

Software Management

1 Chapter 12: Analysis Modeling	2
1.1 Overview At a technical level	2
1.2 The Structure of the Analysis Model	2
1.3 Data Modeling	2
1.4 Functional Modeling and Information Flow	2
1.5 Real-Time Extensions	2
1.6 Behavioral Modeling	3
1.7 The Data Dictionary	3
2 Chapter 21: Object-Oriented Analysis	3
2.1 Overview	3
2.2 Domain Analysis	3
2.3 Generic Components of OOA	3
2.4 The OOA Process	3
2.5 The Object-Relationship Model	3
2.6 The Object-Behavior Model	4
3 Chapter 22: Object-Oriented Design	4
3.1 Overview	4
3.2 The Object-Oriented Design Pyramid	4
3.3 System Design	4
3.4 Object Design	4
3.5 Design Patterns	4
4 Chapter 23: Object-Oriented Testing	5
4.1 Overview	5
4.2 Testing OOA and OOD Models	5
4.3 Object-Oriented Testing Strategies	5
4.4 Test Case Design	5
4.5 Class and Interclass Testing Methods	5
5 Chapter 25: Formal Methods	6

5.1 Overview	6
5.2 Deficiencies of Less Formal Approaches	6
5.3 Core Concepts of Formal Methods	6
5.4 Mathematical Foundations	6
5.5 Formal Specification Languages	6
5.6 Practical Guidelines (The Ten Commandments)	7
6 Chapter 4: Requirements Engineering	7
6.1 Overview	7
6.2 Levels of Requirements	7
6.3 Types of Requirements	7
6.4 The Requirements Engineering Process	7
7 Chapter 17: Distributed Software Engineering	8
7.1 Overview	8
7.2 Key Design Issues	8
7.3 Models of Interaction and Middleware	8
7.4 Architectural Patterns	9
7.5 Software as a Service (SaaS)	9
8 Chapter 22: Project Management	9
8.1 Overview	9
8.2 Risk Management	9
8.3 Managing People	10
8.4 Teamwork	10
9 Chapter 23: Project Planning	10
9.1 Overview	10
9.2 Software Pricing	10
9.3 Plan-Driven Development	10
9.4 Project Scheduling	11
9.5 Agile Planning	11
9.6 Estimation Techniques	11
9.7 COCOMO Cost Modeling	11

Pressman

1 Chapter 12: Analysis Modeling

1.1 Overview At a technical level

Software engineering begins with analysis modeling. This activity creates a clear specification of what the software must do. It uses a combination of text and diagrams to map out the system's data, functions, and behavior. The main goal of analysis modeling is to build a representation that is easy to understand and can be thoroughly reviewed for correctness, completeness, and consistency before any code is written.

1.2 The Structure of the Analysis Model

The structured analysis model resembles a core surrounded by three specific diagrams:

Data Dictionary: This forms the core repository. It holds organized, precise definitions of all data objects, control items, and data stores used in the software. Entity Relationship Diagram (ERD): Focuses purely on data, mapping out what data objects exist and how they are connected. Data Flow Diagram (DFD): Focuses on function, illustrating how data enters the system, gets transformed by various processes, and exits the system. State Transition Diagram (STD): Focuses on behavior, showing how the system reacts and changes states in response to external events.

1.3 Data Modeling

Data modeling defines the information domain of the problem without worrying about how that data is processed.

Data Objects: A representation of any composite information the software needs to understand (like a customer, sensor, or report). Attributes: The specific properties that describe or identify a data object (like a sensor's ID number or a car's color). Relationships: The logical connections between

different data objects. Cardinality and Modality: These concepts define the limits of a relationship. Cardinality asks "how many?" (e.g., one-to-one, one-to-many), while modality asks whether the relationship is optional or mandatory.

1.4 Functional Modeling and Information Flow

Functional modeling explains how inputs are transformed into outputs.

