Agile Software development

Vaishnavi

Assistant Professor

Agile is a set of values, principles, and practices that enable individuals and teams to create and respond to change in uncertain and turbulent environments.

An **Agile team** is a small, self-organizing, and cross-functional group responsible for delivering working software in short, iterative cycles (sprints). The team follows Agile principles, emphasizing collaboration, adaptability, and continuous improvement.

Key features of **agile**:

1. Specification, design, and coding happen together.
2. Software is built in small increments with user feedback.
3. Tools are used for testing, integration, and UI.

- **Agility** in software development refers to the ability to respond quickly and effectively to changing customer needs, market conditions, and technological advancements. It emphasizes adaptability, continuous improvement, and delivering value in small, frequent releases rather than following a rigid, pre-defined plan.

Incremental approach:

It is a method where the system is developed and delivered in small, functional pieces called increments, rather than waiting for the full system to be built before delivery.

Working of Incremental approach:

1. The project is divided into smaller increments, each delivering a portion of the system's functionality.
2. Each increment includes activities like requirements gathering, design, coding, and testing.
3. Customers review each increment and provide feedback, which is incorporated into future increments.
4. The system evolves until it reaches its final version.

characteristics of agility :

1. **Flexibility**: Ability to adapt to changing customer needs, market demands, or technology. The plans are not rigid; changes can be incorporated at any stage. Example: Adding a new feature based on customer feedback mid-project.
2. **Customer Focus**: The product is built around user needs and expectations. Example: Regular demos and review meetings with stakeholders.
3. **Incremental Delivery:** Software is delivered in small increments, enabling early use and feedback. Example: Start with login and profile features, then add shopping cart and payment.
4. **Collaboration:** Emphasizes teamwork between developers, testers, business analysts, and customers. Example: Daily stand-ups and cross-functional sprint planning sessions.
5. **Continuous Improvement:** Teams reflect on past work to identify what can be improved. It implements improvements in subsequent iterations. Example: Retrospective meetings at the end of each sprint.
6. **Simplicity:** Build only what's necessary to meet user needs. Example: Start with a simple, functional interface before adding complexity.
7. **Technical Excellence:** Promotes clean code, good design, and testing for maintainable, scalable software. Example: Regular code reviews and coding standards.
8. **Sustainable Pace:** Plans workload to avoid burnout and maintain steady productivity. Example: Set realistic sprint goals, avoid overtime.

Agile Methodology

Agile methodology is a flexible approach to software development that focuses on delivering value to customers quickly through small, frequent releases. It emphasizes collaboration, adaptability, and continuous improvement rather than following a rigid, step-by-step process.

Key Features:

- **Iterative Development** – Work is divided into short cycles (sprints) of 1–4 weeks.
- **Incremental Delivery** – Each sprint delivers a usable part of the software.
- **Customer Collaboration** – Continuous feedback ensures the product meets evolving needs.
- **Adaptability** – Plans and priorities can be adjusted as requirements change.
- **Minimal Documentation** – Focus is on working software and communication over heavy paperwork.

Benefits:

- Faster delivery of usable features.
- Improved ability to adapt to changing requirements.
- Increased customer satisfaction through frequent feedback.

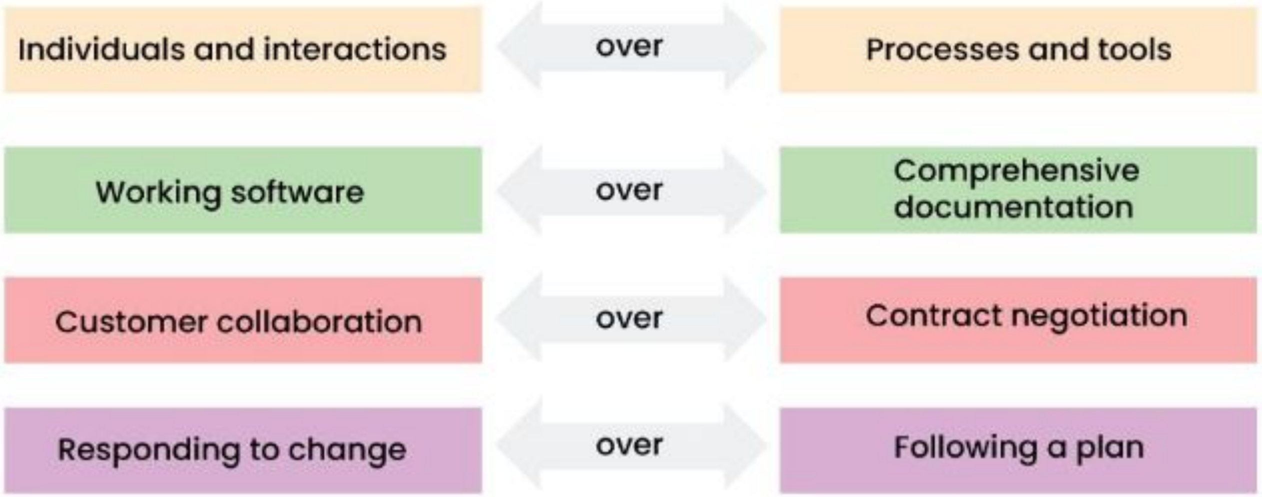
The **Agile Manifesto**, created in 2001, has **4 values** and **12 principles** that guide software development, focusing on **teamwork, flexibility, and customer satisfaction**.

