Software Engineering – Module 1 Notes

1. Introduction to Software Engineering

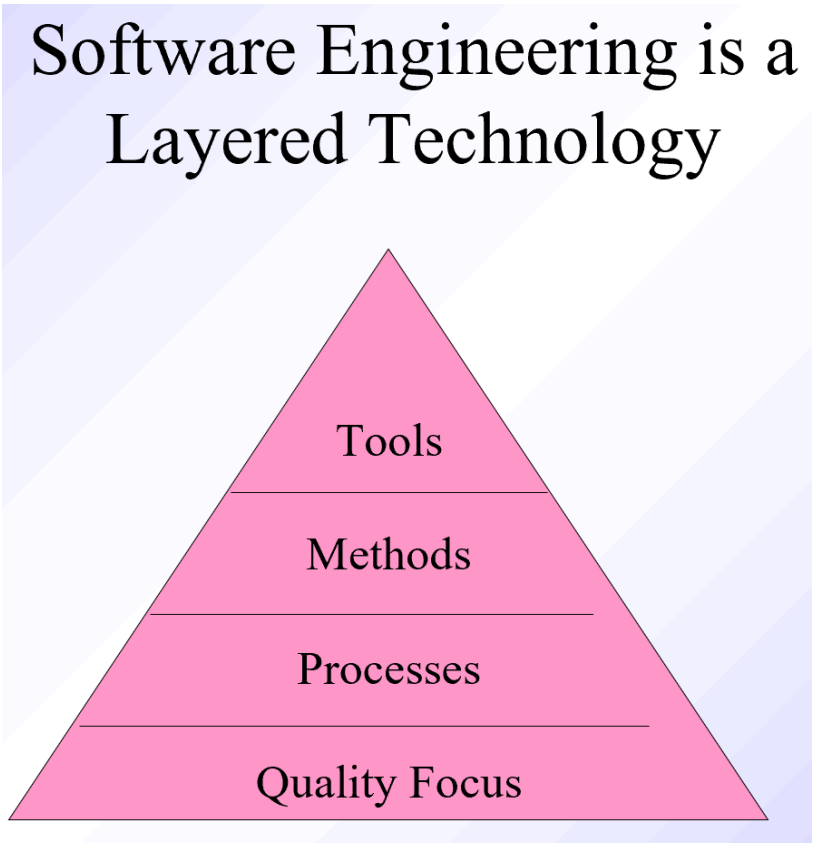
- **Software**: More than just program(executable code) code. Includes:
    - Executable programming code
    - Associated libraries
    - Documentation
- **Software Product**: Software built for a specific requirement.
- **Engineering**: Application of scientific principles and methods to develop products.

Definition of Software Engineering

- **1969 Definition**: Establishment and use of sound engineering principles to obtain economically reliable software that works efficiently.
- Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product
- **IEEE Definition**: Application of a systematic, disciplined, quantifiable approach to development, operation, and maintenance of software.

2. Software Engineering as Layered Technology

- **Process**: Glue holding everything together.
    - Defines framework, milestones, quality measures.
    - Ensures rational and timely development.
- **Methods**: Technical “how-to” for building software.
    - Includes modeling, analysis, and design tasks.
- **Tools**: Automated/semi-automated support for process & methods.
    - Integrated to aid in system/software development.
- **Quality**: Foundation layer. Strong quality assurance ensures reliable outcomes.



---

3. Software Process Models

3.1 Prescriptive Process Models

- Promote orderly, well-defined workflow.
- Activities may be **linear, incremental, or evolutionary**.
- Define tasks, actions, milestones, and work products that are required to engineer a high quality software.

3.2 Generic Process Framework

1. Communication

- Interaction with **customers and stakeholders**.
- Goal: Gather all possible **requirements** of the system.
- Output: **Requirement Specification Document** that clearly defines what the system should do.

2. Planning

- Defines the **roadmap** for development.
- Includes:
    - Technical tasks
    - Resource allocation (time, people, tools)
    - Work products (deliverables)
    - Schedule of activities

3. Modeling (Analysis & Design)

- **Analysis**: Study requirement specifications to understand the problem.
- **Design**: Prepare system design describing:
    - Hardware and software requirements
    - Overall **system architecture**
- Helps visualize how the system will work before coding begins.

4. Construction (Coding & Testing)

- **Coding**: Convert design into actual programs.
- Developed in **small units**, then integrated.
- **Unit Testing**: Each module is tested for its functionality.
- **System Testing**: After integration, the full system is tested for errors and failures.

5. Deployment

- Deliver the developed software to the customer.
- Customer evaluates the system and provides **feedback**.
- Feedback may lead to refinements or updates.

4. Software Development Life Cycle (SDLC) Models

4.1 Waterfall Model

Overview

- **Oldest and simplest SDLC model**, widely recognized by management.
- Workflow follows a **linear, sequential approach** where each phase must be completed before moving to the next.

When It Is Used

- Requirements are **well understood, clear, and fixed**.
- **Product definition is stable**.
- Technology is established and not rapidly changing.
- No ambiguity in requirements.
- Suitable for **short projects** or enhancements of existing systems.