Data Flow Diagrams (DFDs): These diagrams use basic shapes to show the journey of data. Rectangles represent external entities (like a user), circles or bubbles represent processes that change the data, arrows show the flow of data, and parallel lines show where data is stored. DFDs are built in layers, starting with a simple top-level view (Level 0) and breaking down into increasingly detailed sub-levels. Process Specifications (PSPECs): Because a diagram cannot show detailed logic, a PSPEC is written for each process bubble at the lowest level. It uses plain text or pseudo-code to explain the exact algorithm or rules that process follows.

1.5 Real-Time Extensions

Traditional structured analysis was built for basic information systems and struggled with real-time engineering applications (like avionics or manufacturing robotics). To fix this, extensions were added to handle time-continuous data and events.

Control Flow Diagrams (CFDs): These diagrams look like DFDs but use dashed arrows to show the flow of control signals or events, rather than data. Control Specifications (CSPECs): This acts as an "executive" window that dictates how the software behaves when an event occurs, often using a Process Activation Table to decide which processes to turn on or off based on incoming signals.

1.6 Behavioral Modeling

Behavioral modeling dictates how the software responds to the outside world.

State Transition Diagrams (STDs): These diagrams represent the various observable modes of behavior (states) the system can be in. Arrows connect these states to show how specific events trigger the system to transition from one state to another, and what actions happen during that transition.

1.7 The Data Dictionary

As systems grow, keeping track of every piece of data becomes incredibly difficult. The data dictionary solves this by acting as a master list. It enforces strict naming conventions, prevents duplicate entries, and tracks exactly where and how every piece of data is used across all the different diagrams. This makes it much easier to understand the potential impact of changing a specific data requirement later in the project.

2 Chapter 21: Object-Oriented Analysis

2.1 Overview

Object-Oriented Analysis (OOA) is the first technical step in building an object-oriented system. It defines the classes that represent the problem, how they relate to one another, their internal attributes and operations, and how they communicate. The main goal is to build a model that accurately reflects real-world requirements before designing and writing the software.

2.2 Domain Analysis

Domain analysis is the process of finding and defining classes that are common across a broad category of applications. By creating a library of reusable software components, development teams can build new systems faster, at a lower cost, and with higher quality, rather than starting from scratch every time.

2.3 Generic Components of OOA

An OOA model is built using two main types of components:

- **Static Components:** These describe the permanent structure of the system, including the classes themselves, their descriptive attributes, and their logical relationships with other classes.
- **Dynamic Components:** These describe how the system behaves and changes over time, including how objects communicate and how they react to specific events.

2.4 The OOA Process

The analysis process follows several key steps to break down the system:

- **Use-Cases:** The process starts by describing scenarios of how end-users or outside systems interact with the product.
- **Class-Responsibility-Collaborator (CRC) Modeling:** This step uses simple cards to identify candidate classes, define their responsibilities (what they know or do), and list their collaborators (other classes they need to interact with to finish a task).
- **Structures and Hierarchies:** Classes are organized into logical structures, such as moving from general categories to specific types, or showing how smaller parts make up a larger whole.
- **Subsystems:** For complex systems, related groups of classes are bundled into subsystems or packages to make the overall model easier to understand and manage.

2.5 The Object-Relationship Model

This model focuses on how classes are connected. Using the information gathered from CRC modeling, it maps out the network of relationships and dependencies between objects, showing exactly how and where messages flow between different parts of the system.

2.6 The Object-Behavior Model

While other models focus on static structure, the behavior model shows how the system reacts to the outside world. It maps out the active states of an object and identifies the specific external events or triggers that cause the system to transition from one state to another.

3 Chapter 22: Object-Oriented Design

3.1 Overview

Object-oriented design (OOD) takes the conceptual models created during analysis and transforms them into a detailed blueprint for building the software. The main goal is to create a highly modular system by organizing major components into subsystems and encapsulating data and operations into individual objects.

