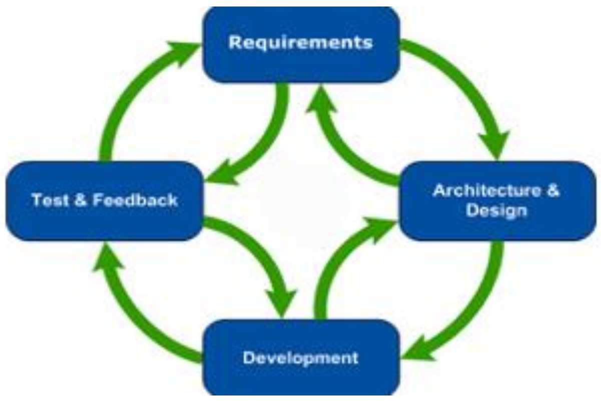
DAY 29

REPORT – ANALYSIS OF SDLC MODELS FOR EMBEDDED SYSTEMS

Software Development Life Cycle (SDLC) model is defined as a frame work used to describe the activities to be performed at each stage of a software development project. Different SDLC models include Waterfall model, V model, Iterative model, Incremental model, Spiral model, Agile Methods.

AGILE METHOD

Agile method for software development is an evolutionary approach where requirements and solutions evolve through collaboration between self organizing, cross-functional teams within an effective governance framework to provide solutions in cost effective and timely manner which meets the changing needs of stakeholder.



The major agile development methods include

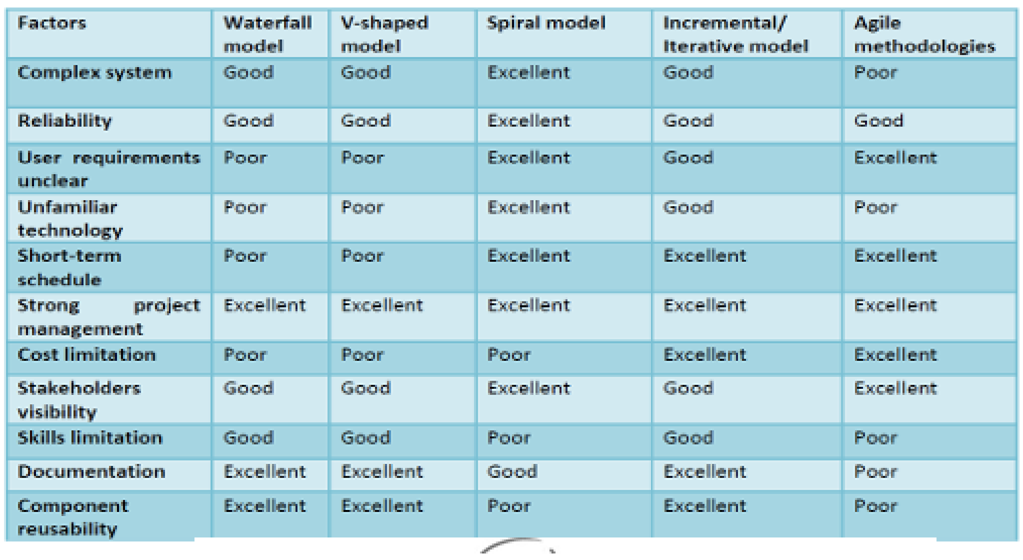
- Extreme Programming (XP): evolved from problems caused by the long development cycles of traditional development models.
  The life cycle of XP consists for five phases namely
  1. Exploration,
  2. Planning, and Iterations to Release,
  3. Productionizing,
  4. Maintenance
  5. Death Phase
- Scrum : is an empirical approach applying the ideas of industrial process control theory to systems development by reintroducing the ideas of flexibility, adaptability and productivity. Scrum involves environmental and technical variables (such as requirements, time frame, resources, and technology) that are likely to change during the process
  Phases of scrum include
  1. Pre-game,
  2. Development
  3. Post game phase.
- Dynamic Systems Development Method (DSDM): The idea behind DSDM was to fix the time and resources and then adjust the amount of functionality.
- Adaptive Software Development (ASD): provides a framework with enough guidance to prevent projects from falling into chaos which could suppress emergence and creativity of project .
  Its phases include

1. Speculate (planning),
2. Collaborate (highlights importance of team work) and
3. Learn (stresses on need to acknowledge and react to mistakes).

- Crystal and Feature Driven Development (FDD): In the crystal method, each member is marked with a color indicating heaviness (darker color) of methodology. The choice of color for a project is based on its size and criticality i.e: larger projects are likely to ask for more co-ordination and heavier methodologies than smaller ones. Crystal clear is designed for small projects (relatively restricted communication structure), crystal orange for medium sized projects and crystal red is for larger projects.