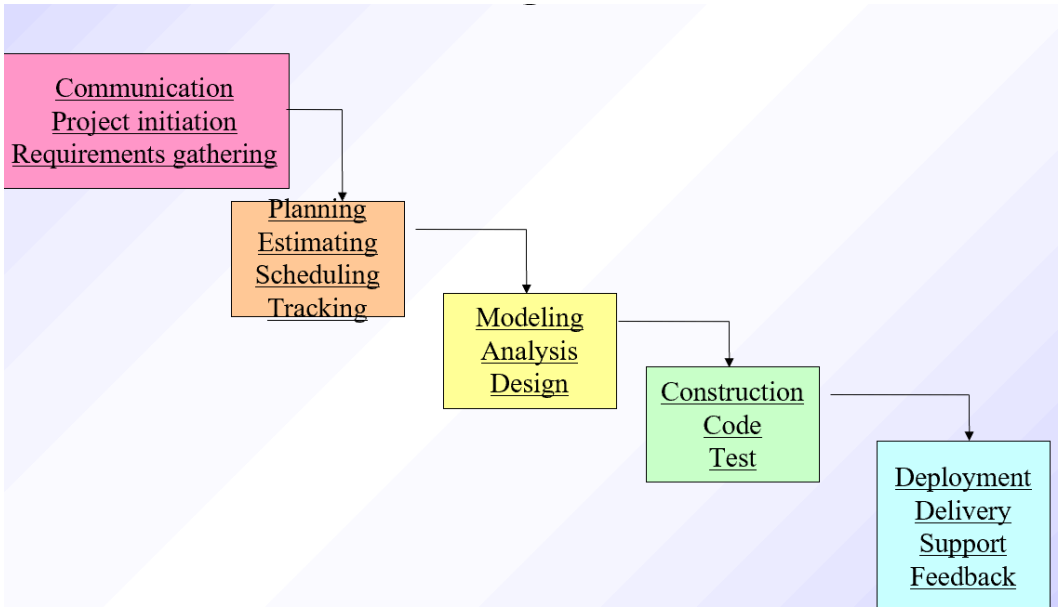
Advantages

- Easy to **understand and manage** due to its structured approach.
- Each phase has **defined deliverables and reviews**.
- Works well for **smaller projects** with stable requirements.
- Process and results are well **documented**.

Disadvantages

- **No working software** until late in the cycle.
- **No iteration support** → changes cause problems.

- Hard for customers to **specify all requirements upfront**.
- Requires **patience**, as results are visible only at the end.
- High **risk and uncertainty**, especially for larger or evolving projects.



4.2 Rapid Application Development (RAD)

Overview

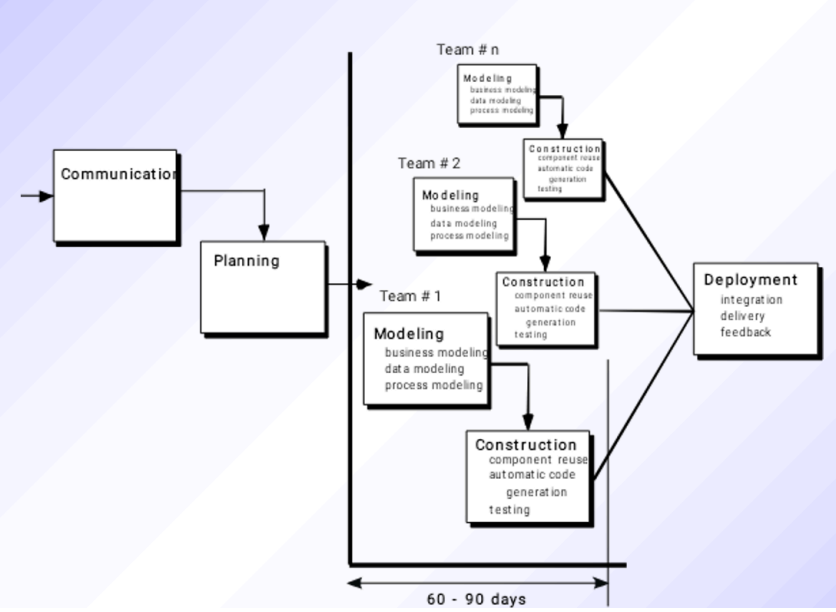
- A **fast-paced SDLC model** focused on quicker development and higher quality than traditional models.
- Functional modules are developed **in parallel as prototypes** and later integrated into the final product.
- Follows an **iterative and incremental approach**.
- Involves **small teams** of developers, domain experts, customer representatives, and IT resources.
- **Minimal preplanning**, making it easier to adapt to changes.
- Key to success: **Prototypes must be reusable**.

Advantages

- Accommodates **changing requirements** easily.
- Progress can be **measured at each iteration**.
- Faster development with powerful RAD tools.
- Encourages **customer involvement and feedback**.
- **Early integration** reduces later compatibility issues.
- High **component reusability**.
- Greatly **reduces overall development time**.

Disadvantages

- Depends heavily on **skilled and experienced team members**.
- Only suitable for **modular systems**.
- Not suitable for **low-budget projects** (high cost of tools and modeling).
- Requires **continuous user involvement**.
- Management becomes more **complex**.
- Best suited only for projects needing **shorter development times**.



4.3 Spiral Model

Overview

- Used when **requirements are unclear** and **risks are high**.
- Combines **prototyping** with aspects of the **Waterfall model**.
- Development proceeds in **spirals (iterations)**, each consisting of planning, risk analysis, engineering, and evaluation.

Working

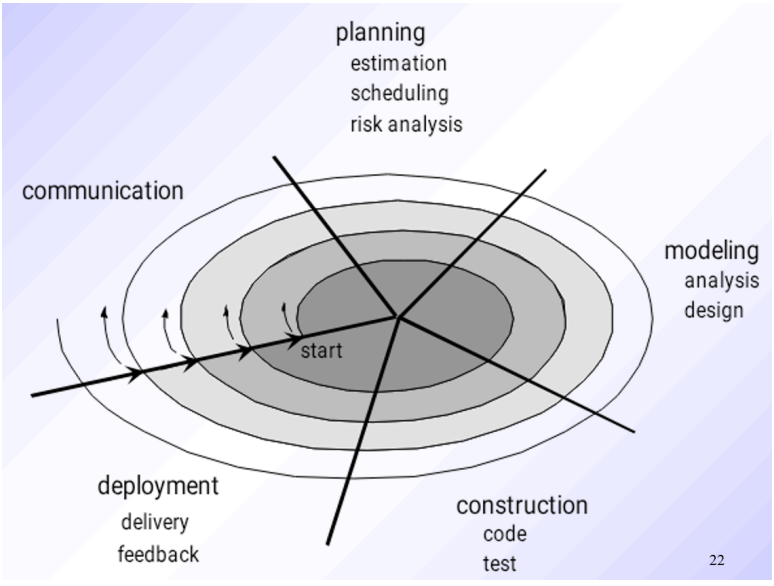
- **Inner Spirals**: Identify requirements and assess risks (may include prototyping).
- **Outer Spirals**: Follow a structured waterfall-like process with iterative growth.
- **Key Idea**: In each iteration, **tackle the highest-risk problems first**.
- Suitable for new product lines where feedback is needed in **phases**.
- Handles **significant changes** expected during the lifecycle.