3.2 The Object-Oriented Design Pyramid

An OOD model is built in four distinct layers:

- **Subsystem layer:** Contains the major structural blocks that fulfill system requirements and provide the technical infrastructure.
- **Class and object layer:** Defines the hierarchies of classes (generalizations and specializations) and represents the individual objects.
- **Message layer:** Establishes the communication network by detailing how objects send messages to one another.
- **Responsibilities layer:** Focuses on the internal details, including the data structures for attributes and the algorithmic logic for operations.

3.3 System Design

System design focuses on the big picture and the overall software architecture. Key activities include:

- **Partitioning:** Grouping related classes, relationships, and behaviors into cohesive subsystems. These subsystems should communicate through very clear, limited interfaces.

- **Layering:** Organizing the architecture into different levels of abstraction, such as separating the user interface (presentation layer) from the data management (database layer).
- **Concurrency and Task Management:** Identifying tasks that must run simultaneously and deciding whether to assign them to different hardware processors or use operating system features to manage them.
- **Infrastructure Components:** Designing the user interface based on user scenarios, setting up data management for storing and retrieving objects, and creating mechanisms to control access to shared global resources.

3.4 Object Design

While system design acts as the floor plan, object design provides the specific details for each “room.”

- **Object Descriptions:** An object is described in two parts. The **protocol description** defines the public messages it can receive (for users of the object). The **implementation description** defines the hidden internal data structures and procedures (for the supplier of the object).
- **Algorithms and Data Structures:** Detailed algorithms are written to carry out each operation, and data structures are defined to manage the object’s specific attributes.
- **Program Components:** The design specifies how these objects will be written in a programming language, clearly separating public interfaces from private, hidden details to ensure information hiding.

3.5 Design Patterns

Design patterns are recurring, proven solutions to common design problems. Instead of starting from scratch, software engineers use these patterns as building blocks.

- **Applying Patterns:** Patterns provide ready-made templates for classes, responsibilities, and collaborations.
- **Composition over Inheritance:** When using patterns to build complex systems, it is generally better to assemble smaller, focused objects together

(composition) rather than creating massive, complicated family trees of classes (inheritance). This keeps the design flexible and easy to maintain.

4 Chapter 23: Object-Oriented Testing

4.1 Overview

The primary goal of object-oriented testing is the same as traditional testing: finding the greatest possible number of errors with a manageable amount of effort. However, the nature of object-oriented software requires a shift in strategy. Testing must broaden to include formal technical reviews of the analysis and design models to catch errors early, well before any code is written.

4.2 Testing OOA and OOD Models

Instead of executing code, early testing involves rigorously reviewing the conceptual models for correctness and consistency.

- **Correctness:** Models are reviewed by domain experts to ensure they accurately represent the real-world problem domain and adhere to proper modeling conventions.
- **Consistency:** The class-responsibility-collaborator (CRC) cards and object-relationship diagrams are cross-checked. This ensures that all class collaborations, connections, and delegated responsibilities are valid and logically organized.

4.3 Object-Oriented Testing Strategies

The traditional hierarchy of software testing is adapted to fit the object-oriented paradigm.

- **Unit Testing (Class Testing):** Because objects encapsulate both data and operations, the smallest testable unit is the class itself, rather than an individual module. Operations are not tested in isolation, but rather within the context of the class and its specific attributes.

- **Integration Testing:** Conventional top-down or bottom-up integration is ineffective because object-oriented systems lack a traditional hierarchical control structure. Instead, teams use **thread-based testing** (integrating the classes required to respond to a specific event) or **use-based testing** (testing independent classes first, followed by the dependent classes that rely on them).
- **Validation Testing:** At the system level, testing focuses on user-visible actions and outputs, heavily driven by the use-case scenarios created during the analysis phase.

4.4 Test Case Design

Designing tests for object-oriented systems involves evaluating both the surface structure (user interactions) and the deep structure (internal technical architecture).