 Individuals and interactions over processes and tools

 Working software over comprehensive documentation

 Customer collaboration over contract negotiation

 Responding to change over following a plan



Individuals and interactions over Processes and tools

Working software over Comprehensive documentation

Customer collaboration over Contract negotiation

Responding to change over Following a plan

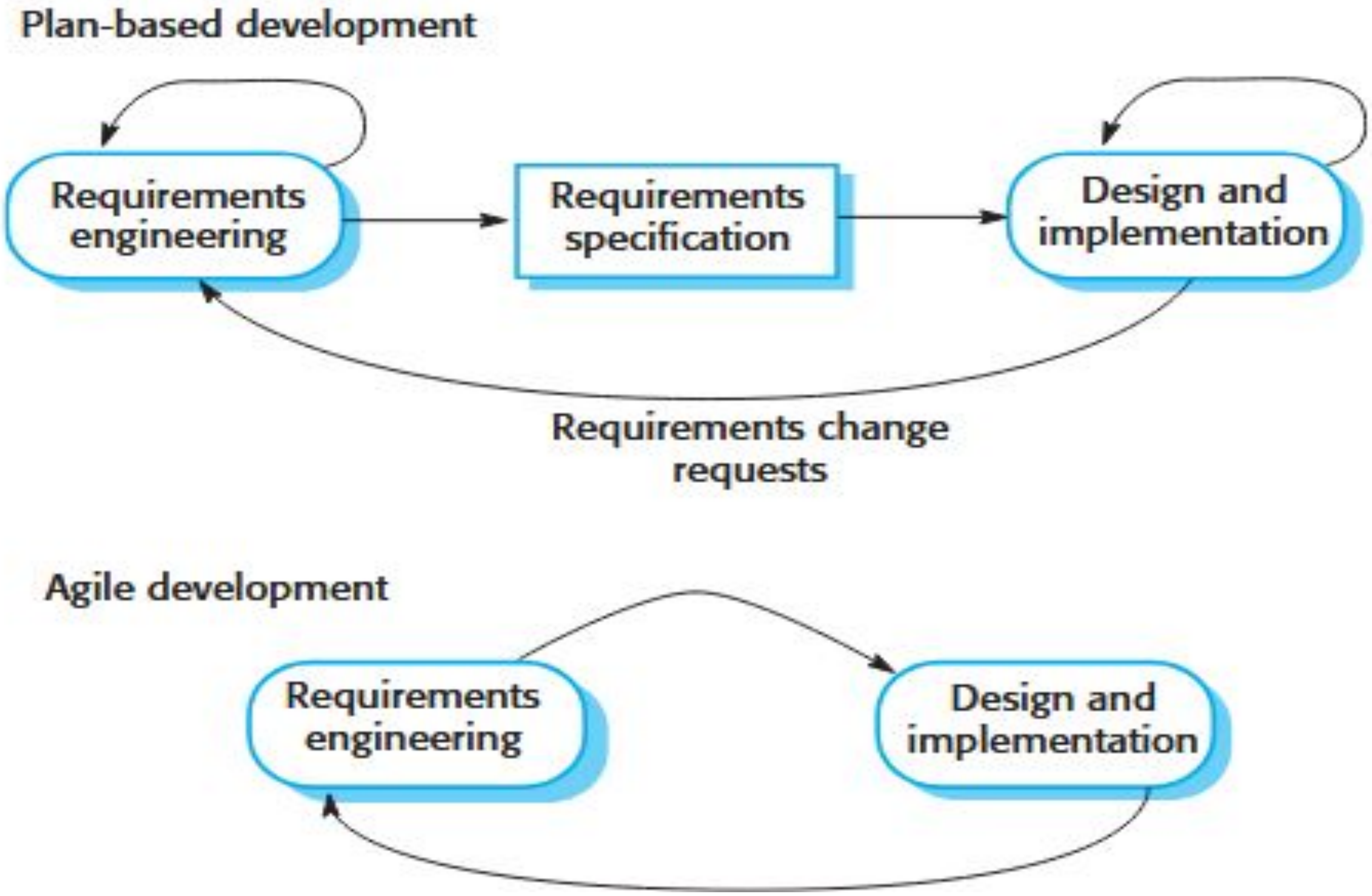

Plan-based development
Requirements engineering
Requirements specification
Design and implementation
Requirements change requests
Agile development
Requirements engineering
Design and implementation

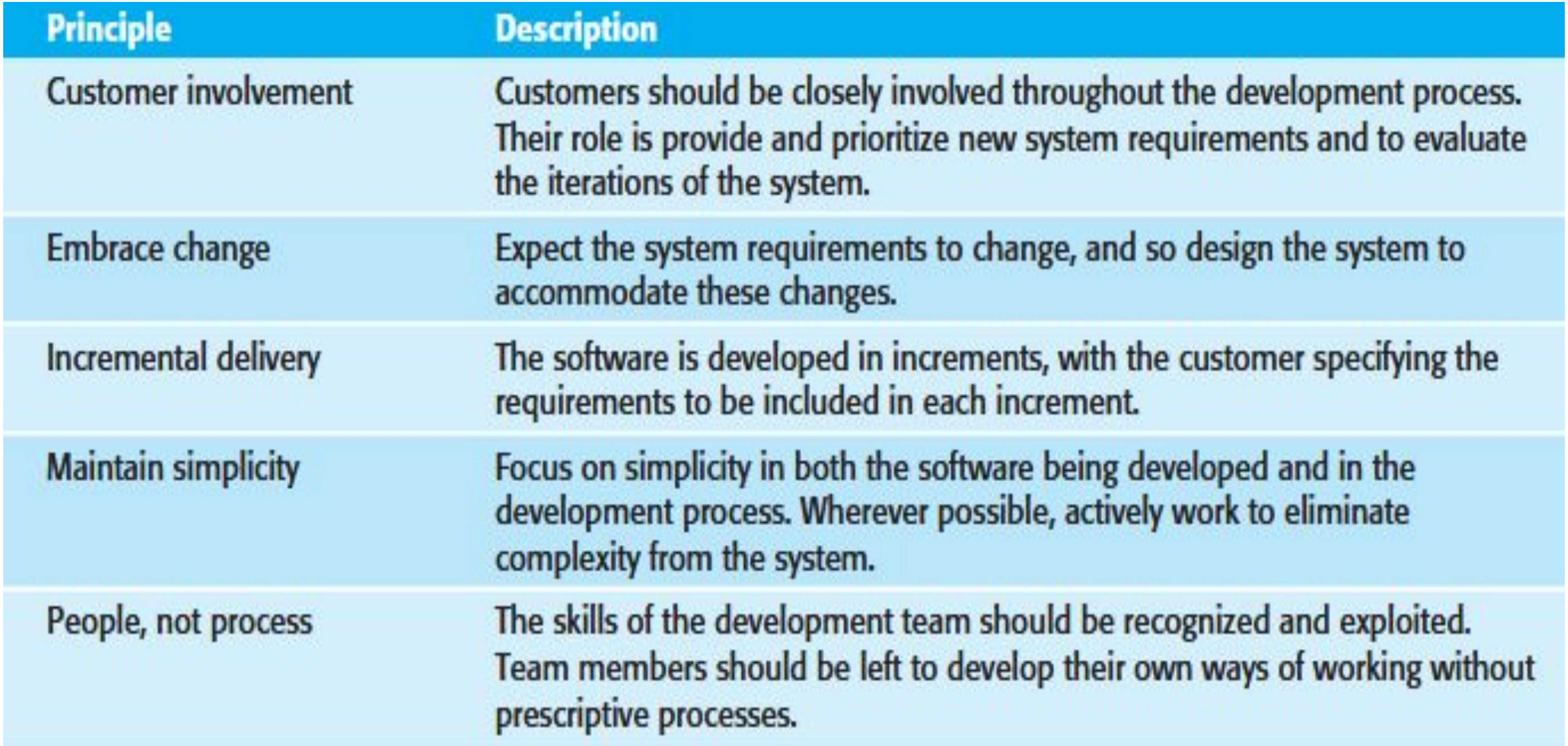
Figure 3.1 Plan-driven and agile development