Advantages

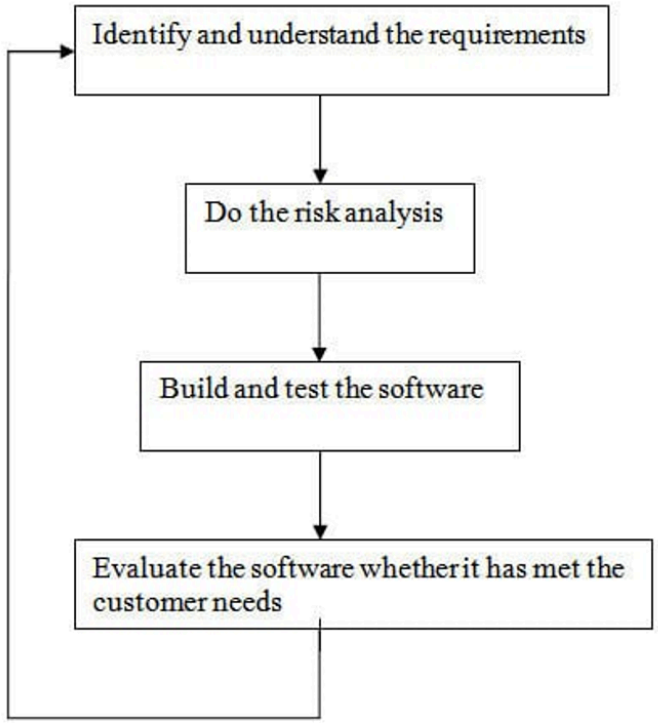
- Realistic: Reflects iterative nature of software development.
- Allows **extensive prototyping** for unclear requirements.
- Users see the system early.
- Requirements captured more accurately.
- Supports **incremental development**, focusing on risky parts earlier → better risk management.
- Flexible: Combines strengths of **Waterfall** and **Evolutionary** methods.

Disadvantages

- Requires **expertise in risk assessment**.
- **Complex to manage**.
- Project completion time may not be clear early on.
- Not suitable for **small or low-risk projects**; can be expensive.
- Spiral may continue **indefinitely** if not controlled.
- Large number of intermediate stages → **excessive documentation**.



To explain in simpler terms, the steps involved in the spiral model are:



4.4 Open Source Model

Overview

- Development model where software is made **freely available on the internet**.
- Typically begins with a **single individual's idea** (e.g., Linux, Firefox, Apache).
- Initial version is released openly, encouraging community involvement.

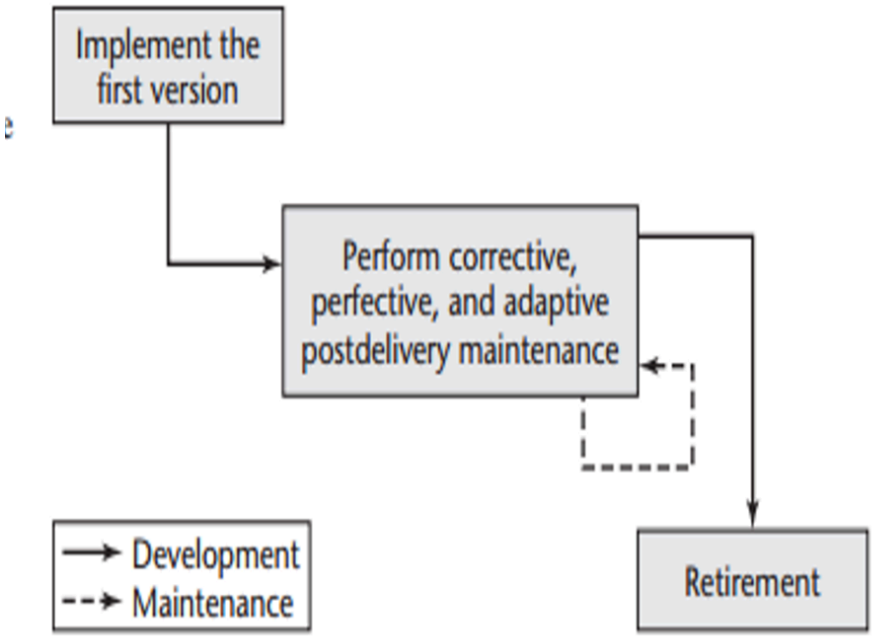
Phases

1. **Phase 1 – Initial Development**
    - Individual or small group creates the **first version** of the program.
    - Released to the public as **free software**.
2. **Phase 2 – Community Involvement**
    - Users become **co-developers**:

        - Report defects.
        - Suggest and implement fixes.
        - Propose and add new features.
        - Port software to different OS/hardware platforms.
    - Work is **voluntary** and usually done in free time.
    - This phase mainly involves **post-delivery maintenance and enhancement**.

Key Characteristics

- Driven by **community collaboration**.
- Development is **voluntary, unpaid**.
- Software evolves continuously through **feedback and contributions**.
- Long-term maintenance relies on user-developers.



4.5 Agile Process Models

- Combination of **iterative and incremental models**.
- Focus: **Adaptability** + **Customer Satisfaction** through **rapid delivery** of working software.
- Product is broken into **small incremental builds**, delivered in **short iterations** (1–3 weeks).
- Each iteration involves **cross-functional teams** working on:
    - Planning
    - Requirements analysis
    - Design
    - Coding
    - Unit testing
    - Acceptance testing
- At the end of every iteration, a **working product** is shown to customers/stakeholders.
- Builds are **incremental** → final build contains all features required.

Advantages

- **Realistic approach** to software development.
- Encourages **teamwork** and **cross-training**.
- Rapid development and demonstration of functionality.
- Suitable for both **fixed and changing requirements**.
- Provides **early, partial working solutions**.
- Works well in **steadily changing environments**.
- Easy to manage, with **little upfront planning**.
- Highest priority: **Customer satisfaction**.

Disadvantages

- Hard to **estimate effort** for large projects at the start.
- Less emphasis on **design and documentation**.
- Project may go **off track** if customer goals are unclear.
- Requires **experienced and skilled resources**.