- **Fault-Based Testing:** Testers hypothesize plausible faults, such as integration issues or boundary errors, and design specific tests to prove or disprove their existence.
- **Scenario-Based Testing:** This method focuses on the user's perspective, utilizing use-cases to uncover errors in how different subsystems interact while performing realistic, everyday tasks.

4.5 Class and Interclass Testing Methods

When testing individual classes and the ways they communicate with one another, several tactical methods are applied.

- **Random Testing:** A sequence of operations is randomly generated based on the acceptable life history and constraints of a class instance.
- **Partition Testing:** To reduce the overall number of test cases needed, operations are grouped into distinct categories based on their function, the specific attributes they manipulate, or how they change the internal state of the object.
- **Behavioral Models:** State transition diagrams are used to track the behavioral flow of the system, ensuring that a class and its collaborators

properly transition through all allowable states in response to external events.

5 Chapter 25: Formal Methods

5.1 Overview

Formal methods represent the highly mathematical end of the software engineering spectrum. They use set theory and logic to create software specifications that are more complete, consistent, and unambiguous than traditional methods. Because they reduce specification errors dramatically, formal methods are especially important for safety-critical or mission-critical systems where software failures could result in severe economic consequences or loss of life.

5.2 Deficiencies of Less Formal Approaches

Traditional analysis and design methods rely heavily on natural language and graphical notations, which can introduce several problems into a specification.

- **Contradictions:** Different parts of the system specification conflict with one another.
- **Ambiguities:** Statements that can be interpreted in multiple different ways.
- **Vagueness:** Imprecise statements that lack useful, actionable information.
- **Incompleteness:** Omitting important details or operational scenarios by mistake.
- **Mixed levels of abstraction:** Randomly mixing high-level system overviews with low-level technical details.

5.3 Core Concepts of Formal Methods

A formal specification is built around three main mathematical concepts.

- **Data Invariant:** A set of mathematical conditions or constraints that must remain true throughout the entire execution of the system.
- **State:** The specific collection of stored data that the system accesses and alters during its operation.
- **Operations:** The actions that take place within the system to read or write data to the state. Every operation has two rules: a **precondition** (the circumstances that must be met for the operation to be valid) and a **postcondition** (the exact changes made to the state after the operation finishes).

5.4 Mathematical Foundations

Software engineers using formal methods must understand discrete mathematics.

- **Sets:** Collections of unique objects with no duplicates. Sets can be defined by listing their items explicitly or by using a **constructive specification**, which uses a mathematical rule (Boolean expression) to define the set's contents.
- **Set Operators:** Standard mathematical symbols are used to check set membership or combine sets through operations like union, intersection, and difference.
- **Logic Operators:** The algebra of true and false expressions, including standard operators like “and”, “or”, “not”, and “implies”.
- **Sequences:** Unlike sets, sequences are mathematical structures where the elements are ordered and duplicates are allowed.

5.5 Formal Specification Languages

A formal specification language (such as Z, VDM, or LARCH) provides the specific mathematical tools to write the specification.

- **Syntactic Domain:** The actual notation and symbols used, usually derived from standard set theory and predicate calculus.
- **Semantic Domain:** The rules that dictate how the language represents system requirements and behaviors.

- **The Z Language:** Z is a popular formal language that structures specifications into **schemas**. A schema is a box-like structure that declares variables and lists the mathematical predicates that constrain them, acting much like a procedure or subroutine in a programming language.

5.6 Practical Guidelines (The Ten Commandments)

Applying formal methods in the real world requires careful planning. The “Ten Commandments” of formal methods serve as a guide, suggesting that development teams should not overformalize every part of a system and must anticipate high initial training costs. Furthermore, software engineers must not abandon traditional development methods or rigorous software testing, as formal methods are best used in combination with other techniques to ensure high-quality software.

Somerville

6 Chapter 4: Requirements Engineering

6.1 Overview

Requirements engineering is the process of discovering, analyzing, documenting, and checking the services a system must provide and the constraints under which it must operate. It is a critical early stage in software development that helps establish exactly what the system needs to do before implementation begins.