3.1 - Agile methods:

- **1980s–1990s:** Plan-driven methods dominated software development.
- **Focus:** Heavy planning, documentation, and strict processes.
- **Best for:** Large, long-term systems (e.g., aerospace, government).
- **Problems:** Too much overhead for small/medium projects; slow to adapt to changes.
- **Late 1990s:** Agile methods emerged in response.

Two perspectives on scaling agile methods are:

1. **Scaling Up**: Applying Agile practices to large, complex systems that require multiple teams and possibly multiple organizations. It focuses on coordination, integration, and managing dependencies across teams. Example: A large banking application developed by 15+ teams working on different modules.
2. **Scaling Out:** Expanding Agile methods across the entire organization, beyond the software development teams. It involves adopting Agile principles in areas like marketing, HR, and operations to improve overall agility. Example: Using Agile for product development, customer support, and business strategy alignment.



Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Embrace change	Expect the system requirements to change, and so design the system to accommodate these changes.
Incremental delivery	The software is developed in increments, with the customer specifying the requirements to be included in each increment.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.
People, not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.

Figure 3.2 The principles of agile methods

Agile methods works best for two kinds of system development:

1. **Product development:** Used for developing **new products** where requirements evolve based on market trends and customer feedback. Example: Building a new mobile app where features are added and improved through user feedback.

2. **Custom system development**: Applied when creating **tailored solutions** for specific clients or organizations with unique needs. Example: Developing a custom CRM (Customer Relationship Management) system for a company with specialized workflows.

3.2 - Agile development techniques

Extreme Programming (XP) is an Agile software development methodology introduced by **Kent Beck in the 1990s**. It focuses on improving software quality and responsiveness to changing requirements through frequent releases and strong collaboration between developers and customers.

Key Practices

- **Frequent Releases** – Deliver small, functional increments of software frequently.
- **Pair Programming** – Two developers work together at one workstation to improve code quality.
- **Test-First Development (TDD)** – Tests are written before the code to ensure correctness and maintainability.
- **User Stories** – Requirements are captured in short, simple descriptions of desired functionality.
- **Continuous Integration** – Code is integrated and tested daily to detect issues early.

Its goal is to produce high-quality software quickly while accommodating changing customer requirements through constant feedback and collaboration.

