UNIT-3

Software Implementation

Converts program structures into actual code in some programming language

Pre-Implementation

Choose language for compilation and interpretation

- For example, if performance is critical, a low-level language like C++ might be suitable. If rapid development is a priority, a high-level language like Python could be chosen.

Programming Language

- Assembly Level - map directly onto CPU Architecture
- Procedural - modest level of abstraction from underlying arch
- Aspect Oriented - allows dev to separate aspects within a program

- Object Oriented - further abstract the machine and develop in terms of objects

Choose Development Environment

- Opt for a development environment (IDE or code editor) that suits the project's needs. Consider features, compatibility with the chosen programming language, ease of use, and collaboration support. A well-selected development environment enhances productivity, facilitates teamwork, and streamlines the development process.

Choice of Development Environment

- Commercial VS Opensource
- Support of Development Process
    - Ensure that the chosen environment supports the development workflow of the team. This includes version control integration, build tools, and debugging capabilities.
- Integration between tools and environment
    - Ensure compatibility with other tools crucial to the development process, such as version control systems, issue tracking, and project management tools.
- Security

Follow Configuration Management Plan

- Implement a robust Configuration Management Plan covering version control, build management, change control, release management, and documentation management. This ensures organized and controlled development, enabling the team to track changes, manage configurations, and maintain documentation effectively throughout the software development lifecycle.

Characterising of Software Construction

- Construction produces high volume of configuration items
    - Source files, test cases

- Construction is tool-intensive
  - Relies on tools such as compilers, debuggers, GUI builders, etc
- Related to software quality
  - Code is the ultimate deliverable of a software project
- Extensive use of Computer Science knowledge
  - Algorithms, detailed coding practices

Implementation

Detailed creation of working Software through a combination of coding, reviews and Unit Testing.

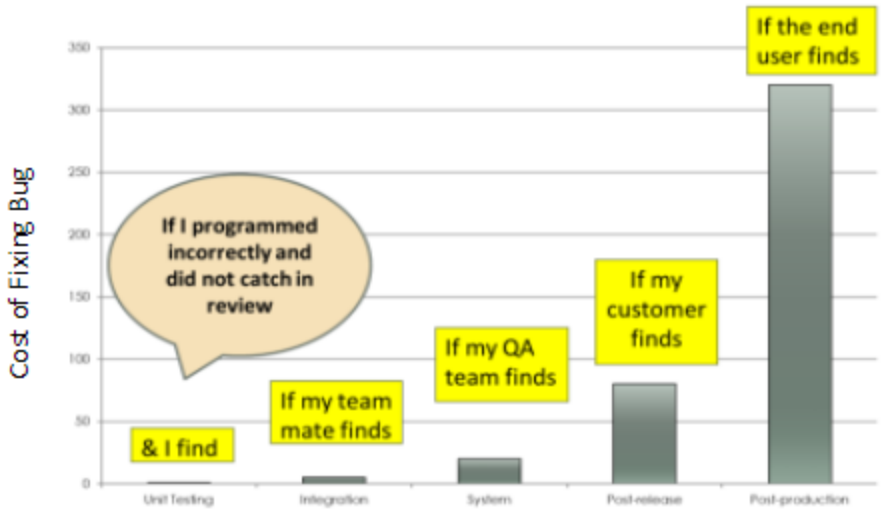
1. **Use Coding Standards and Guidelines:**
   - Maintain consistency in code formatting and documentation.
   - Adhere to best practices for the chosen programming language.
   - Ensure comprehensive documentation for better code understanding.
2. **Follow Language Syntax:**
   - Write code with correct syntax to prevent errors.
   - Utilize language features efficiently for optimal performance.
   - Implement effective error handling mechanisms.
3. **Address Quality, Security, and Testability:**
   - Prioritize code quality with clean, modular, and maintainable code.
   - Incorporate security best practices to prevent vulnerabilities.
   - Design code with testability in mind and implement thorough testing.
4. **Provide Unit Tested, Peer-Reviewed Functioning Code:**
   - Develop and execute unit tests to validate code functionality.
   - Conduct peer reviews to gather feedback and ensure adherence to standards.

- Integrate code into the main repository only after passing tests and reviews.

Implementation Goal

- Minimize Complexity
- Anticipate Change
- Verifable (testing and validation)
- Reuse
- Readable

Coding to catch a bug



1. Coding:

- **Activity:** Writing the initial code.

2. Code Review:

- **Activity:** Peers or team members review the code for quality, adherence to coding standards, and potential issues.

3. Unit Testing:

- **Activity:** Testing individual units or components of the code in isolation.

4. Integration Testing:

- **Activity:** Testing the interaction between different components or modules.

5. System Testing:

- **Activity:** Evaluating the entire system's functionality.

6. Field Trials:

- **Activity:** Deploying the software in a controlled environment or to a limited set of users.

7. Production:

- **Activity:** Software is live and accessible to users.

For Agile Scrum Project:

- **Sprint Breakdown:**
  - Completion of work through sprint story points.
- **Team Velocity Metric:**
  - Amount of software/stories completed during a sprint.
- **Throughput:**
  - Total value-added work output.
- **Cycle Time:**
  - Time between the start of an item and its completion.