6.2 Levels of Requirements

Requirements are written at different levels of detail depending on who is reading them:

- **User Requirements:** High-level statements written in plain language (often with diagrams) that describe what services the system provides and its general constraints. They are meant for customers, end-users, and managers.

- **System Requirements:** Detailed, precise descriptions of the software’s exact functions, services, and operational constraints. They often act as a formal contract and a specific blueprint for the software developers.

6.3 Types of Requirements

- **Functional Requirements:** These specify what the system should do, how it should react to specific inputs, and how it should behave in particular situations.
- **Non-Functional Requirements:** These dictate constraints on the system, such as performance speed, security, reliability, or regulatory compliance. They generally apply to the system as a whole rather than individual features. These are often more critical than functional requirements, as failing to meet them can render the entire system completely unusable.

6.4 The Requirements Engineering Process

Instead of a simple straight line, this process is highly iterative and involves four main interconnected activities:

1. **Requirements Elicitation** This is the process of interacting with stakeholders to discover what they need. Because users often struggle to articulate their precise needs, engineers use several techniques:
 - **Interviews:** Talking directly with stakeholders through predefined questions or open-ended discussions.
 - **Ethnography:** Observing users in their actual work environment to uncover implicit needs and real-world workflows that people might not think to mention in an interview.
 - **Stories and Scenarios:** Using real-life examples and step-by-step narratives to help users easily relate to and envision how they will interact with the new system.
2. **Requirements Specification** This involves translating the gathered information into a formal requirements document. While plain natural

language is common, it can sometimes be vague or ambiguous. To minimize misunderstandings, engineers use structured natural language (standardized forms or templates), decision tables, or visual models like use cases to map out system interactions.

3. Requirements Validation This is the process of rigorously checking the drafted requirements to ensure they accurately represent what the customer actually wants. Finding and fixing a requirement error early is much cheaper than fixing a coded system later. The review checks if the requirements are valid, consistent (no contradictions), complete, realistic within the budget, and testable.

4. Requirements Management In reality, requirements constantly evolve due to changing business environments, new technologies, or shifting stakeholder priorities. Requirements management is the formal process of tracking individual requirements and controlling how future changes are proposed, analyzed for cost and impact, and successfully implemented into the system documentation.

7 Chapter 17: Distributed Software Engineering

7.1 Overview

A distributed system is a collection of independent computers that work together to appear as a single, unified system to the user. While they offer significant benefits—such as resource sharing, concurrency, scalability, and fault tolerance—they are inherently more complex to design and manage due to network unpredictability and the lack of a single centralized control.

7.2 Key Design Issues

When designing distributed systems, engineers must balance several critical challenges:

- **Transparency:** Deciding how much of the system's distributed nature (like network delays or multiple servers) should be hidden from the user.

- **Openness:** Designing systems using standard internet protocols so that software and hardware from different vendors can seamlessly work together.
- **Scalability:** Ensuring the system can handle growth. This can involve “scaling up” (using more powerful hardware) or “scaling out” (adding more computers to the network).
- **Security:** Protecting the distributed network against attacks like data interception, service interruption, unauthorized modifications, and fabricated data.
- **Quality of Service (QoS):** Delivering a dependable response time and throughput, which is especially critical for time-sensitive data like streaming video or audio.
- **Failure Management:** Designing mechanisms to detect, contain, and automatically recover from inevitable hardware or network failures.

7.3 Models of Interaction and Middleware

Computers in a distributed system communicate in two primary ways:

- **Procedural Interaction:** A synchronous method where one computer calls a service on a remote computer and waits for the result (e.g., Remote Procedure Calls). Both sides must be online at the same time.
- **Message-Based Interaction:** A method where a computer sends a structured message detailing what it needs. This is more flexible and tolerates system downtime better, as messages can wait in a queue until the receiver is ready.