SDLC Models – Comparison Table

Aspect	Waterfall	RAD	Spiral	Agile	Open Source
Approach	Linear & sequential	Prototype-driven, modular, fast	Iterative with risk analysis	Iterative + incremental (sprints)	Community-driven, evol release
When Used	Requirements are clear, stable, and fixed	Project is modular & time-sensitive	Requirements unclear, risks high	Requirements may change frequently	When community collaboration is possible
Customer Involvement	Low (only at beginning & end)	High (reviews prototypes throughout)	High (feedback after each spiral)	Very high (feedback after each iteration)	Very high (users act as co-developers)
Flexibility	Rigid, no iteration	Flexible (easy to change requirements)	Flexible (adapts to risks & changes)	Highly flexible (changes welcomed anytime)	Extremely flexible (anyone can contribute)
Risk Handling	Poor (risks found late)	Moderate (early prototyping helps)	Excellent (focus on risk analysis)	Good (iterative feedback reduces risk)	Moderate (depends on community response)
Time to Deliver	Long (final product only at end)	Very fast (parallel module development)	Long, due to many spirals	Fast (working builds every 1–3 weeks)	Continuous (software evolves over time)
Cost	Relatively low (simple tools)	High (requires advanced prototyping tools)	High (complex, risk experts needed)	Moderate (depends on team skill)	Low (free contribution, voluntary work)
Team Requirement	Average-skilled team okay	Highly skilled developers needed	Expert risk analysts needed	Cross-functional, experienced team	Global community of volunteers
Documentation	Heavy documentation	Less documentation, focus on prototypes	Heavy (many intermediate stages)	Light (focus on working software)	Light (community-driven, informal)
Output	Final product only at end	Fast prototypes, then integrated product	Prototypes + incremental builds	Working software after every sprint	Software evolves continuously
Best For	Small, stable projects (e.g., Payroll System)	Modular, short-timeline projects (e.g., Hospital Mgmt System)	Large, risky projects (e.g., Defense, Banking)	Dynamic, evolving projects (e.g., Food Delivery App)	Community projects (e.g., Linux, Firefox)

---

5. Agile Methods in Detail

Extreme Programming (XP):

- An **Agile method** that relies heavily on the **object-oriented approach**.
- Based on a **set of rules and practices** that occur within defined activities.
- Agile processes in XP are **incrementally adapted** to handle unpredictability.

1. XP Planning

- Starts with **User Stories** → short descriptions of required features.
- The team **estimates the cost** of each story.
- Stories are grouped into **deliverable increments**.
- A **delivery date** is committed.
- After the first increment, **project velocity** (rate of progress) is used to set timelines for future increments.

2. XP Design

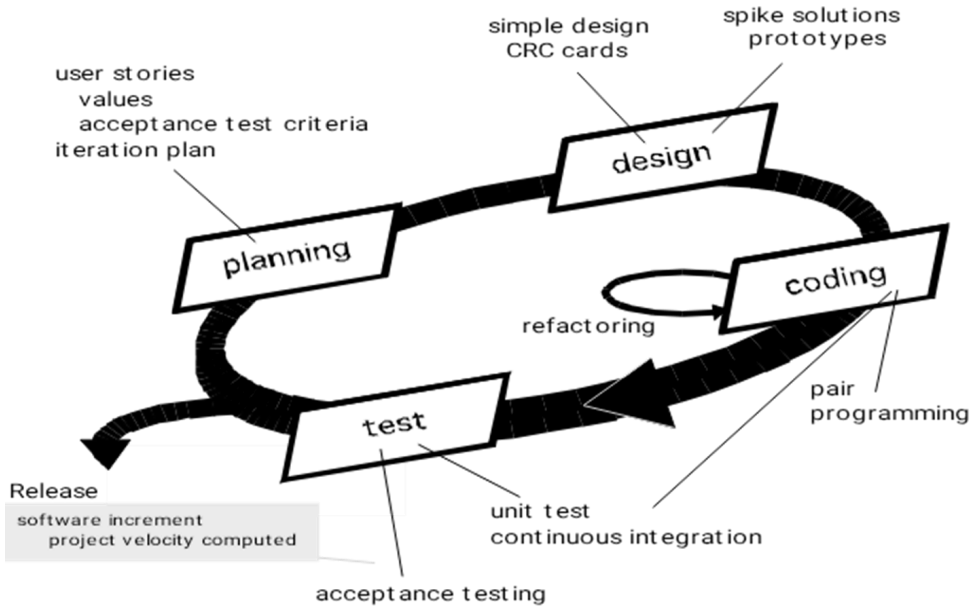
- Follows **KIS (Keep It Simple)** principle.
- Uses **CRC cards** (Class–Responsibility–Collaborator) for solving design issues.
- May create **"spike solutions"** → quick design prototypes for complex problems.
- Encourages **refactoring** (restructuring) → iterative improvement of program design.

3. XP Coding

- **Unit tests** are written **before coding** begins.
- Promotes **pair programming**: two developers work together on the same program.

4. XP Testing

- **All unit tests executed daily** to detect issues early.
- **Acceptance tests** are defined by the customer to validate visible functionality.



Adaptive Software Development (ASD):

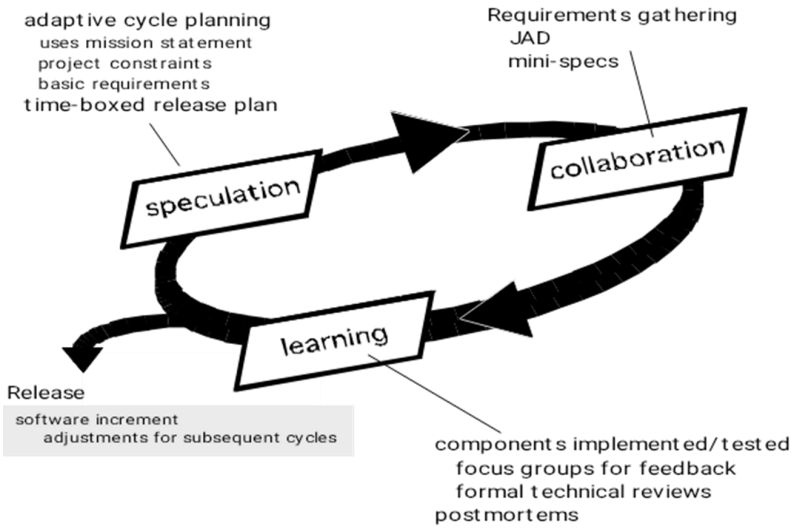
- A technique for building **complex software and systems**.
- Relies on **self-organizing teams** where independent agents collaborate to solve problems beyond individual capability.
- Focuses on **team collaboration, continuous learning, and adaptability**.
- Lifecycle emphasizes **results (application features)** rather than tasks.