Programmer Personality

- Under documenter
- CYA Specialist

- CIO types (pass responsibility to other modules)
- Dynamic types (create variables on the fly)
- Fakers (have repetitive code)
- Multitasker (uses wrappers and glue instead of rewriting)
- True Believer (Extensive Documentation)

Code Characterstics

- **Simple and Clear**
  - Should have reasonable lines per block , function , arguments and nesting loops
- **Naming**
  - meaningful name with no special characters at start
  - Pascal Casing for variable names
  - Camel Casing for methods
  - Underscore prefix for internal use in class
  - No variable name is a keyword
  - Name cannot start with an underscore
  - Not longer than 31 characters
  - Not contain numerical value
  - Descriptive
- **Structure:** Dependencies between software components (subprograms, parameters, code blocks, etc)
  - **Dependencies** can be highlighted via proper naming and layout
  - Well Encapsulated and logically grouped
  - Well Partitioned
- **Easy to read and understand**

- Visual layout of the code is maintained with standardized indentation use of comments to inform the meaning of the code

Do's	Don'ts
1. Use a few standards and agreed upon control constructs	1. Don't be too clever
2. Use GOTO in a disciplined manner Use user-defined data types to model entities in the problem domain	2. Avoid null Then statements (dangling if)
3. Hide data structures behind access functions	3. Avoid Then if statements
4. Use appropriate variable names	4. Don't nest too deeply
5. Use indentation, parentheses, blank spaces and lines to enhance readability	5. Don't use the same identifier for multiple purposes
	6. Examine routines that have more than five formal parameters

Coding Standards

- Provides a uniform appearance to code written by different engineers
- Improves readability, maintainability and reduces complexity
- Proactively addresses some commonly occurring issues with code
- Helps in code reuse
- Promotes sound programming practices and increases efficiency of programmers

Standards :- Rules to be followed

- Standard headers for different modules
- Naming conventions for local and global variables

Guidelines :- Rules recommended to follow

- One declaration per line
- Avoid magic numbers

Defensive Programming (code can fail)

Defensive programming is a coding approach that aims to anticipate and minimize the impact of potential issues by incorporating checks, validation, and error handling in the code. It involves assuming that anything that can go wrong will go wrong and taking steps to mitigate such situations

- Redundant code is incorporated to check system state after modifications
- Implicit assumptions are tested explicitly

```
pid = fork();
If (pid==0)
{/* child process is created; execute commands */}
else
{*/ parent process commands are executed */}
```

modify this code by adding a perror case when pid < 0

Secure Programming (intruder security)

Developing computer software to guard against accidental introduction of security vulnerabilities (Defects, Bugs, Logic Flaws)

1. **Validate Input:**
    - Validate and sanitize input from untrusted sources to prevent injection attacks and ensure data integrity.
2. **Heed Compiler Warnings:**
    - Compile with the highest warning level, address warnings, and enhance code quality for improved security.
3. **Use Static and Dynamic Analysis Tools:**
    - Employ automated tools for static and dynamic code analysis to detect and rectify security flaws in the codebase.
4. **Default Deny:**

- Base access decisions on permissions and follow a "deny by default" approach to reduce the attack surface.

5. **Adhere to Principle of Least Privilege:**
    - Grant the minimum necessary access or permissions to processes and users to minimize the risk of unauthorized actions.
6. **Elevated Permissions for Least Amount of Time:**
    - Provide elevated permissions only when essential and for the shortest duration possible to limit the window of vulnerability.
7. **Sanitize Data Sent to Other Systems:**
    - Cleanse and validate all data before sending it to external systems to mitigate the risk of injection attacks and maintain data integrity.

Testable Programming (fix bugs)

Software that is easy to test , find and fix bugs

1. Assertions: Asserting helps check if certain conditions in the code are true, like making sure a variable has a valid value.
2. Test Points: Test points help you keep track of where your code is during testing, making it easier to spot and fix issues.
3. Scaffolding: Scaffolding is creating temporary code to imitate a part of the system you're testing.
4. Test Harness: A test harness is code that helps you control and test different parts of your program, as if they were part of the whole system.
5. Test Stubs: Test stubs provide fake responses or values to test a specific part of the code without involving everything else.
6. Instrumenting:Instrumenting involves adding code to log or record what's happening during the execution of your program. It helps understand the flow and identify issues.

7. Building test data sets: reating various sets of data (both valid and invalid) to see how the program responds. This helps ensure the code handles different scenarios correctly.

Refactoring Code:

Refactoring code involves making improvements to the internal structure of a program's source code while preserving its external behavior.

Goals of Refactoring:

1. **Improve Design, Structure, and Implementation:**
   - *In Programming:* Enhance the structure and organization of the code without altering how it works.
2. **Improve Objective Attributes:**
   - *In Programming:* Focus on aspects like code length, duplication, and other factors that affect maintenance.
3. **Help Code Understanding:**
   - *In Programming:* Make the code more readable and comprehensible for developers.