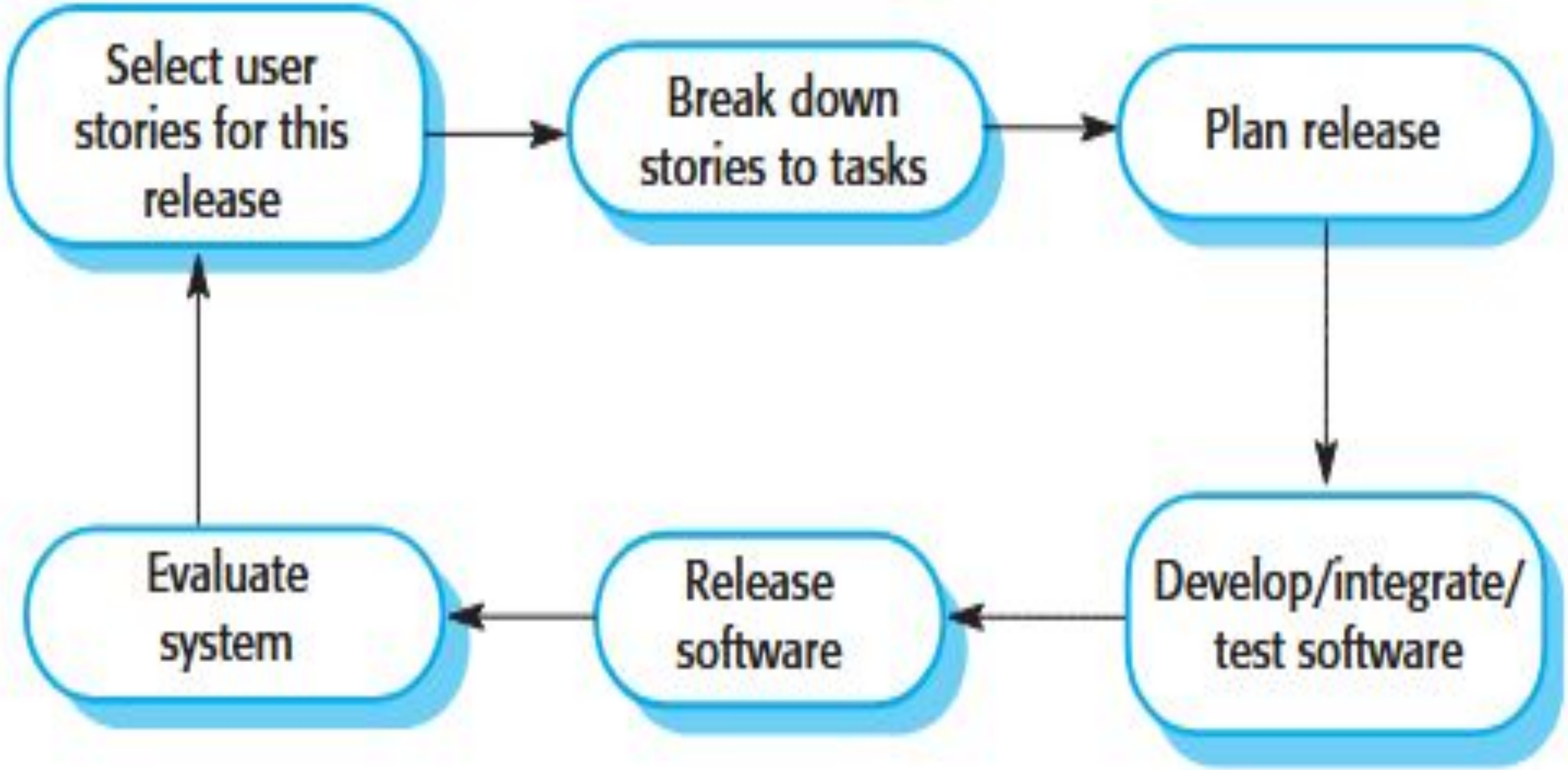

Select user stories for this release
Break down stories to tasks
Plan release
Develop/integrate/ test software
Release software
Evaluate system

Figure 3.3 The XP release cycle

The fig 3.3, illustrates the **XP release cycle**, which shows how features are selected, developed, and delivered in small increments.

1. Select User Stories for This Release: User stories (short descriptions of features) are chosen based on customer priorities and business value. These stories define what functionality will be included in the upcoming release.

2. Break Down Stories to Tasks : Each user story is divided into smaller development tasks. These tasks are manageable units of work for developers to implement.

3. Plan Release: A release plan is created, outlining which stories will be delivered and when. Effort estimation and resource planning are done at this stage.

4. Develop / Integrate / Test Software: Development teams write code, integrate components, and perform continuous testing. Practices like pair programming and test-first development (TDD) are used to maintain quality.

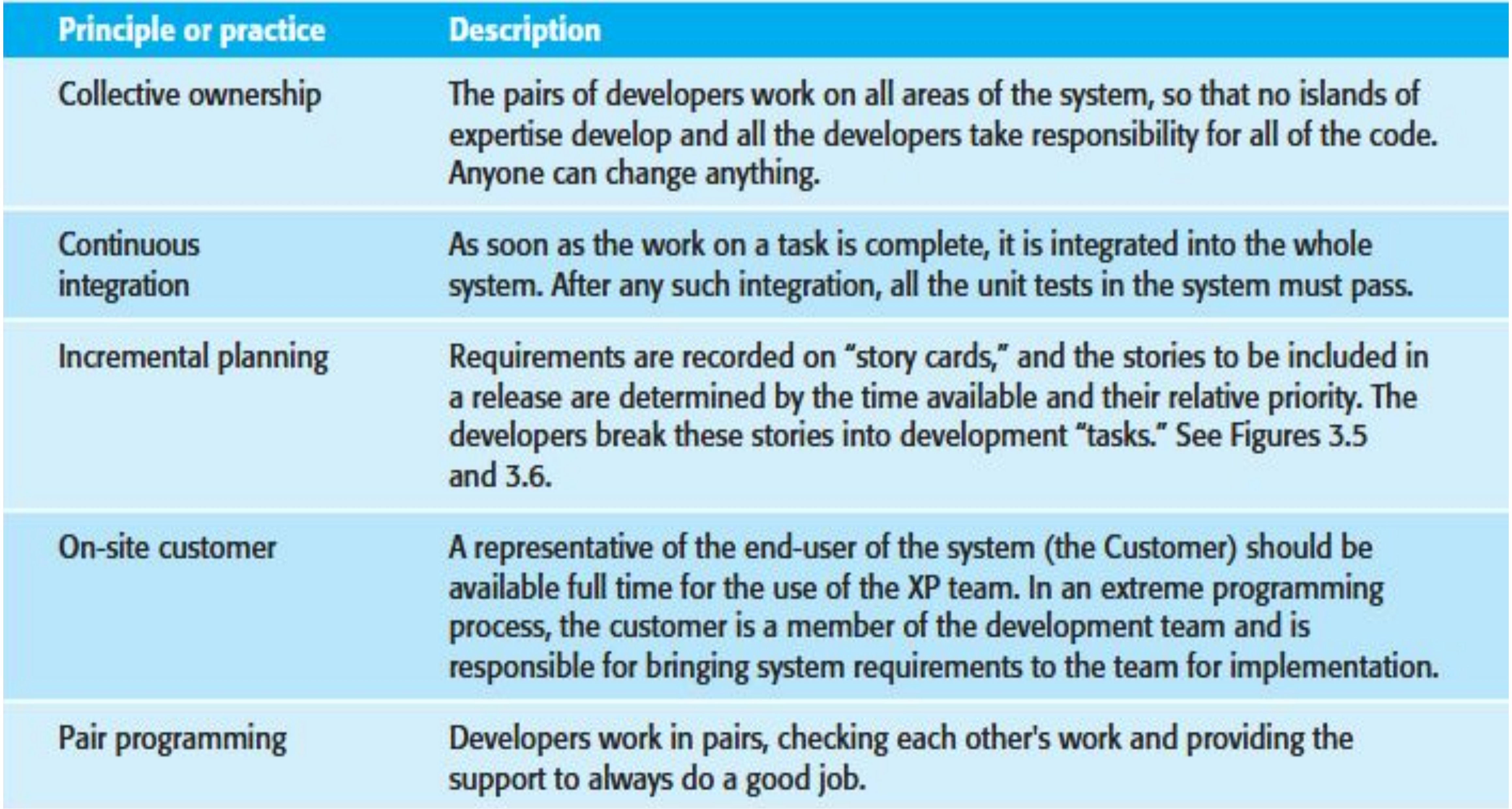
5. Release Software: Once tested, the increment is delivered to the customer. The customers can start using the new features immediately.

6. Evaluate System: Feedback is collected from users and stakeholders. Performance, usability, and feature improvements are assessed. The insights are used to plan the next iteration.