Key Characteristics (Distinguishing Features)

1. **Mission Driven** – Each cycle must align with the overall project mission.
2. **Component Based** – Work focuses on delivering working software (results), not just tasks.
3. **Iterative** – Encourages redoing and refining development, rather than aiming for perfection the first time.
4. **Time-Boxed** – Fixed delivery timelines are set for each cycle.
5. **Change Tolerant** – Change is seen as a **competitive advantage**.
6. **Risk Driven** – High-risk components are developed early to reduce uncertainty.

Phases of ASD

1. **Speculation**
    - Project initiated with mission statement, requirements, and delivery dates.
    - Adaptive planning allows requirements to **evolve with change**.
2. **Collaboration**
    - Strong teamwork and communication emphasized.
    - Uses **Joint Application Development (JAD)** for requirement gathering.
3. **Learning**
    - Components are implemented and tested.
    - Feedback from **focus groups, reviews, and postmortems** helps improve future cycles.



DSDM(Dynamic Systems Development Method):

- Provides a framework for **building and maintaining systems under tight time constraints**.
- Uses **incremental prototyping** in a controlled environment.
- Based on **Pareto principle**: 80% of value can be delivered in 20% of the total time.
- Each increment delivers just enough functionality to progress to the next.

- Uses **time boxes** to fix time and resources, defining how much functionality is delivered.

Key Principles

- **Active user involvement** is essential.
- Teams must be **empowered to make decisions**.
- **Frequent delivery** of functional prototypes.
- Deliverables must meet **business fitness criteria**.
- **Iterative and incremental development** is mandatory.
- All changes during development must be **reversible**.
- **Requirements are baselined at a high level** → prevents uncontrolled changes.
- **Testing is integrated** throughout the lifecycle.
- Strong **collaboration between business and technical staff** is mandatory.

DSDM Lifecycle Phases

1. **Feasibility Study**
    - Identifies business requirements and constraints.
    - Assesses whether the project is suitable for DSDM.
2. **Business Study**
    - Defines **functional and information requirements** needed for business value.
3. **Functional Model Iteration**
    - Produces incremental prototypes to demonstrate functionality.
    - Allows gathering of **additional requirements**.
4. **Design and Build Iteration**
    - Revisits earlier prototypes.
    - Ensures solutions are engineered for **operational business value**.
5. **Implementation**

    - Delivers the latest increment (an “operationalized prototype”) into the live environment.

Scrum:

Overview

- An **Agile software development process model**.
- Focuses on **self-organization** of small teams working **intensively and interdependently**.
- Adaptable to both **technical and business challenges**.
- Produces **frequent increments** that can be inspected, tested, documented, and improved.

Key Concepts

- **Packets**: Development work is divided into small, manageable units.
- **Backlog**:
    - A **prioritized list** of project requirements.
    - Items can be **added or reprioritized** as the project evolves.
- **Sprints**:
    - Short, time-boxed development cycles.
    - Derived from the backlog.
    - Each sprint delivers a functional increment.
    - **No changes** allowed during a sprint → ensures stability.
- **Daily Scrum Meetings**:
    - Short (≈15 minutes).
    - Team reviews progress and discusses issues.
- **Demos**:
    - At the end of a sprint, the increment is delivered and shown to the customer.

- Demonstrates completed functionality (even if not all features are ready).

Principles & Benefits

- Promotes **team self-organization**.
- Handles uncertainty by adapting continuously.
- Encourages **incremental delivery**, so customers see progress early.
- Testing and documentation are **ongoing throughout development**.

Crystal:

Overview

- A **family of Agile methods** designed for projects of varying **sizes** and **criticalities**.
- Known as one of the most **lightweight and adaptable** methodologies.
- Tailored by factors like **team size, system criticality, and project priorities**.

Key Characteristics

- Focus on **teamwork, communication, and simplicity**.
- Encourages **reflection** to continuously adjust and improve the process.
- **Primary Goal**: Deliver useful, working software.
- **Secondary Goal**: Prepare for future development ("set up for the next game").

Feature Driven Development (FDD):

Overview

- Agile method with focus on **defining features**.
- A **feature** = client-valued function that can be built in **≤ 2 weeks**, expressed as: `<action> <result> <object>` (e.g., *"Calculate monthly interest"*).
- Emphasizes **collaboration** and **feature-based decomposition**.

Key Characteristics

- **Incremental and iterative** development.
- Strong focus on **software quality** through:
    - Design & code inspections
    - SQA audits
    - Metric collection
    - Use of design and analysis patterns
- Uses **verbal, graphical, and textual communication** for clarity.

FDD Process Steps

1. **Develop Overall Model** – Create a domain model with business classes.
2. **Build Features List** – Extract, categorize, and prioritize features (in 2-week chunks).
3. **Plan by Feature** – Assess effort, priority, technical dependencies, and schedule.
4. **Design by Feature** –
    - Select relevant classes.
    - Prepare design details.
    - Assign class ownership.
5. **Build by Feature** –
    - Class owners write code and perform unit testing.
    - Chief programmer integrates features.

Agile Modeling (AM):
Overview

- A **practice-based methodology** for effective **modeling and documentation** of software systems.
- Focus: Keep modeling **practical, lightweight, and purposeful** within Agile projects.

Key Principles

1. **Model with a Purpose**
    - Always create models with a **specific goal** in mind.
2. **Use Multiple Models**
    - Different notations/models can describe a system.
    - Only a **small subset** is essential for most projects.
3. **Travel Light**
    - Keep only models that provide **long-term value**.
    - Avoid unnecessary or excessive documentation.
4. **Content over Representation**
    - Focus on **information** delivered by the model, not on its appearance.
5. **Know the Models and Tools**
    - Understand the **strengths and weaknesses** of modeling techniques and tools used.
6. **Adapt Locally**
    - Tailor modeling practices to suit the **needs of the Agile team** and project context.