Coding Guidlines

Provide uniform apperarance to the code reduces cost of error correction , complexity and maintenance over lifetime in project

Few Guidlines

1. Code should be well Documented
2. Indent code with spaces and lines
3. Minimize the length of functions
4. Reduce the usage of GOTO statements
5. Avoid using identifiers for multiple purposes

Software Configuration Management

Software Configuration Management: Process to systematically organize, manage and control changes in documents, code and other entities that constitute a software product

Goal: Increase productivity by increased and planned coordination among the programmers and eliminates confusion and mistakes

Need for SCM

- Multiple people work on the same project
- Working on different version and on released system
- Cordination between stakeholders
- Cost management

SCM in Agile Approach

- Whole developer team has access to the project repository
- Developer copy a version and make changes to the code and copy the same environment to their system
- Once changes are made and approved it is pushed to the repository
- New modified code is available to the whole team

Benefits of SCM

- Orderly release and implementation new software items they are revised sofware products
- Only approved change is implemented to the new system
- Documentation reflects the new updates
- Prevents unauthorized changes

Configuration Mangement Roles

Configuration Manager

- Identify configuration items
- Defines procedures for creating and releases the updates

Developer

- Resolves conflict
- new updates
- create versions on change request

Auditor

- Evaluates and validates process of release and updates
- Ensure consistency and completeness

Change Control Board

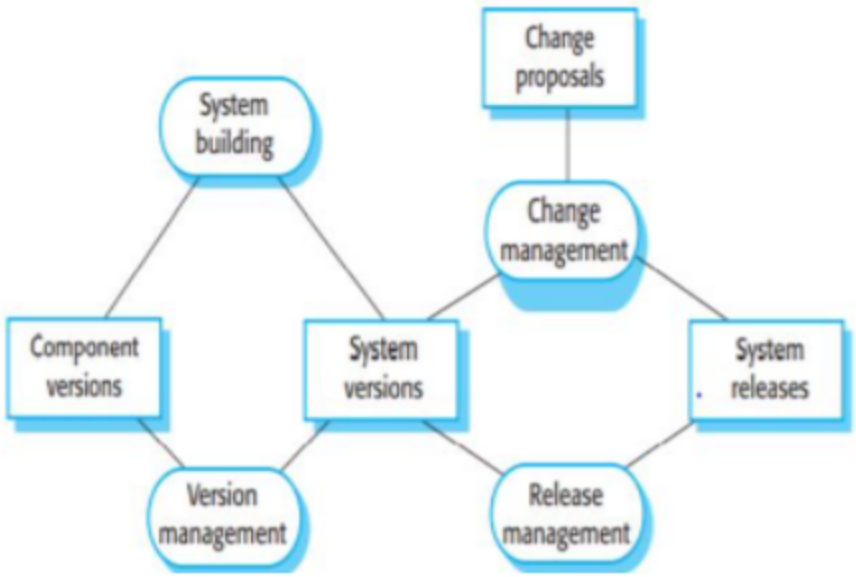
Approves or rejects releases and change requests

SCM Planning

Software Configuration Management Plan

Done by configuration manager

- May follow a public or internal standard
- Defines the Configuration Items and a naming scheme
- Define who takes responsibility for Configuration Management procedures and creation of baselines
- Defines policies for change control and version management
- Describes tools to assist in Configuration Management and any limitations
- Defines the configuration management database



1. Configuration item Identification
2. Configuration Management Directories
3. Baselining
4. Branch Management
5. Version Management
6. Build Management
7. Install
8. Promotion Management
9. Change Management
10. Release Management
11. Defect Management

1) Configuration Item (CI):

- **Definition:** An independent or aggregated entity of hardware, software, or both, designated for configuration management, treated as a single unit.
- **Examples of Software Configuration Items:**
  - Code files and test drivers.
  - Requirement, analysis, design, test, and other non-temporary documents.
  - User or developer manuals.
  - System configurations.

Challenges Associated with Configuration Items:

- **Identifying Items Needing Configuration Control:**
  - Some items must be maintained for the software's lifetime, akin to object modeling.

When to Place Entities Under Configuration Control:

- **Early Start:**
  - **Pros:** Ensures a systematic approach to CM.
  - **Cons:** May introduce bureaucracy if done excessively early.
- **Late Start:**
  - **Pros:** Avoids unnecessary overhead.
  - **Cons:** Can lead to chaos, especially in larger projects.

2) SCM Directories

Programmer's Directory
(Dynamic Library)

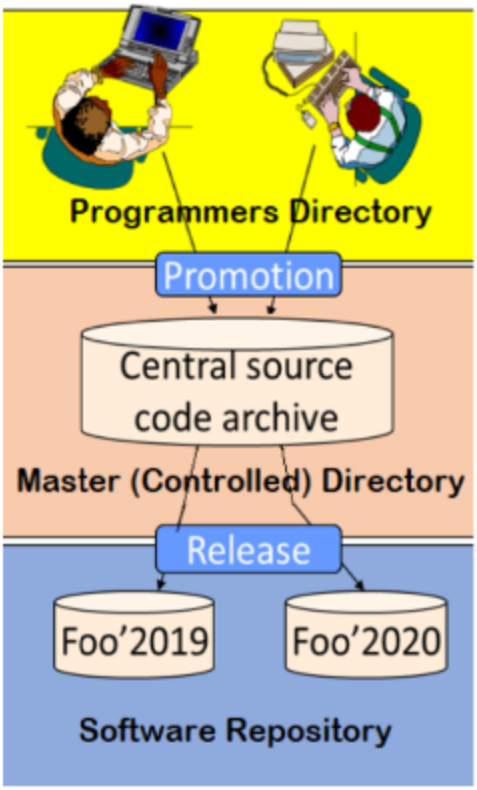
- Library for holding newly created or modified software entities
- Controlled by the programmer

Software Repository
(Static Library)

- Archive for various baselines in general use
- Copies may be made available on request

Master Directory
(Controlled Library)

- Manages current baselines and for controlling changes
- Entry is controlled, after verification
- Changes must be authorized



3) Baseline

Specification or product that has been formally reviewed and agreed to by responsible management and serves as a basis and can only be changed on formal review.