To manage these complex interactions, systems use **Middleware**. This is specialized software that sits between the operating system and the applications. It acts as a translator and coordinator, providing common services (like security and transaction management) so different components can communicate without needing to know physical network locations.

7.4 Architectural Patterns

Distributed systems are organized using various architectural styles based on their goals:

- **Master-Slave:** Commonly used in real-time systems. A “master” process coordinates the overall system, while “slave” processes handle specific dedicated tasks, like gathering sensor data or controlling hardware.
- **Two-Tier Client-Server:** The simplest model, splitting duties between a user’s device (client) and a central server. In a **thin-client** model, the server does almost all the processing, and the client just displays the interface (like a basic web browser). In a **fat-client** model, the user’s device does a lot of the computational work, relying on the server mostly for database management.
- **Multi-Tier Client-Server:** The system is broken into three or more distinct layers (such as user interface, application logic, and database processing) running on separate servers. This greatly improves performance and makes it easier to scale the system.
- **Distributed Component:** Instead of strict layers, the system is built as a web of independent components that provide specific services. It is highly flexible and allows new resources to be added dynamically, though it is very complex to design.
- **Peer-to-Peer (P2P):** There is no central server. Every computer (node) in the network acts as both a client and a server, sharing its own processing power and storage. This is highly redundant and useful for massive computational tasks or sharing files.

7.5 Software as a Service (SaaS)

SaaS is a modern deployment model where software is hosted remotely (usually in the cloud) and accessed by users via a web browser, usually paid for via subscription rather than a one-time purchase.

- **Configurability:** The software must be flexible enough to adapt to the specific branding, business rules, and workflows of different organizations without needing a separate installation for each one.

- **Multi-Tenancy:** The system must efficiently share the exact same underlying computing resources among many different companies (tenants) while keeping each company’s data completely isolated and secure.
- **Scalability:** The application must be designed using independent, stateless services so that the provider can instantly add more servers to handle unpredictable spikes in user demand without degrading performance.

8 Chapter 22: Project Management

8.1 Overview

Software project management ensures that software is delivered on time, within budget, and meets customer expectations while maintaining a well-functioning team. Managing software is uniquely challenging because the product is intangible, large projects are often unique, and development processes vary widely between organizations. Key management activities include project planning, risk management, people management, reporting, and proposal writing.

8.2 Risk Management

Risk management involves anticipating and mitigating threats to the project schedule, software quality, or the business itself. The process follows four main stages:

- **Risk Identification:** Discovering potential estimation, organizational, people, requirements, technology, and tool risks.
- **Risk Analysis:** Judging the probability (from insignificant to very high) and the seriousness (from insignificant to catastrophic) of each identified risk.
- **Risk Planning:** Developing strategies to handle risks. These include avoidance to reduce the probability of a risk, minimization to reduce its impact, and contingency plans to prepare for the worst.

- **Risk Monitoring:** Regularly tracking risk indicators during management reviews to check if the probability or impact of risks has changed over time.

8.3 Managing People

People are a software organization's greatest asset. Effective people management relies on treating staff with consistency, respect, inclusion, and honesty.

- **Motivation:** Managers must organize the work environment to satisfy team members' social, esteem, and self-realization needs.
- **Personality Types:** People are motivated differently based on their personalities. **Task-oriented** people are driven by the intellectual challenge of the work, **self-oriented** people are driven by personal success and recognition, and **interaction-oriented** people are motivated by the presence and actions of their co-workers.

8.4 Teamwork

Professional software is built by teams, with small, cohesive groups of 4 to 6 members being the most effective. Cohesive groups naturally establish their own quality standards, learn from each other, and share knowledge seamlessly.