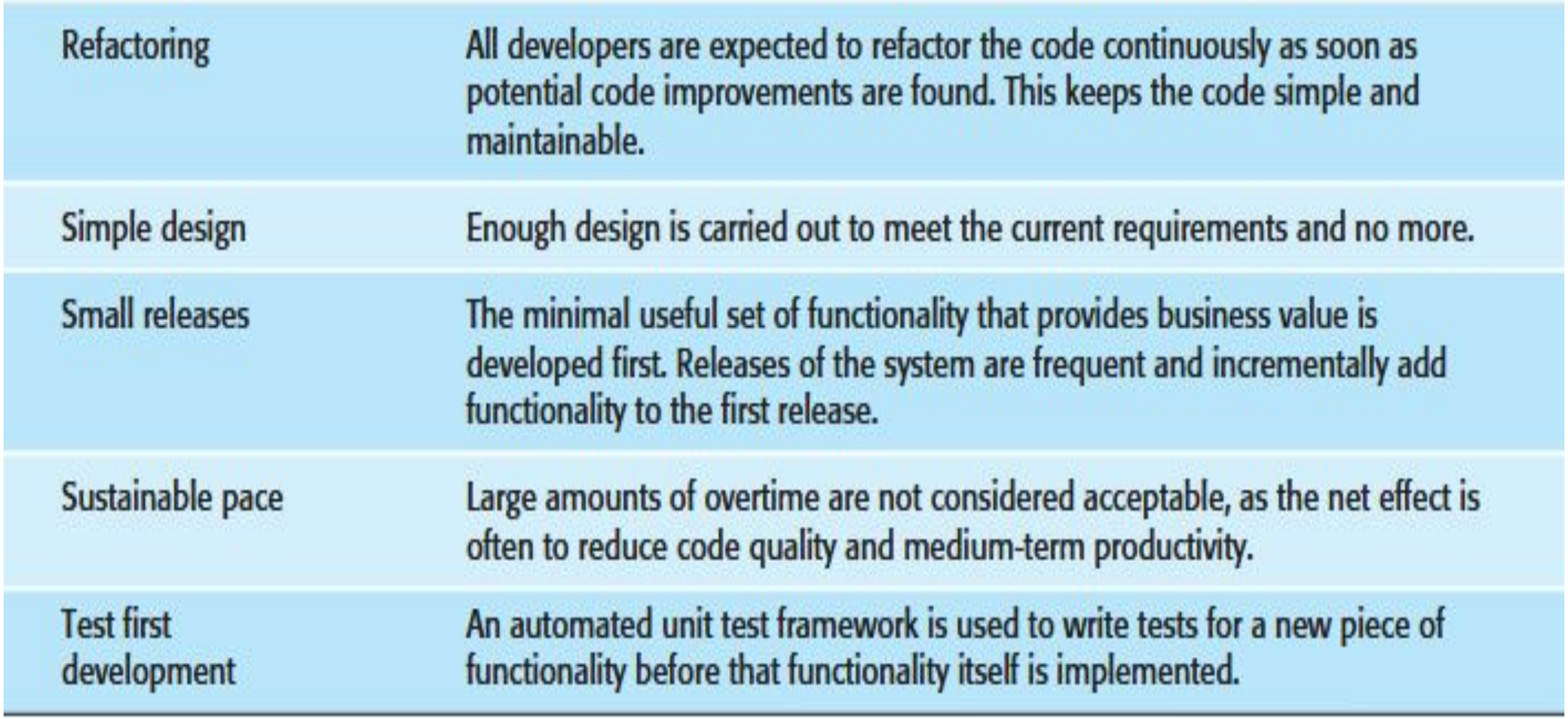
The cycle continues with new user stories for the next release, ensuring frequent delivery and continuous improvement.

Extreme Programming (XP) focuses on:

- Frequent small releases with simple user stories.
- Continuous customer involvement.
- Pair programming and shared code ownership.
- Test-first coding and regular refactoring.
- Keeping designs simple and adaptable.



Principle or practice	Description
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Incremental planning	Requirements are recorded on "story cards," and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development "tasks." See Figures 3.5 and 3.6.
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.



Refactoring	All developers are expected to refactor the code continuously as soon as potential code improvements are found. This keeps the code simple and maintainable.
Simple design	Enough design is carried out to meet the current requirements and no more.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Sustainable pace	Large amounts of overtime are not considered acceptable, as the net effect is often to reduce code quality and medium-term productivity.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.

Figure 3.4 Extreme programming practices

3.2.1 - User stories

- Agile uses **user stories** to capture changing requirements as simple, user-focused scenarios.
- Customers help create, prioritize, and refine these stories.
- Stories guide short development cycles with regular releases.

3.2.2 – Refactoring:

- XP avoids planning for future changes; it's often unnecessary.
- Instead, XP uses **continuous refactoring -** constantly improving code to keep it clean and flexible.
- Refactoring keeps code **clean, simple, and easier to change** later.
- Examples: removing duplicate code, renaming methods, reorganizing classes.

3.2.3 - Test-first development:

- In **incremental development**, there's no full spec for testing, so XP introduced a new approach:
- **Test-first development** (write tests before code)
- **Automated testing**
- **Customer involvement** in creating acceptance tests
- Tests based on user stories and tasks

-Benefits:

- Catches problems early
- Reduces misunderstandings
- Avoids "test-lag"
- Tests run quickly and often

-Challenges:

- Incomplete test coverage
- Some tests (e.g., UI logic) are hard to write
- Developers may skip or simplify tests

-Tools like **JUnit** help automate testing and maintain quality.

3.2.4 - Pair programming

Pair programming (from XP) involves two programmers working together at one computer, with pairs rotating over time.

Pros:

- Shared code ownership
- Fewer bugs (peer review)
- Encourages refactoring
- Shares knowledge across the team

Cons:

- May reduce speed
- Mixed results in studies
- Some teams prefer a mix of solo and pair work

3.3 - Agile project Management

Agile Project Management focuses on iterative development, customer collaboration, and continuous improvement. **Scrum** is one of the most widely used Agile frameworks, and the diagram illustrates its **Sprint Cycle**, which organizes work into fixed-length iterations (usually 2–4 weeks).