A system gets develped and becomes baseline after approval

Baseline A :- Developmental

Review and Approval

Baseline B :- First Prototype

Review and Approval

Baseline C :- Beta Test

Review and Approval

Official release of product

4) Branch Management

Codeline :- Source code

Branch :- Copy of the source code

Reasons for Branching

- Support concurrent development
- Can get new ideas and experiment with code
- The source code is untouched and safe
- Minimize conflicts

Branching Types

- Single Branch
- Branch by customer
- Branch by developer
- Branch by module

5) Version Management

Version Management: keeping track of different versions of software components and systems

- Usage of tools like git for version management
- Changes to a version are identified by a number, termed the revision number (7.5.5)
    - 7 – Release number (defined by customer)
    - 5 – Version number (defined by developer)
    - 5 – Revision number (defined by developer)
- All changes are traceable
- Change history is recorded and can be reverted back
- Better conflict resolution
- Easier code maintenance and monitoring
- Less software regression
- Better organisation and communication

6) Build Management

- **Definition:** The process of creating the application program for a software release by compiling and linking source code and libraries to generate build artifacts such as binaries or executables.
- **Objective:** Ensure that the software is built accurately, efficiently, and consistently, allowing for reliable and reproducible releases.

Tools Used:

- **Make, Apache Ant, Maven, etc.:**
    - Automated build tools that manage the compilation, dependency resolution, and packaging processes, streamlining the build management workflow.

Key Aspects of Build Management:

1. **Compilation and Linking:**
    - **Description:** Involves translating source code into machine-readable code (compilation) and combining it with necessary libraries and dependencies (linking).
    - **Importance:** Ensures the creation of executable software from source code.
2. **Avoiding Unnecessary Recompilation:**
    - **Description:** Build tools are configured to recompile only the modified or dependent files, reducing build time.
    - **Importance:** Efficient use of resources and faster build times when there are minimal changes.

Build Process:

1. **Fetching Code from Source Control Repository:**
2. **Compiling Code and Checking Dependencies:**
3. **Linking Libraries, Code, etc.:**
4. **Running Tests and Building Artifacts:**
5. **Archiving Logs and Sending Notification Emails:**
6. **Version Number Change:**

7) Install Management

Makes sure that customer can run the software or application in their device

- This is done by placing multiple files containing executable code, downloading or copying from a repository, and handling images, libraries, and configuration files from the internet.
- This ensures that all necessary components are correctly placed for the software to run.

- Make sure the software is allowed to be run in users device by adhering to the agreement license
- Installation may be automated using tools like zip, shell scripts, InstallAware, and Jenkins.

8) Promotion Management

- Changes made by a programmer is only available in their environment and needs to be promoted to a central master directory
- Promotion is based of promotion policy
- They can be further authorized to make change in master directory

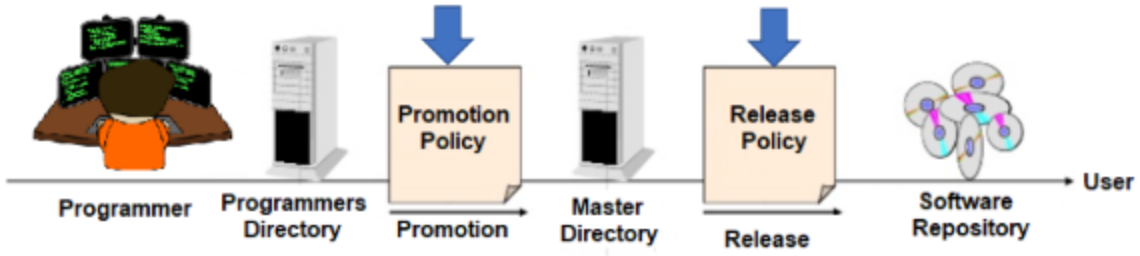
9) Change Management

- Approves the change to the software
- Deals with change in configuration Items which are baselined
- Change can be requested by anyone and is stored in the log unique id
- Change is assessed based on impact on directory , modules branches etc.
- Its either accepted or rejected
- If accepted change is implmented
- Implemented change is audited
- Complex Projects have Change Control Board (CCB) who approve the change.

Information required to process a change to a baseline

- Description of proposed changes
- Reasons for making the changes
- List of other items affected by the changes
- Tools, resources and training are required to perform baseline change assessment

- File comparison tools to identify changes
- Other resources and training depending on size and complexity of project



10) Release Management

Movement from master directory to software repository

- Releases it to customer or make change to the repository
- **Release Policy: Gating quality criteria that is planned for and includes verification with metrics**
- **Definition:** The management, planning, scheduling, and control of a software build through various stages and environments, encompassing testing and deployment of software releases.
- **Importance:** Ensures systematic and controlled progression of software through different phases, guaranteeing stability and reliability.

Release vs Version vs Revision:

1. **Release:**
    - **Definition:** A formal distribution of an approved version.
    - **Characteristics:**
        - Marks a significant milestone.
        - Represents a stable and complete version of the software.
        - Ready for distribution to users.
2. **Version:**

- **Definition:** An initial release or re-release of a configuration item associated with a complete compilation or recompilation of the item.
- **Characteristics:**
  - Different versions have distinct functionalities.
  - Can include improvements, new features, or bug fixes.
  - Versions are often identified by numbers (e.g., version 1.0, 2.0).