Step 2: Map Characteristics to Agile Models

Agile Model	Best Fit	Key Points
Scrum	Medium to large teams, evolving requirements	Iterative sprints, strong roles (PO, Scrum Master, Dev Team), good for both critical and non-critical systems
XP (Extreme Programming)	Small teams, highly dynamic requirements	Focus on coding practices (TDD, pair programming), very high collaboration
FDD (Feature-Driven Development)	Projects needing **clear, client-valued features**	Best for banking/financial apps where features can be defined clearly
DSDM (Dynamic Systems Development Method)	Time-critical, business-focused projects	Works well if deadlines are fixed and requirements are high-level initially
Crystal	Varies by team size and criticality	Lightweight, flexible, emphasizes communication and reflection
Agile Modeling (AM)	Any Agile project	Focused on **lightweight documentation and modeling** alongside other Agile methods
ASD (Adaptive Software Development)	High uncertainty, rapidly changing environments	Emphasizes learning, speculation, and collaboration

6. Software Process and Metrics

6.1 Process

Definition of a Process:

- A system of operations to produce something.
- A series of actions, changes, or functions aimed at achieving a specific result.
- Can also be seen as a sequence of steps performed for a given purpose.

Process Measurement:

- Measures the effectiveness of a software process indirectly.
- Based on metrics derived from the outcomes of the process.

Outcomes Measured:

- **Errors detected before release** – indicates process quality.
- **Defects reported by end-users** – shows delivered software reliability.
- **Work products delivered (productivity)** – evaluates output efficiency.
- **Human effort expended** – tracks resource utilization.
- **Calendar time expended** – measures time efficiency.
- **Schedule conformance** – assesses adherence to planned timelines.
- **Other measures** – any additional relevant performance indicators.

6.2 Software Process

Software Process:

- A set of activities, methods, practices, and transformations used to **develop and maintain software**.
- Includes producing associated products such as **project plans, design documents, code, test cases, and user manuals**.

Process Maturity:

- As an organization matures, its software process becomes **better defined** and **more consistently implemented**.
- **Software Process Maturity** is the degree to which a process is:
    1. **Explicitly defined** – clearly documented and understood.
    2. **Managed** – actively controlled and organized.
    3. **Measured** – performance and outcomes are quantified.
    4. **Controlled** – deviations are monitored and corrected.
    5. **Effective** – achieves desired results reliably.

Importance of Software Process Metrics:

- Help **measure and improve process maturity**.
- Provide insight into **process effectiveness and areas for improvement**.

6.3 Process Metrics

Time Metrics:

- Measures the duration of a process.
- Includes total time devoted, calendar time, and engineer's time spent.

Resource Metrics:

- Tracks resources required for a process.
- Examples: person-days, travel costs, or computing resources.

Count Metrics (Event Occurrences):

- Measures the number of times specific events occur.
- Examples: defects discovered during code inspection, requirement changes requested, lines of code modified due to changes.

Reuse Metrics:

- Measures the number of components produced and their degree of reusability.

Quality Metrics:

- Focuses on the **quality of work products and deliverables**.

---

7. Software Process Maturity & CMMI

7.1 Immature Organizations

- Software processes are generally **improvised** rather than planned.
- Even if a process is specified, it is **not rigorously followed or enforced**.
- The software organization tends to be **reactionary**, addressing problems as they arise.
- Managers focus only on **solving immediate (crisis) problems**.
- **Schedules and budgets** are often exceeded due to **unrealistic estimates**.
- When **hard deadlines** are imposed, **product functionality and quality** are often compromised.
- There is **no systematic basis** for judging process quality or solving product/process problems.
- Activities like **reviews and testing** are often **curtailed or eliminated** when projects fall behind schedule.

7.2 Capability Maturity Model Integration (CMMI)

- A proven framework to **improve product quality and development efficiency** for hardware and software.
- **Staged CMMI** uses **5 maturity levels** to describe organizational maturity.
- Each maturity level has **associated process areas and goals** (specific & generic).

- **Maturity Level:** Evolutionary plateau toward achieving a mature software process.
- **Capability Level:** Evolutionary plateau describing capability relative to a process area; consists of practices to improve processes.

- **Maturity Levels**:
  - **Initial Level (Level 1):**
    - Processes are **ad-hoc or chaotic**.
    - Few processes are defined; success depends on **individual effort**.
    - Projects often **exceed budget and schedule**.
    - Organizations tend to **over-commit**, abandon processes in crises, and cannot **repeat past successes**.
  - **Managed Level (Level 2):**
    - All **specific and generic goals** of level 2 process areas are achieved.
    - **Process discipline** ensures practices are retained during stress.
    - Projects are **performed and managed according to documented plans**.
    - **Requirements, processes, work products, and services are managed**, with visibility to management.
    - Work products and services **satisfy specified requirements, standards, and objectives**.
  - **Defined Level (Level 3):**
    - Software process for **management and engineering** is documented, standardized, and integrated.
    - All projects use an **approved, tailored version** of the organization's standard process.
    - **Difference from Level 2:**
      - Level 2: Standards/procedures may vary per project.

        - Level 3: Standards/procedures are **tailored from organization-wide standard processes**.
    - Processes are **more detailed and rigorous** than at level 2.
- **Quantitatively Managed (Level 4):**
    - Achieves **all goals of levels 2, 3, and 4**.
    - Selected **sub-processes** significantly contributing to performance are **controlled using statistical/quantitative techniques**.
    - **Quantitative objectives** for quality and performance guide process management.
    - **Process performance is measured and analyzed statistically**; special causes of variation are identified and corrected.
    - **Difference from Level 3:**
        - Level 4: Process performance is **quantitatively predictable**.
        - Level 3: Processes are **qualitatively predictable**.
- **Optimizing Level (Level 5):**
    - Achieves **all goals of levels 2–5**.
    - Focuses on **continuous improvement** through incremental and innovative improvements.
    - **Level 4:** Addresses **special causes** of process variation.
    - **Level 5:** Addresses **common causes** of process variation for ongoing optimization.
-

- **Capability Levels** (for each process area):

1. **Level 0 – Incomplete:**
    - Process area is **not formed** or does not achieve **defined goals and objectives** of CMMI.
2. **Level 1 – Performed:**
    - **All specific goals** of CMMI are satisfied.
    - Work tasks are **completed** for defined work.

