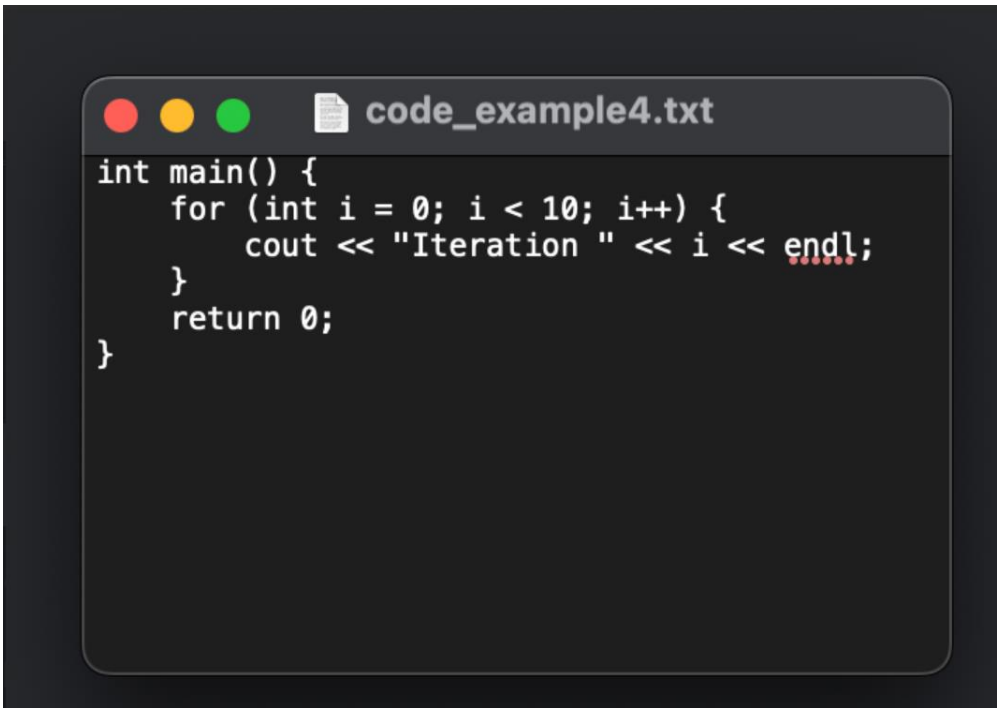
```
Cyclomatic Complexity matrics:

  - using Predicate Method: 4
  - using McCabe's Method: 4
```

Output Report 3



Example 4


code_example4.txt
int main() {
for (int i = 0; i < 10; i++) {
cout << "Iteration " << i << endl;
}
return 0;
}

Token Stream 4

```
==========================================
printing token stream:
Token: int          Type: DataType
Token: main         Type: Identifier
Token: (            Type: LeftParenthesis
Token: )            Type: RightParenthesis
Token: {            Type: LeftBrace
Token: for          Type: Keyword
Token: (            Type: LeftParenthesis
Token: int          Type: DataType
Token: i            Type: Identifier
Token: =            Type: AssignmentOperator
Token: 0            Type: Number
Token: ;            Type: Semicolon
Token: i            Type: Identifier
Token: <            Type: LessThanOperator
Token: 10           Type: Number
Token: ;            Type: Semicolon
Token: i            Type: Identifier
Token: ++           Type: IncrementOperator
Token: )            Type: RightParenthesis
Token: {            Type: LeftBrace
Token: cout         Type: Identifier
Token: <            Type: LessThanOperator
Token: <            Type: LessThanOperator
Token: "            Type: SingleQuote
Token: Iteration    Type: Identifier
Token: "            Type: SingleQuote
Token: <            Type: LessThanOperator
Token: <            Type: LessThanOperator
Token: i            Type: Identifier
Token: <            Type: LessThanOperator
Token: <            Type: LessThanOperator
Token: Iteration    Type: Identifier
Token: "            Type: SingleQuote
Token: <            Type: LessThanOperator
Token: <            Type: LessThanOperator
Token: i            Type: Identifier
Token: <            Type: LessThanOperator
Token: <            Type: LessThanOperator
Token: endl         Type: Identifier
Token: ;            Type: Semicolon
Token: }            Type: RightBrace
Token: return       Type: Keyword
Token: 0            Type: Number
Token: ;            Type: Semicolon
Token: }            Type: RightBrace
Token:              Type: UnknownToken
```

Identifiers 4

```
================================================
printing list of unique identifiers:
int
main
for
int
i
i
i
cout
Iteration
i
endl
return
================================================
```

Halstead metrics 4

```
Calculating Halstead complexity matrics...


============================
printing the list of operators:
for =   <   ++  <   <   <   <   <   <
============================

printing the list of distinct operators:
++ < = for
============================

 n1 = 4
 N1 = 10

============================
printing the list of operands:

main    i   0   i   10  10  i   cout    Iteration   i   endl    0
============================

printing the list of distinct operands:
0 10 Iteration cout endl i main
============================

 n2 = 7
 N2 = 12
```

```
Halstead complexity matrics:
  - Program length: 22
  - Program vocabulary: 11
  - Estimated length: 19.1665
  - Purity Ratio: 0.871207
  - Volume: 45.9594
  - Difficulty: 2
  - Effort: 91.9187
  - Number of bugs from volume: 0.0153198
  - Number of bugs from effort: 0.000333333
```

Cyclomatic Complexity 4

```
Cyclomatic Complexity matrics:

  - using Predicate Method: 2
  - using McCabe's Method: 2
```

Output Report 4