3. **Revision:**
   - **Definition:** A change to a version that corrects errors in the design/code but does not affect the documented functionality.
   - **Characteristics:**
     - Focused on fixing identified issues without altering functionality.
     - Typically represented by a change in the revision number (e.g., version 1.1, 1.2).

Key Distinctions:

- Releases are formal distributions, marking complete and stable versions.
- Versions represent distinct functionalities and can include improvements or bug fixes.
- Revisions are changes to a version focused on correcting errors without altering functionality.

Software Management Tools

Software Code Administration

- RCS - Very Old
- ClearCase(Linux,Solaris,Window) -Multiple servers , process modelling
- GitHub - Development platform for version control and project management

- CVS (Concurrent Version Control) - allows concurrent working without modelling , has frontend

GitHub

- Can create repositories and add contributors
  - Access the repository using ssh or https
- Changes are pushed as commits
  - Commits are logged with a unique commit number and message
- Master branch is auto cloned
  - Different branches can be created to add code without impacting the stable version
- General contribution method
  - Fork a project
  - Make changes
  - Generate a pull request to merge new code (Are all pull requests merged?)
  - Whenever merged, a difference is done and changes are logg

```
git init (intialise repo)
git status (shows commits)
git add <files> (adds untracked files to git)
git commit -m <message> (commits with an message)
git push (pushes to local repository)
git remote -v (shows all the remote repositories in your syst
```

Commands
• Git init
• Git status
• Git add

- Git commit -m
- Git remote add origin
- Git remote -v
- Git push
- Git push origin

Software Build

Converts source code into standalone software artefact

- Make :- automatically makes the makefile
- CruiseControl :- Open source tool for continuous software build
- FinalBuilder :- Automate build and release management tools
- Maven :- Handles compilation, distribution, documentation, team collaboration and other tasks

MakeFile

A Makefile is a special file used in software development to define a set of tasks and their dependencies. The `make` command reads the Makefile and executes the specified tasks. Here are some basic commands commonly used in a Makefile:

1. `all`:
   - **Description:** The default target that is executed when you run `make` without specifying a target.
   - **Example:**

     ```
     all:
         gcc -o myprogram main.c
     ```

2. `clean`:
   - **Description:** Typically used to remove generated files or clean up the project directory.

- **Example:**

```
clean:
    rm -f myprogram
```

3. `target`:
   - **Description:** Defines a specific task or output file and its dependencies.
   - **Example:**

```
myprogram: main.c
    gcc -o myprogram main.c
```

4. `variable`:
   - **Description:** Defines a variable to store values that can be reused in multiple places within the Makefile.
   - **Example:**

```
CC = gcc
myprogram: main.c
    $(CC) -o myprogram main.c
```

5. `phony target`:
   - **Description:** A target that doesn't represent a file but is used for grouping other targets or for defining tasks that don't produce output files.
   - **Example:**

```
.PHONY: clean
clean:
    rm -f myprogram
```

Typical Makefile

```
CC = gcc  #indicates the compiler being used
CFLAGS = -g
LDFLAGS =  #to inform the compiler to include any linkers

all: helloworld
helloworld: helloworld.o
	$(CC)$(LDFLAGS) -o $@ $^
	#@ is used to indicate current target (helloworld)
	#Question: what does the -o flag indicate?

helloworld.o: helloworld.c
	 $(CC)$(LDFLAGS) –c -o $@ $<

clean: FRC rm –f helloworld helloworld.o
	#Question: Is this command necessary? What is its role?
```

In the provided Makefile, several common conventions and rules for building a simple C program named "helloworld" are used. Let's break down the key components:

1. **Variables:**
   - **CC (Compiler):** `CC = gcc`
     - Indicates the compiler being used (in this case, the GNU Compiler Collection).
   - **CFLAGS (Compiler Flags):** `CFLAGS = -g`
     - Compiler flags, in this case, `g` is used to include debugging information in the executable.
   - **LDFLAGS (Linker Flags):** `LDFLAGS =`
     - Linker flags, which are currently empty. This can be used to include specific linker options.
2. **Targets:**

- `all:`
  - The default target, which will be executed when you run `make` without specifying a target. In this case, it depends on the target `helloworld`.
- `helloworld:`
  - The target for building the executable "helloworld." It depends on the target `helloworld.o`.
  - The line `$(CC) $(LDFLAGS) -o $@ $^` links the object file "helloworld.o" to create the executable "helloworld."
    - `o $@`: Specifies the output file (in this case, the executable named "helloworld").
    - `$^`: Represents all the prerequisites (in this case, only "helloworld.o").
- `helloworld.o:`
  - The target for building the object file "helloworld.o" from the source file "helloworld.c."
  - The line `$(CC) $(LDFLAGS) -c -o $@ $<` compiles the source file into an object file.
    - `c`: Indicates compilation without linking.
    - `o $@`: Specifies the output file (in this case, "helloworld.o").
    - `$<`: Represents the first prerequisite (in this case, "helloworld.c").
- `clean:`
  - A target for cleaning up the project by removing generated files. It depends on the target `.PHONY` (a special target indicating that it is not a file) and executes the command `rm -f helloworld helloworld.o`.
    - `rm -f`: Removes the specified files forcefully.
    - "helloworld" and "helloworld.o" are the files to be removed.

3. **Special Rules:**

- `.PHONY:`
  - A special target indicating that the target names following it are not real files but rather actions to be performed.
  - In this case, it's used to declare the "clean" target as a phony target.