3. **Level 2 – Managed:**
    - **Level 1 criteria satisfied**.
    - Work conforms to the **organization's defined policy**.
    - **Adequate resources** are available.
    - **Stakeholders are actively involved**.
    - Work tasks are **evaluated** and adhere to **CMMI standards**.
4. **Level 3 – Defined:**
    - **Level 2 criteria satisfied**.
    - Processes are **tailored from the organization's standard guidelines**.
5. **Level 4 – Quantitatively Managed:**
    - **Level 3 criteria satisfied**.
    - **Quantitative objectives** for quality and process performance are established and used to **manage processes**.
6. **Level 5 – Optimized:**
    - **Level 4 criteria satisfied**.
    - Process areas are **optimized** to meet **changing customer requirements**.
    - Focus on **continuous improvement** of **software efficiency and quality**.

---

8. Project Planning & Estimation

Project Planning

Overall Goal:

- Establish a **practical strategy** for controlling, tracking, and monitoring a complex technical project.
- Ensure the **end result is delivered on time and with quality**.

Objectives of Software Project Planning:

1. Provide a **framework** for managers to make reasonable estimates of **resources, cost, and schedule**.

2. Estimates should define **best-case and worst-case scenarios** to **bound project outcomes**.

3. **Plans must be adapted and updated** as the project proceeds.

Key Considerations in Project Planning:

- **Project scope** must be clearly understood.
- **Elaboration (decomposition)** of tasks is necessary.
- Use **historical metrics** wherever possible.
- Apply **at least two different estimation techniques**.
- **Uncertainty** is inherent and must be accounted for.

Metrics

1. Function Point (FP) Metric

Purpose:

- Measures the **functionality delivered by a system** from the **user's perspective**.
- Focuses on **what the user requests and receives**, rather than lines of code.

Uses of FP Metric:

1. **Estimate cost or effort** required to **design, code, and test** the software.
2. **Predict the number of errors** likely to occur during testing.
3. **Forecast the number of components** and/or **projected source lines** in the implemented system.

Information Domain Values for FP Measurement

1. **Number of External Inputs (EIs):**
    - Processes **data or control information** coming **from outside the application**.
    - **Inputs** are different from **inquiries**, which are counted separately.
2. **Number of External Outputs (EOs):**
    - Generates **data or control information** sent **outside the application**.
    - Derived data within the application **provides information to the user**.
    - Examples: **reports, screens, error messages**.
3. **Number of External Inquiries (EQs):**
    - A **transaction function** with **input and output components** that result in **data retrieval**.
    - Defined as an **online input** that produces an **immediate software response** in the form of an **online output**.
4. **Number of Internal Logical Files (ILFs):**
    - User-identifiable group of **logically related data or control information** that resides **entirely within the application**.
    - Maintained via **external inputs (EI)**.
5. **Number of External Interface Files (EIFs):**
    - Data resides **outside the application** and is **maintained by another application**.
    - Serves as an **internal logical file** for the other application.

Function Point Calculation:

- **Total FP count** = Sum of all FP entries (EIs + EOs + EQs + ILFs + EIFs).
- Each FP entry is evaluated using **Value Adjustment Factor (VAF)** on a scale:
    - **0** = Not important/applicable
    - **5** = Absolutely essential

$$FP = \text{count total} \times [0.65 + 0.01 \times \Sigma (F_i)]$$

Information Domain Value	Count		Weighting factor: Simple	Average	Complex		
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
Count total	→						

The F_i (i = 1 to 14) are *value adjustment factors* (VAF) based on responses to the following questions [Lon02]:

1. Does the system require reliable backup and recovery?
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require online data entry?
7. Does the online data entry require the input transaction to be built over multiple screens or operations?
8. Are the ILFs updated online?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

2. Lines of Code (LOC) Metric

Definition:

- Measures the **size of a computer program** by counting the number of **lines in the source code**.
- **Comments and header files** are **ignored**.

Purpose:

- Predict the **effort required** to develop a program.
- Estimate **programming productivity**.
- Assess **maintainability** of the software product.

Types of LOC:

1. **Physical LOC:** Counts **all lines of text** in the program.

2. **Logical LOC:** Counts the **number of executable statements**.

Structure of Estimation Models

- **Core Concept:**
    - Estimation models are mathematical formulas used to predict effort, cost, and schedule of a software project.
    - Based on historical data and key project variables.
- **Generic Model Form:**
    - **E = A + B * (ev)^C**
        - **E:** Effort in person-months
        - **ev (Estimation Variable):** Project size (LOC or FP)
        - **A, B, C:** Empirically derived constants (organization/industry-specific)
- **Purpose:**
    - Input estimated size → predict effort → convert to cost and schedule

---

- **COCOMO II Overview:**
    1. Hierarchical set of estimation models, used at different project stages
- **Three Models:**
    1. **Application Composition Model:** Early stages, prototyping, uses Object Points
    2. **Early Design Stage Model:** Requirements stable, uses Function Points (convertible to LOC)
    3. **Post-Architecture Model:** Main development, uses LOC
- **Sizing Options:** Object Points, Function Points, or Lines of Code

---

- **Object Points:**
    - Measure of software size for GUI/database-based applications

- Computation: Count number of Screens, Reports, Components
- Complexity Weighting: Each item classified as Simple, Medium, or Difficult → higher complexity = higher effort

---

- **New Object Point (NOP):**
  - Adjusts raw Object Point count for reuse
  - **Formula:** NOP = Object Points × [(100 - %reuse)/100]
  - Example: 1000 Object Points, 30% reused → NOP = 700

- **Effort Estimation:**
  - **Effort = NOP / PROD**
  - **Productivity Rate (PROD):**
    - Based on historical data, developer experience, and tool maturity
    - Higher PROD → lower estimated effort

- **Software Equation:**
  - **Formula:** E = [LOC * B^(0.333) / P]^3 * (1/t^4)
  - **Variables:**
    - E: Effort (person-months)
    - LOC: Estimated Lines of Code
    - B: Special skills factor (0.16–0.39)
    - P: Productivity parameter (2,000–28,000)
    - t: Project duration (months)
  - **Key Insight:** Effort inversely proportional to t^4 → compressing schedule drastically increases effort

---

- **Simplified Equations:**
    - Minimum Development Time: **t_min = 8.14 * LOC / (P)^0.43**
    - Estimated Effort: **E = 180 * B * (LOC / P)^1.333**

- **Reasons Projects Are Late:**
    - Unrealistic Deadlines
    - Changing Requirements (Scope Creep)
    - Underestimation of effort/resources
    - Unforeseen Risks
    - Technical Difficulties
    - Human Difficulties (team issues, sickness, turnover)
    - Miscommunication
    - Managerial Failure (not recognizing delays or taking corrective action)