SDLC SELECTION FACTOR

Table 1. SDLC Selection Factor for Embedded Systems



Factors	Waterfall model	V-shaped model	Spiral model	Incremental/ Iterative model	Agile methodologies
Complex system	Good	Good	Excellent	Good	Poor
Reliability	Good	Good	Excellent	Good	Good
User requirements unclear	Poor	Poor	Excellent	Good	Excellent
Unfamiliar technology	Poor	Poor	Excellent	Good	Poor
Short-term schedule	Poor	Poor	Excellent	Excellent	Excellent
Strong project management	Excellent	Excellent	Excellent	Excellent	Excellent
Cost limitation	Poor	Poor	Poor	Excellent	Excellent
Stakeholders visibility	Good	Good	Excellent	Good	Excellent
Skills limitation	Good	Good	Poor	Good	Poor
Documentation	Excellent	Excellent	Good	Excellent	Poor
Component reusability	Excellent	Excellent	Poor	Excellent	Poor

APPLICATION OF AGILE METHODS

The different aspects of Agile Methods (AM) were analyzed to fit for embedded system development as follows:

1. **Rapid Cycles and Feature Granularity**: AM's rapid cycles and deliverables require granular feature sets. However, decomposing embedded systems into small, parallel tasks is challenging due to their specific functionality. Frequent customer releases for such systems may not be feasible.
2. **Documentation vs. Coding Focus**: AM prioritizes coding over planning and documentation, but embedded systems require extensive documentation for optimized code and long-term maintenance, as original developers may not always be available.
3. **Refactoring**: Continuous refactoring in AM is advantageous for optimizing embedded systems, provided the initial architecture is robust. Iterative improvements can resolve performance issues, like memory and cycle inefficiencies.
4. **Team Communication**: Extensive communication among engineering teams is beneficial in embedded systems to minimize hardware shortages and ensure proper documentation for future maintenance.
5. **Customer Interaction**: AM's emphasis on developer-customer interaction is less effective in embedded systems, as software is not directly visible to customers. Using a customer proxy can bridge the gap between customer expectations and developer deliverables.

6. **Testing**: AM’s test-driven development and regression testing are valuable for early defect reduction. Debugging at the unit level is easier, but system-level debugging poses challenges due to memory, timing, and multitasking constraints.
7. **Continuous Planning and Adjustments**: AM’s iterative planning aids embedded system development by managing feature sets, resources, delivery timelines, and identifying areas for performance optimization.

LEAN AGILE APPROACH

Lean process is a management philosophy developed by Toyota production system. The principle of Lean is to minimize waste in embedded system development.
Combining lean with Agile generate better embedded software development with much better position to recalibrate and reprioritize when conditions changes during development phase without compromising on quality or time.
The core tenants of the Lean Agile approach for embedded system development are:

1. Eliminate Waste
2. Empower the team
3. Amplify Learning
4. Build Integrity
5. Deliver as fast as possible

the requirement captured through the user stories. The Lean Agile method allows to focus on particular features of embedded system to provide the better business value for the system development with priority.

Benefits:

1.fast turnaround

2.direct feedback

3.prioritized projects

4.high value delivery

Agile methodologies are adaptable to changing requirements, enables face to face communication continuously with customers focus on lesser documentation and working software is delivered in rapid cycles. But these features are advantageous during an embedded system development since it add value to the system development and business.

REPORT – SOFTWARE DEVELOPMENT LIFE CYCLE EARLY PHASES AND QUALITY METRICS

Software Development Life Cycle (SDLC) is the time required for the activities such as de ning, developing, testing, delivering, operating and maintaining the software or the system. Early defects detection in early phases of the software process is one of the sources in the way to a successful project. However, the classi cation of early phases can vary referring to the methodologies the company uses. Hence, methods for assessing and evaluating the quality of the software process depend on the company preferences. Therefore the set of measurements that should be tracked during the evaluation process differ as well.
In general, almost all of the studies elaborated software life early phases into Requirements Management and Design phase, and sometimes Code. The publications weight representing the respective early phases of software life is described in the following list:

- Requirements phase
- Requirements and Design phases
- Design
- Design and Code phases
- Code and Testing phases
  the software development life cycle early phases includes User Needs Analysis, De nition of the Solution Space, External Behavior De nition and Preliminary Design. Where the first three stages union is the Requirements stage. It consists of all the activities up to the decomposition of the software architectural components.
- Initial planning phase- the technical and economic basis for the project should be established
- Analysis- the functional performance requirements for the software con guration items are de ned. This phases result is the successful completion of Preliminary Design Review (PDR)
- Design- this phase include the allocation of requirements to software components and culminates with complete Critical Design Review (CDR

The paper discusses various models and approaches used to assess and evaluate software quality in the early phases of development, with a focus on requirements documentation. Key approaches include:

1. **Aversano et al. (2018)**: This study analyzed the documentation quality of ERP open-source systems, focusing on **Structure Quality** and **Content Quality**. Structure Quality was assessed using metrics that required Natural Language Processing (NLP), Information Extraction, and Information Retrieval techniques. Content Quality involved evaluating a set of quality attributes for documentation.

2. **Quantitative Metrics Integration**: The study suggested integrating quantitative measures and metrics with requirements specification languages and automated tools like Ada, ISDOS, PSL, and others.