Answering the Questions:

- **Question 1: What does the `o` flag indicate?**
  - In the context of the Makefile, the `o` flag specifies the output file name. For example, in the line `$(CC) $(LDFLAGS) -o $@ $^`, it indicates that the output file for the linking step is "helloworld" (represented by `$@`).
- **Question 2: Is the `clean` command necessary, and what is its role?**
  - The `clean` command is not strictly necessary, but it is a good practice. Its role is to remove generated files (e.g., the executable and object files) to clean up the project directory. This ensures that the project can be built from a clean state, and it's helpful when you want to ensure that there are no remnants of previous builds when rebuilding the project. The `f` option in `rm -f` allows the removal to proceed even if the files don't exist, preventing error messages in case the files are already deleted.

Maven

- Simplifies and standardizes the project build process
- Handles compilation, distribution, documentation, team collaboration and other tasks
- Increases reusability and takes care of most build tasks
- Supports multiple development team environments
- Supports creation of reports, checks, build and testing automation
- Standard directory layout and default build lifecycles with environment variables

Maven Goals:

- `clean`:
  - Removes the `target` directory and any compiled files.
- `compile`:
  - Compiles the source code.
- `test`:
  - Runs unit tests.
- `package`:
  - Packages the compiled code into a distributable format (e.g., JAR or WAR).
- `install`:
  - Installs the package into the local repository for use as a dependency in other projects.

Software Installation Tool

Installer: installs all the necessary files on the system

Bootstrapper: small installer that does the pre-requisites and updates the big bundle

DeployMaster (Windows)

- Distribute windows software or files via internet/CD/DVD
- Works with different versions of windows

InstallShield

- Simplifies creation of windows installers, MSI packages, and InstallScript installers for Windows
- De-facto for MSI installations

InstallAware (Windows)

- Windows installer platform for MS windows OS
- Supports internet deployment

Wise Installer

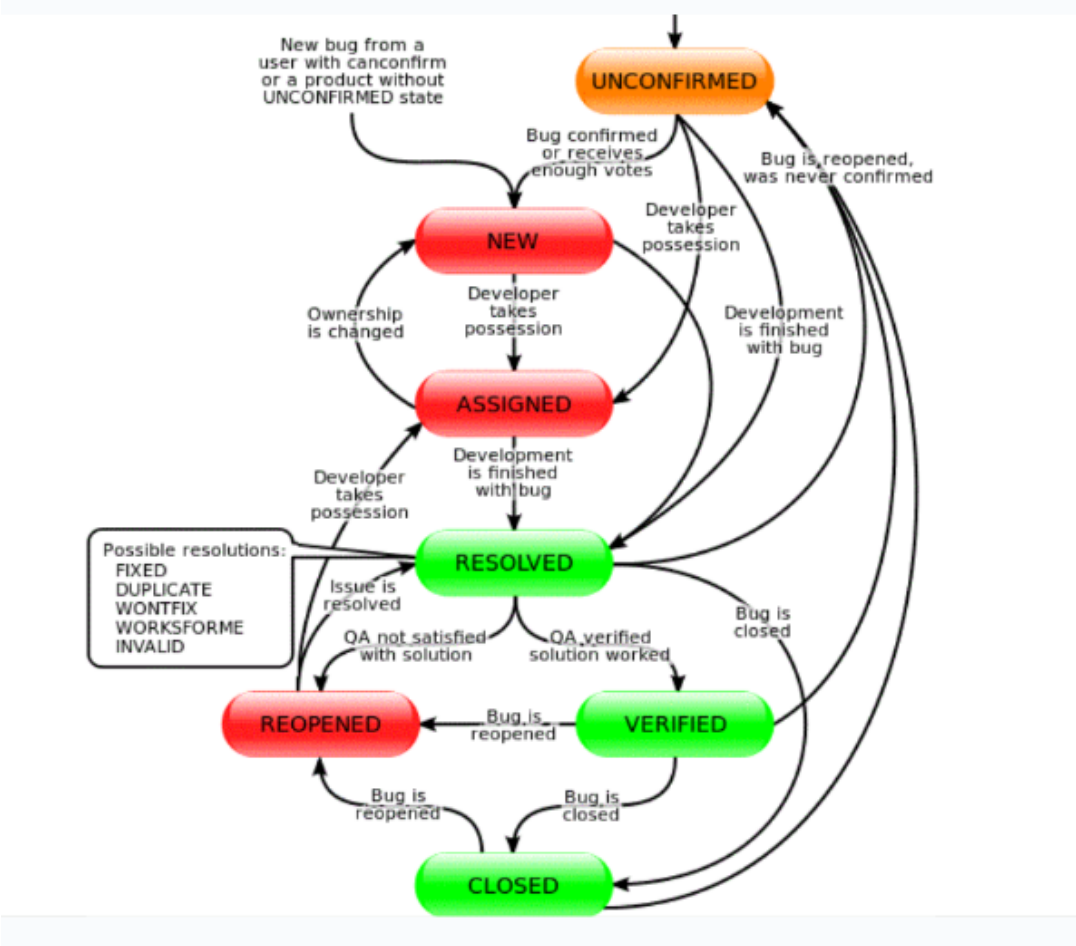
- Configure and install Microsoft windows applications

Software Bug Tracking Tool

Software application that keeps track of reported software bugs in a software project

Bug tracking systems support the lifecycle of a bug from logging to resolution

Examples: Bugzilla, FogBugz, Trac Edgewall, Backlog, etc



Software Quality

- Software systems are complex and evolve leading to need for continuous assessment and evaluation of quality
- Bad quality of software could impact experiences in life can also lead to dissatisfied customers
- Software quality enhances long term profitability of products

Product Operation Perspective

- **Correctness:** Does it do what I want?
- **Reliability:** Is it always accurate?
- **Efficiency:** Does it run as well as it can?
- **Integrity:** Is it secure?
- **Usability:** Can I use it?
- **Functionality:** Does it have necessary features?
- **Availability:** Will the product always run when needed?