- **Group Composition:** A successful team requires a mix of technical skills and complementary personalities. Having interaction-oriented people is especially important because they facilitate communication and detect early tensions.
- **Group Organization:** Teams can be organized informally (common in Agile methods) or hierarchically. Informal structures usually work best for complex software engineering because they allow for rapid communication and consensus across all levels.
- **Group Communications:** Effective communication is vital and is influenced by the group's size, structure, and composition. The physical workspace (balancing private concentration areas with shared spaces) and

the choice of communication channels (like face-to-face meetings, wikis, blogs, or remote collaboration tools) deeply impact how well information flows within the team.

9 Chapter 23: Project Planning

9.1 Overview

Project planning is an essential management activity that involves scheduling, estimating costs, and pricing a software project. Planning occurs at three main stages: during the initial proposal to secure a contract, at project startup to allocate resources, and continuously throughout development as requirements evolve and more information is uncovered.

9.2 Software Pricing

The price quoted for a software system is rarely just the development cost plus a standard profit margin. Prices are influenced by broader business factors such as the company's financial health, market opportunities, the potential for changing requirements, and uncertainty in cost estimates. A common strategy is "pricing to win," where a company bids based on what they believe the customer expects to pay. Once the contract is secured, the system's exact requirements are negotiated so that development costs remain within the agreed budget.

9.3 Plan-Driven Development

In a traditional plan-driven approach, the entire development process is detailed in advance to track progress and organize the team.

- **The Project Plan:** This core document defines the project's objectives, team organization, risk analysis, hardware and software resource requirements, work breakdown, and mechanisms for monitoring progress.
- **The Planning Process:** Planning is highly iterative. Managers continuously review actual progress against the plan, handle minor schedule

slippages, and initiate risk mitigation actions or complete project re-planning when serious problems arise.

9.4 Project Scheduling

Scheduling breaks the project down into individual, manageable tasks (ideally taking between one and eight weeks), estimating the necessary effort, duration, and dependencies for each.

- **Visualizing Schedules:** Managers typically use calendar-based bar charts (Gantt charts) to visualize task start and end dates, parallel activities, and staff allocation over time.
- **Milestones and Deliverables:** Schedules rely on milestones (logical review points marking the end of a stage) and deliverables (tangible work products handed over to the customer).

9.5 Agile Planning

Agile planning is highly flexible, adapting to changing customer requirements by making decisions during development rather than upfront. It occurs in two stages: release planning (looking months ahead to select features) and iteration planning (planning the exact work for the next 2 to 4 weeks).

- **The Planning Game:** The team and customer define functionality using user stories. The team ranks these stories and assigns relative effort points. Using their “velocity” (the number of points completed per day), they determine how many stories can fit into an iteration.
- **Task Allocation and Scope:** Developers voluntarily sign up for tasks. Importantly, if the planned work cannot be completed, the delivery schedule is never extended; instead, the scope of the software increment is reduced.

9.6 Estimation Techniques

Estimating the effort and schedule for a project is notoriously difficult early on. Organizations generally use two approaches:

- **Experience-Based:** Managers predict effort based on their judgment and past experience with similar projects.
- **Algorithmic Cost Modeling:** Uses a mathematical formula to predict project costs based on the estimated size of the software, system complexity, and other project factors.

9.7 COCOMO Cost Modeling

COCOMO II is a comprehensive algorithmic cost model that provides increasingly detailed estimates as the project progresses through different submodels:

- **Application Composition Model:** Estimates effort for systems built rapidly from reusable components or scripts, using an “application point” metric.
- **Early Design Model:** Used once requirements are agreed upon but before a detailed architecture is set. It uses function points and a simplified set of seven multipliers to estimate costs.
- **Reuse Model:** Estimates the distinct effort required to assess, understand, and integrate reusable “white-box” code into the new system.
- **Post-Architecture Model:** The most detailed model, utilized once the system architecture is designed. It uses a comprehensive set of 17 cost drivers (such as required reliability, tool use, and team cohesion) to refine the overall estimate.
- **Staffing Impact:** The model highlights that doubling the workforce does not halve the schedule. Adding people increases communication overhead and reduces individual productivity, meaning that adding staff to a delayed project can sometimes delay it even further.