3. **CAME Tools**: These tools (Computer Assisted Software Measurement and Evaluation) help model and determine metrics for software development components, covering process, product, and resource aspects. CAME tools also support model-based analysis, metrics application, and statistical evaluation.
4. **ESQUT Tool**: Used in software development departments, this tool evaluates software quality by measuring C and COBOL source code and design documents using tree-structured charts.
5. **SORTT (Service Oriented Requirements Traceability Tool)**: A prototype tool developed to automate the extraction, indexing, querying, filtering, and rendering of trace links, thus improving the traceability of requirements.
6. **Source Monitor and CCCC Tool**: These tools compare programs written with design patterns versus those written without, helping assess the impact of design patterns on software quality.
7. **Clustering Analysis Techniques**: Research has applied clustering techniques, such as fuzzy c-means, k-means, and Gaussian mixture models, to predict software metrics quality.

The software development life cycle (SDLC) consists of several phases, each involving specific activities that help guide the project from inception to completion. The activities performed during the early phases of the SDLC, as described in various studies, are summarized as follows:

1. User Needs Analysis (Early Phase)

- **Activities**:
  - Understanding the user's needs and expectations.
  - Identifying the problem or opportunity that the software aims to address.

2. Definition of the Solution Space (Early Phase)

- **Activities**:
  - Defining the boundaries of the solution and determining the possible approaches to solving the identified problem.

3. System Requirements

- **Activities**:
  - **Feasibility Study**: Assessing whether the software can be realistically developed within the given constraints.
  - **Requirements Elicitation**: Gathering requirements from stakeholders.
  - **Requirements Analysis**: Analyzing the gathered requirements to understand their impact and feasibility.

  - **Requirements Validation**: Ensuring the requirements meet user needs and project objectives.
  - **Requirements Documentation**: Creating clear, comprehensive documentation of the software requirements.

4. System Design

- **Activities**:
  - **Examining the Requirements Document**: Reviewing the documented requirements to understand the design needs.
  - **Choosing Architectural Design Method**: Selecting the architectural model (e.g., monolithic, microservices, etc.) to guide the system design.
  - **Choosing the Programming Language**: Deciding on the programming languages and tools to be used.
  - **Verifying Design**: Ensuring the design aligns with the requirements and constraints.
  - **Specifying Design Details**: Defining the system architecture, components, interfaces, and data flow.
  - **Documenting Design**: Creating detailed design documentation for the development team.

During the software development life cycle (SDLC), various metrics are collected to assess the quality, complexity, reliability, and maintainability of the software at different stages. These metrics are important for managing uncertainties and evaluating the software's progress and potential defects. Some of the key metrics collected during each phase are

1. Requirements Management Phase

- **Requirement Defect Density**: Measures the number of defects in the requirements.
- **Requirement Specification Change Request**: Tracks the number of changes requested in the requirement specifications.
- **Requirement Inspection and Walkthrough**: Evaluates the effectiveness of the inspection and walkthrough processes in identifying defects early.

Key factors of interest in this phase include:

- Functions performed
- Response time
- User interface
- Reliability

Customer interests during this phase include:

- Functions performed
- Development time
- Cost
- Maintainability
- Modifiability
- Reliability

2. Design Phase

- **Cyclomatic Complexity**: Measures the complexity of the software design, indicating the number of linearly independent paths through the program.
- **Design Review Effectiveness**: Evaluates the effectiveness of the design review process in identifying issues.

Other design metrics and factors include:

- **Module Complexity**: Assesses the complexity of individual software modules.
- **Module Maintainability**: Evaluates the ease of maintaining the software modules.
- **Module Functionality**: Measures how well the software modules perform their intended functions.

3. Early Phases (Design and Early Coding)

- **Complexity, Coupling, and Cohesion (CCC) Metrics**: These metrics help in defect detection and predict potential issues in the early stages.
- **Object-Oriented Metrics**: Studies such as Victor R. Basili’s work on Chidamber and Kemerer OO metrics provide insights into predicting class fault-proneness during high- and low-level design phases.

4. Reliability and Other Factors

- **Reliability Metrics**: The reliability of the system can be influenced by external factors such as operational profile, project maturity, and domain.
- Some studies argue that predicting reliability is difficult due to computational complexity, but key reliability-related metrics include:
    - Fault probability
    - Defect prediction based on metrics like Cyclomatic Complexity and Design Review Effectiveness

5. Fuzzy Set Theory in Metrics

- Fuzzy set theory is used to assess uncertainty, vagueness, and imprecision in software metrics, particularly in early stages such as requirements and design phases.

  Commonly Mentioned Metrics:

- **McCabe Metrics**: Related to Cyclomatic Complexity and used to measure code complexity.
- **Halstead Metrics**: Focus on software complexity based on operators and operands.
- **Chidamber and Kemerer (CK) Metrics**: Used to assess object-oriented software metrics, including class cohesion and coupling.

  In conclusion, the SDLC collects a wide variety of metrics, each helping to evaluate different aspects of the software's development and quality. These include metrics for complexity, maintainability, functionality, reliability, and defects, among others, and can be used to predict and improve software quality across different phases.