Overall Environment Perspective

- **Responsiveness:** Can I quickly respond to change
- **Predictability:** Can I always predict the progress?
- **Productivity:** Will things be done efficiently?
- **People:** Will the customers be satisfied?
  - Will the employees be gainfully engaged?

Product Revision Perspective

- **Maintainability:** Can I fix it?
- **Testability:** Can I test it?
- **Flexibility:** Can I change it?

Product Transition Perspective

- **Portability:** Can it be used on another machine?
- **Reusability:** Can I reuse some/all of the software?
- **Interoperability:** Can I interface it with another system?

FLURPS

Functionality - features

Localizable - available in local language

Usability - documentation

Reliability - less chance of failure

Performance - speed and throughput

Supportability - maintainabilty

Software Terminologies

1. Attribute - physical or abstract property of entity

2. Measure - Quantitative indication like number of errors

   Measure is used in

   - **Feedback: quantifies some of the attribute to help improve the product**
   - **Diagnostics: helps identify issues towards quality**
     - Supports evaluating and establishing productivity
   - **Forecasting: to predict future need and anticipate maintenance**
     - Supports estimating, budgeting, costing and scheduling

3. Metric : No of errors per person

   Quantitative measure of degree to which a system possesses a given attribute

- **Quantitative: Metrics should be quantitative and expressible in values**
- **Understandable: Metric computation should be defined and easily understood**
- **Applicability: Should be applicable at all stages of software development**
- **Repeatable: Metrics are consistent and same when measured again**
- **Economical: Computation of metric should be economical**
- **Language Independent: Metrics should not depend on programming language**

Differences

1. **Nature:**
   - **Attribute:** Describes a property.
   - **Measure:** Quantitative representation of an attribute.

- **Metric:** Quantitative measure with a calculation involving two or more measures.

2. **Examples:**
   - **Attribute:** Complexity, reliability, maintainability.
   - **Measure:** Number of errors, lines of code.
   - **Metric:** Defect density (errors per person-hours), lines of code per function point.

Cost of Good Quality	Cost of Bad Quality
Prevention costs: Investments to prevent/avoid quality problems E.g.: Error proofing, improvement initiatives	**Internal failure costs:** costs associated with defects found before the customer receives the product E.g.: Rework, Re-testing
Appraisal costs: costs to determine degree of conformance to requirements and quality standards E.g.: Quality Assurance, Inspection	**External failure costs:** costs associated with defects found after customer receives product E.g.: Support Calls, Patches
Management Control costs: costs to prevent or reduce failures in management functions E.g.: contract reviews, gating/release criteria	**Technical debt:** cost of fixing a problem, which left unfixed, puts the business at risk E.g.: Structural problems, Increased Complexity
	Management failures: costs incurred by personnel due to poor quality software Eg: Unplanned costs, customer damages

Software Metrics Categories

1. **Direct Measures** (internal attributes)

- Depends only on value
- Other attributes are measured with respect to these
- E.g.: Cost, Effort, LoC, Duration of testing

2. **Indirect Measures** (External Attributes)

- Derived from direct measures
- E.g.: Defect density, productivity

3. **Size Oriented**(size of software in LoC)

   E.g.: Errors/KLoC, Cost/LoC

4. **Complexity Oriented** (LoC – function of Complexity)

- Fan-In, Fan-out
- Halstead's software science (entropy measures)
- Program length, volume, vocabulary

5. **Product Metrics**
   - Assessing the state of the project
   - Tracking potential risks
   - Uncovering problem areas
   - Adjusting workflow or tasks
   - Evaluating teams ability to control quality

6. **Project Metrics**

- Number of software developers
- Staffing pattern over the lifecycle of software
- Cost
- Schedule
- Productivity

7. **Process Metrics**

- Insights of software engineering tasks, work product or milestones?
- Long term process improvements

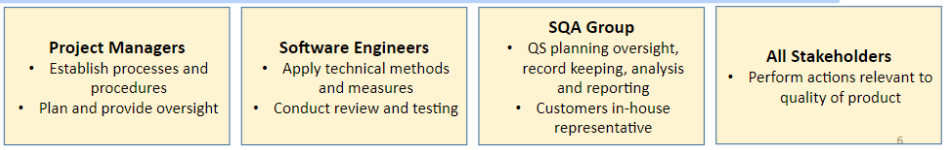
Aspect	Attribute	Measure	Metric
Definition	A characteristic or property of software.	A quantifiable value or set of values.	A quantitative measure or calculation based on data.
Nature	Descriptive and qualitative.	Numeric and quantitative.	Quantitative, often derived from multiple measures.
Example	Usability, reliability, maintainability.	Number of defects, response time, code size.	Defect density, cyclomatic complexity, code churn rate.
Purpose	Identifies a trait or feature of software.	Represents the result of a measurement.	Aggregates and interprets measures for evaluation.
Role in Quality	Forms the basis for defining quality goals.	Used to evaluate the quality of a specific attribute.	Enables comparison, trend analysis, and decision-making.
Scale	Qualitative scale (e.g., low, medium, high).	Quantitative scale (e.g., seconds, lines of code).	Quantitative scale with defined benchmarks and thresholds.
Context	Often subjective and context-dependent.	Objective and can be universally measured.	Combines multiple measures to provide a holistic view.
Examples	Maintainability, reliability, security.	Defect count, response time, code coverage.	Defect density (defects per KLOC), cyclomatic complexity.