- **Scheduling:**
    - Distribute estimated effort across project duration, allocating tasks to resources
    - Actionable plan example: Task A → John → Weeks 1–2; Task B → Sarah → Weeks 2–4
- **Scheduling Principles:**
    - Compartmentalization: Break project into manageable tasks
    - Interdependency: Define task relationships
    - Effort Validation: Ensure resource availability
    - Defined Responsibilities: Assign tasks to specific people/teams
    - Defined Outcomes: Each task must have a tangible deliverable

- Defined Milestones: Review points for quality and progress

- **Effort Allocation (40-20-40 Rule):**
  - 40%: Front-end (Requirements & Design)
  - 20%: Coding
  - 40%: Back-end (Testing & Debugging)
  - Note: Less applicable in iterative models

- **Task Network / Task Set Refinement:**
  - **Task Network:** Graphical representation of task sequences and dependencies
  - **Task Set Refinement:** Breaking large tasks into smaller sub-tasks
  - Example: Concept Scoping → 1.1.1, 1.1.2, 1.1.3 → further refinement 1.1.2.1, 1.1.2.2
  - FTR (Formal Technical Review) included as scheduled milestones

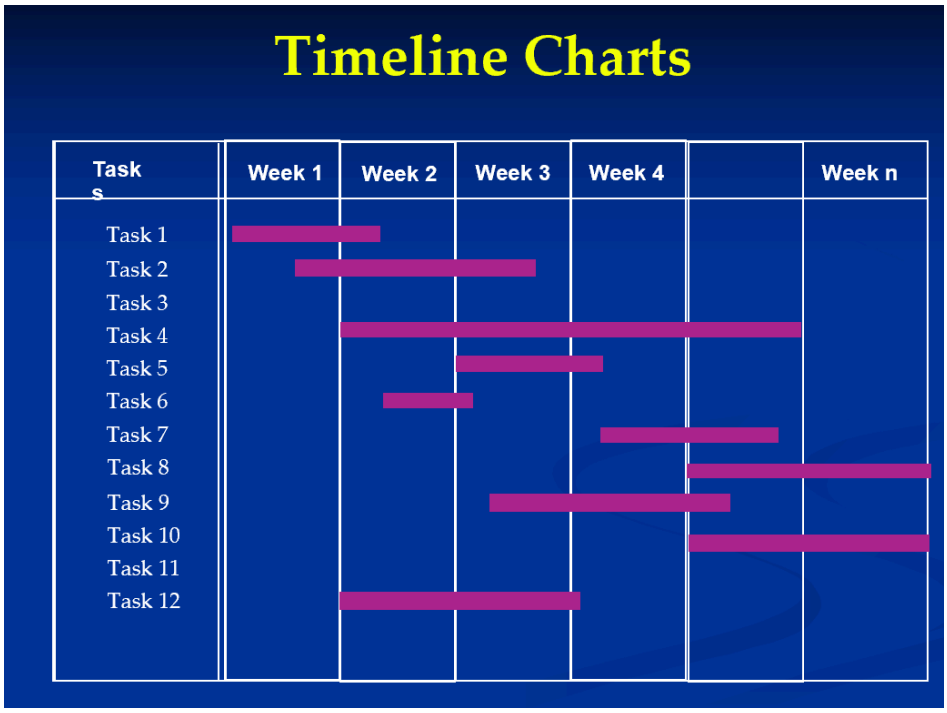
---

- **Scheduling**

Objective of Project Scheduling Tool:

- Define work tasks
- Establish task dependencies
- Assign human resources
- Develop graphs and charts for:
  - Tracking progress
  - Controlling the software project

- **PERT(Program Evaluation & Review Technique) & CPM(Critical Path Method):**
    - Define tasks, dependencies, resources → create charts for tracking and control
    - **Outputs:**
        - Critical Path: Longest dependent task chain → shortest project duration
        - Time Estimates: Optimistic, most likely, pessimistic
        - Boundary Window (Float/Slack): Time non-critical tasks can be delayed without affecting finish date
- **Timeline Charts (Gantt Charts):**
    - Bar chart representation of schedule
    - Vertical: Tasks | Horizontal: Time
    - Horizontal bars: Start date, duration, end date
    - Provides clear visual representation of schedule



- **Schedule Tracking Methods:**
    - Status Meetings
    - Review Results
    - Milestone Analysis
    - Start-Date Comparison
    - Informal Meetings
    - Earned Value Analysis (EVA)

- **Earned Value Analysis (EVA):**
    - is a measure of progress
    - enables us to assess the “percent of completeness” of a project using quantitative analysis rather than rely on a gut feeling
    - “provides accurate and reliable readings of performance from as early as 15 percent into the project.”

      EVA

    - Integrates scope, schedule, cost to assess progress
    - **Key Terms:**
        - BCWS: Budgeted Cost of Work Scheduled

$$\text{BAC} = \sum(\text{BCWS}k) \text{ for all tasks } k$$

        - BAC: Budget at Completion
        - BCWP: Budgeted Cost of Work Performed (Earned Value)
        - ACWP: Actual Cost of Work Performed
    - Value is "earned" as tasks are completed → compare planned vs actual vs cost

$$\text{Percent scheduled for completion} = \frac{\text{BCWS}}{\text{BAC}}$$

    -

- $$\text{Percent complete} = \frac{\text{BCWP}}{\text{BAC}}$$
- $$\text{Cost performance index, CPI} = \frac{\text{BCWP}}{\text{ACWP}}$$
- $$\text{Cost variance, CV} = \text{BCWP} - \text{ACWP}$$

- **EVA Formulas:**
  - **Schedule Variance (SV):** SV = BCWP - BCWS
    - SV < 0 → Behind schedule | SV > 0 → Ahead
  - **Schedule Performance Index (SPI):** SPI = BCWP / BCWS
    - SPI < 1 → Behind schedule | SPI > 1 → Ahead
  - **Cost Variance (CV):** CV = BCWP - ACWP
    - CV < 0 → Over budget | CV > 0 → Under budget
  - **Cost Performance Index (CPI):** CPI = BCWP / ACWP
    - CPI < 1 → Over budget | CPI > 1 → Under budget