Aspect	Attribute	Measure	Metric
Application	Used to define quality requirements.	Used to assess and analyze specific aspects of software.	Applied to evaluate and improve overall software quality.

Software Quality Assurance

methods to monitor software engineering process to ensure quality

- Involves planning (setting up of goals, commitments, activities, measurements and verifications)
- Encompasses
  - Entire software development processes and activities
  - Planning oversight, record keeping, analysis and reporting
  - Auditing designated software work to verify compliance
  - Ensuring deviations from documented procedure are recorded and noncompliance is reported



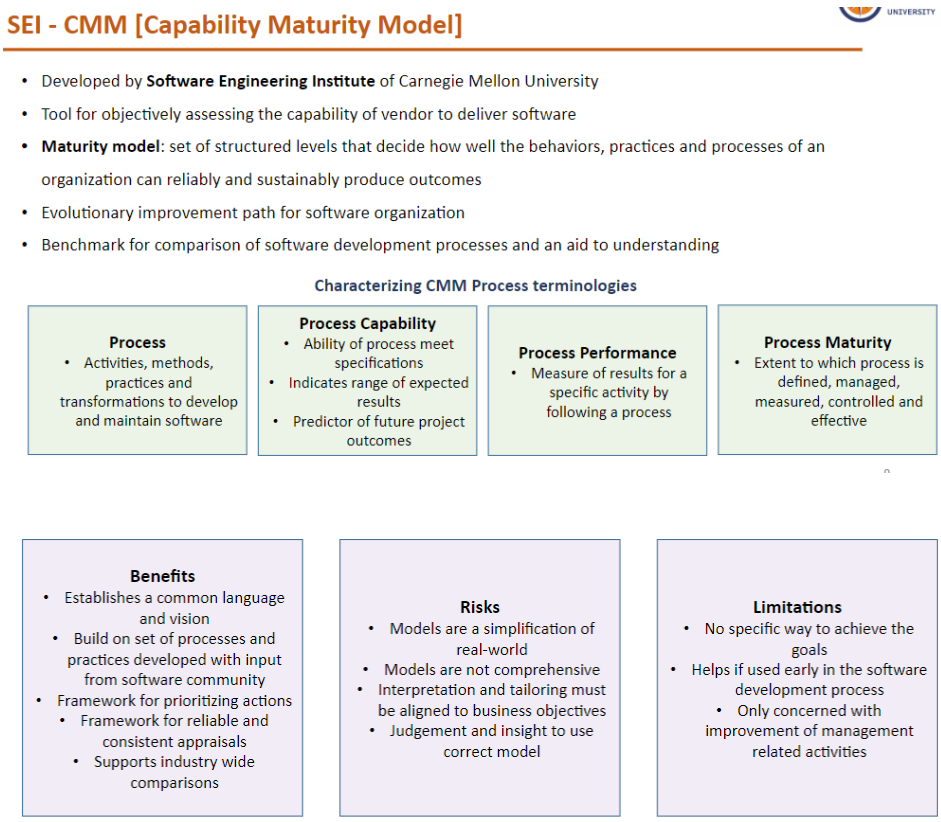
  - Project managers: establish processes and methods for the product and
    provide oversight for execution
  - Stake holders: performs actions relevant to quality
  - Software engineers: apply technical methods and measures; conduct
    formal technical review; perform software testing

- SQA group: QA planning oversight, record keeping, analysis and reporting;
  customers in-house rep

The SQA Plan

Some basic items

- Purpose of plan and scope
- Management
  - Organization structure, SQA tasks and placement
  - Roles and Responsibilities
- Documentation
  - Project, technical and user documents, models
- Standards, practices and conventions
- Reviews and audits
- Test plan and procedure
- Problem reporting and corrective measures
- Tools
- Code Control
- Media Control
- Supplier Control
- Records collection, Maintenance and Retention
- Training
- Risk Management



SEI - CMM

1. **Maturity model:** set of structured levels that describe how well the behaviour, practices and processed of an organisation can reliably and sustainable produce
2. **Evolutionary improvement path** for organisations from an ad hoc, immature process
3. **Benchmark** for comparison of software development processes

- Characterisation
  - **Process:** activities, methods, practices and transformations
  - **Process capability:** ability of process to meet specifications

        - **Process performance:** measure of actual results for the activity
        - **Process maturity**: extent to which process is defined, managed, measured controlled is effective
    - Benefits
        - Establish common language and vision
        - Build on set of processes and practices developed with input from the community
        - Provide framework for prioritisation
        - Provide framework for performing reliable and consistent appraisals
        - Industry wide comparisons
    - Risks
        - Models are simplification of real world
        - Not comprehensive
        - Interpretation and tailoring must be aligned
        - Judgement is necessary
    - Limitations
        - Doesn’t specify a way to achieve goals
        - Only helps if put in place early in SDLC