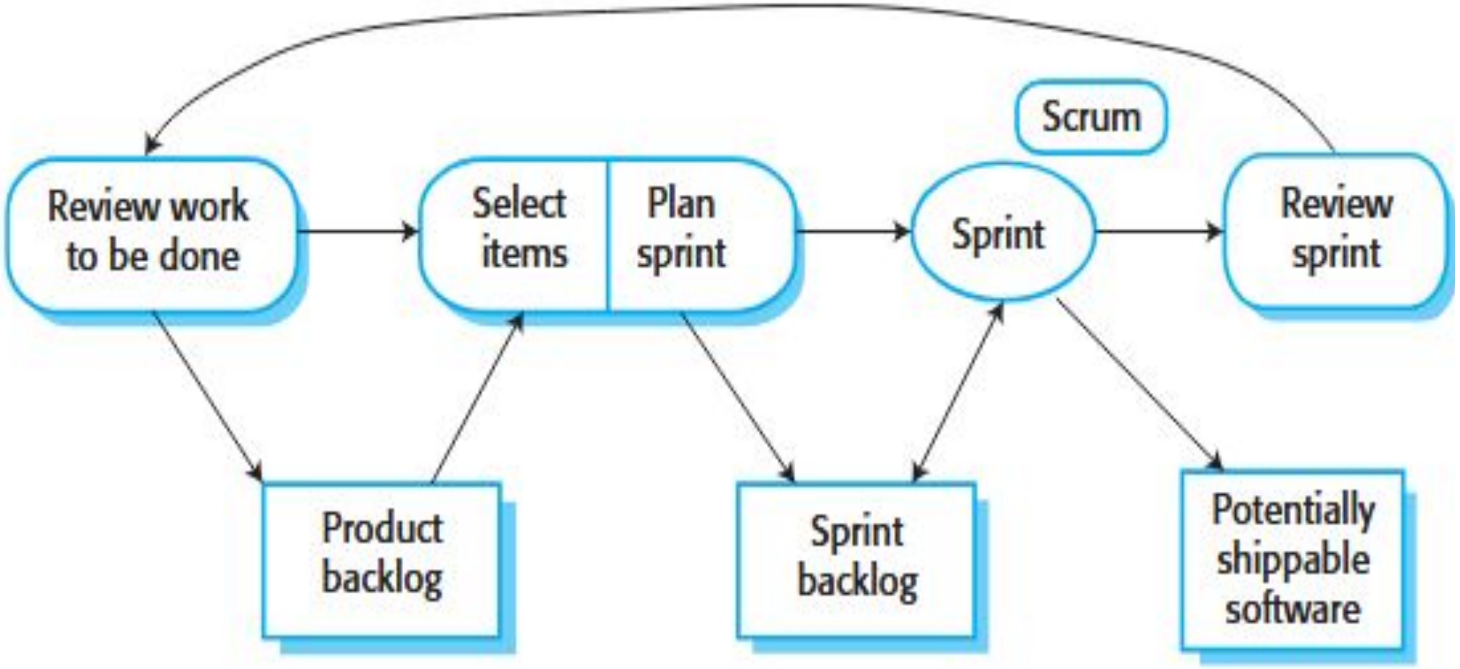


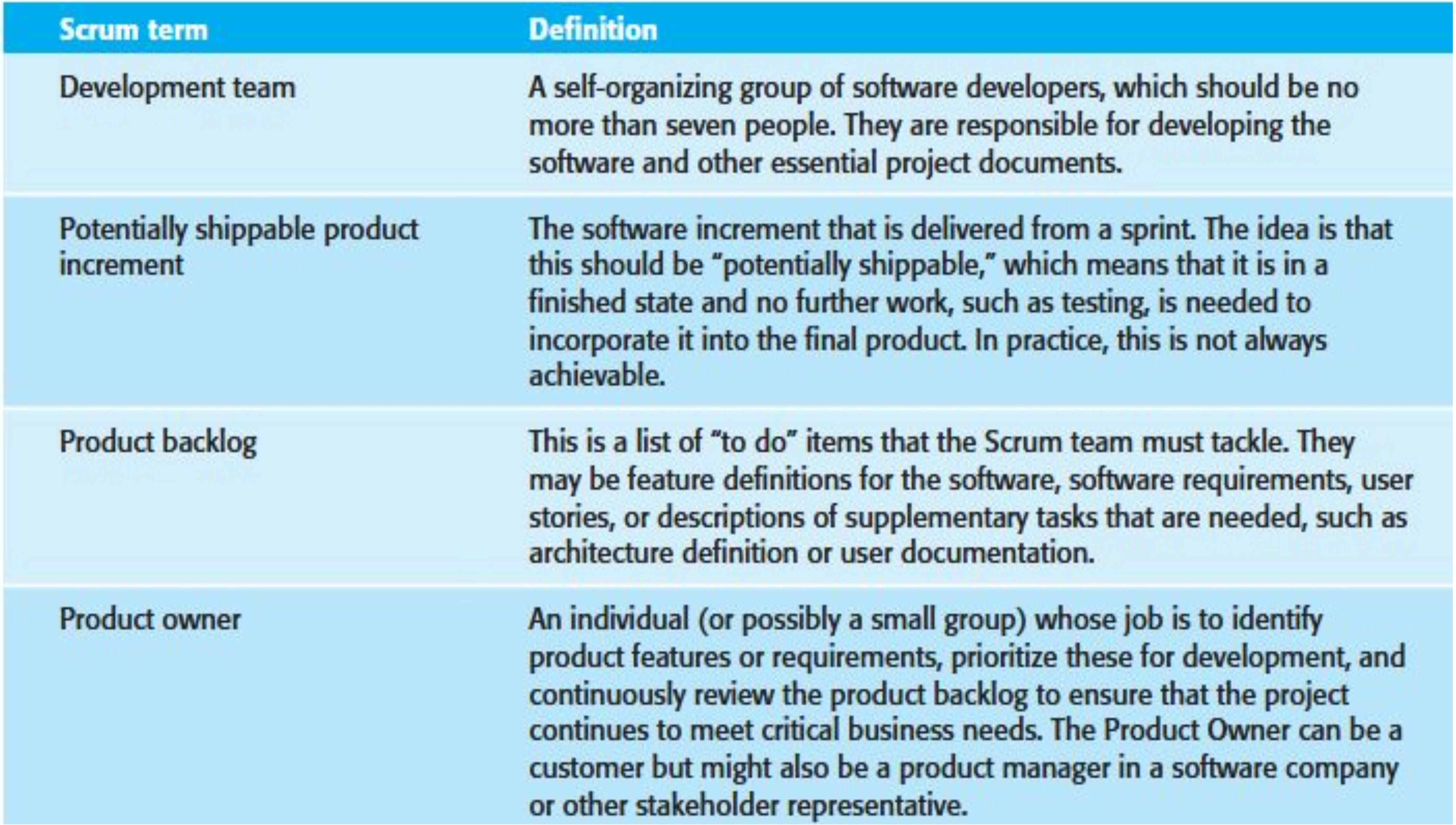
Fig 3.9 The scrum sprint cycle

1. **Review Work to Be Done**: The team and Product Owner review all pending features, enhancements, and bug fixes. These items are stored in the **Product Backlog** (a prioritized list of work).
2. **Select Items & Plan Sprint**: The team selects high-priority items from the product backlog to be completed in the next sprint. A **Sprint Backlog** is created, detailing tasks to be done during the sprint. Sprint planning ensures the scope is realistic and achievable.
3. **Sprint (Development Phase)**: The team works on implementing the selected backlog items. Daily **Scrum meetings** (15-minute stand-ups) track progress and address blockers. At the end of the sprint, the result is a **Potentially Shippable Software Increment**.
4. **Review Sprint**: The team demonstrates the completed work to stakeholders for feedback. Adjustments are made to priorities for the next sprint if needed.
5. **Cycle Repeats:** Feedback and new requirements are incorporated. The next sprint begins, ensuring continuous delivery and improvement.

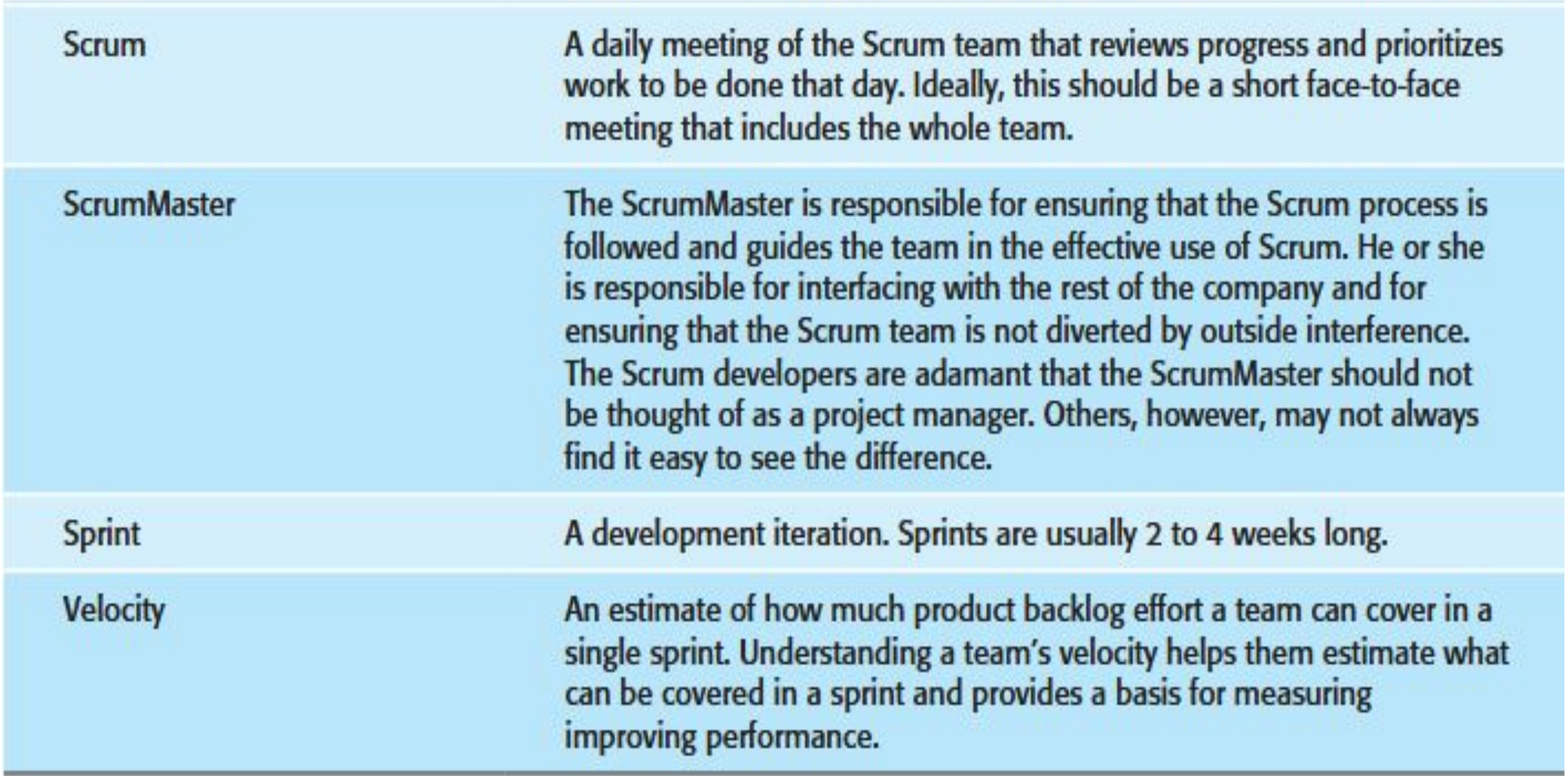
Key Outputs in the Diagram

- **Product Backlog:** Master list of all desired features.
- **Sprint Backlog:** Specific tasks for the current sprint.
- **Potentially Shippable Software:** A working increment ready for release or further enhancement.

- **Scrum** is an Agile framework used for managing and completing complex projects, particularly in software development. It focuses on delivering work in small, iterative cycles called **sprints**, encouraging collaboration, adaptability, and continuous improvement.
- **Scrum** adds structure and visibility to agile projects.
- Created to meet **management needs** for tracking progress.
- Uses roles like **ScrumMaster** instead of traditional managers.
- A **Scrum Master** is a key role in the Scrum framework responsible for ensuring that the team follows Agile principles and Scrum practices effectively. They act as a **facilitator, coach, and servant-leader** for the development team, Product Owner, and organization.
- Widely adopted, especially in **larger organizations**.



Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than seven people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be "potentially shippable," which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of "to do" items that the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories, or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development, and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.



Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2 to 4 weeks long.
Velocity	An estimate of how much product backlog effort a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

Figure 3.8 Scrum terminology

Scrum Process:

- **Input**: Product Backlog, A prioritized list of features, enhancements, and fixes to be completed.
- **Sprint**: A fixed-length iteration (1–4 weeks) where a potentially shippable product increment is developed.
- **Sprint Planning:** Meeting where the team selects backlog items for the sprint and defines how to complete them.
- **Sprint Backlog:** Subset of product backlog items chosen for the sprint, broken into actionable tasks.
- **Daily Scrum:** A short (15-minute) daily meeting to discuss progress, obstacles, and plans for the day.
- **Scrum Board:** A visual tool (physical or digital) to track sprint tasks—commonly showing columns like *To Do*, *In Progress*, and *Done*.
- **Sprint Review:** A meeting at the end of the sprint where the completed work is demonstrated to stakeholders for feedback.

ScrumMaster Role:

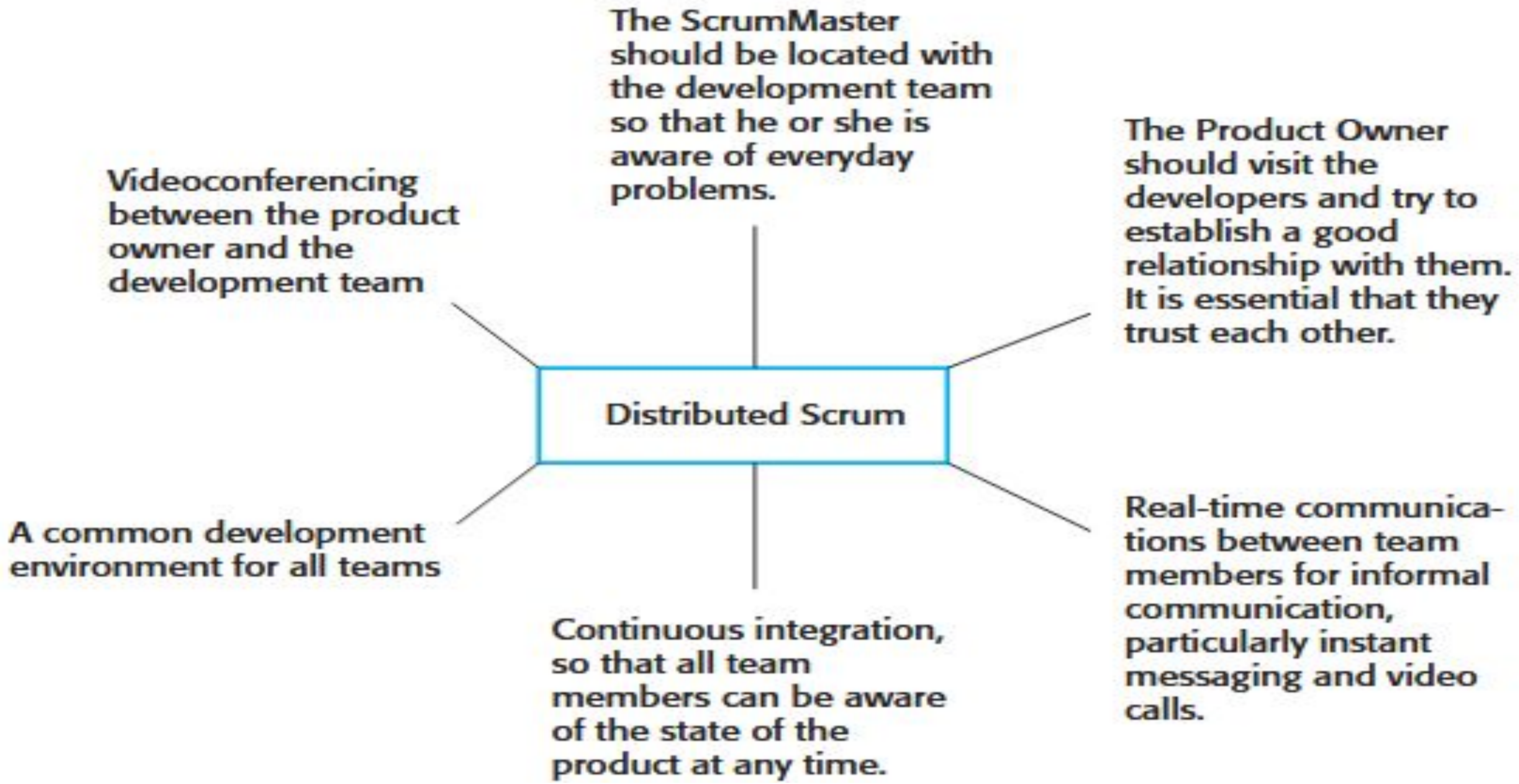
Guides the team, reports progress, handles logistics.

Benefits of Scrum:

- Work is broken into manageable parts.
- Handles changing requirements well.
- Clear progress and better teamwork
- Enables on-time delivery and early feedback.
- Builds trust between customers and developers.

Scrum has been adapted for **distributed teams** and offshore development, where the **product owner and developers** are in different locations.

Figure 3.10 shows the key requirements for **Distributed Scrum**.



Distributed Scrum

The ScrumMaster should be located with the development team so that he or she is aware of everyday problems.

The Product Owner should visit the developers and try to establish a good relationship with them. It is essential that they trust each other.

Real-time communications between team members for informal communication, particularly instant messaging and video calls.

Continuous integration, so that all team members can be aware of the state of the product at any time.

A common development environment for all teams

Videoconferencing between the product owner and the development team

THANK YOU

Software Process activities

Vaishnavi

Assistant Professor

Process Activities:

- Making software involves planning, building, and testing.
- People work together and use different tools.
- **Common tools include**: Planning Tools, Design tools, Code editors, Testing tools ,Debug tools.

Software specification:

- Requirements engineering defines what the system should do and its limits.
- It's a key step—early mistakes can cause big problems later.
- A quick feasibility study checks if the project is needed and possible.

What is Requirement Engineering?

Requirement Engineering is the process of defining, documenting, and maintaining the requirements of a software system throughout its lifecycle. It aims to capture what the system should do (functional requirements) and how it should perform (non-functional requirements).

Why is it Important?

- Prevents misunderstandings between stakeholders and developers.
- Reduces costly changes during later stages of development.
- Ensures the software satisfies both functional and non-functional needs.

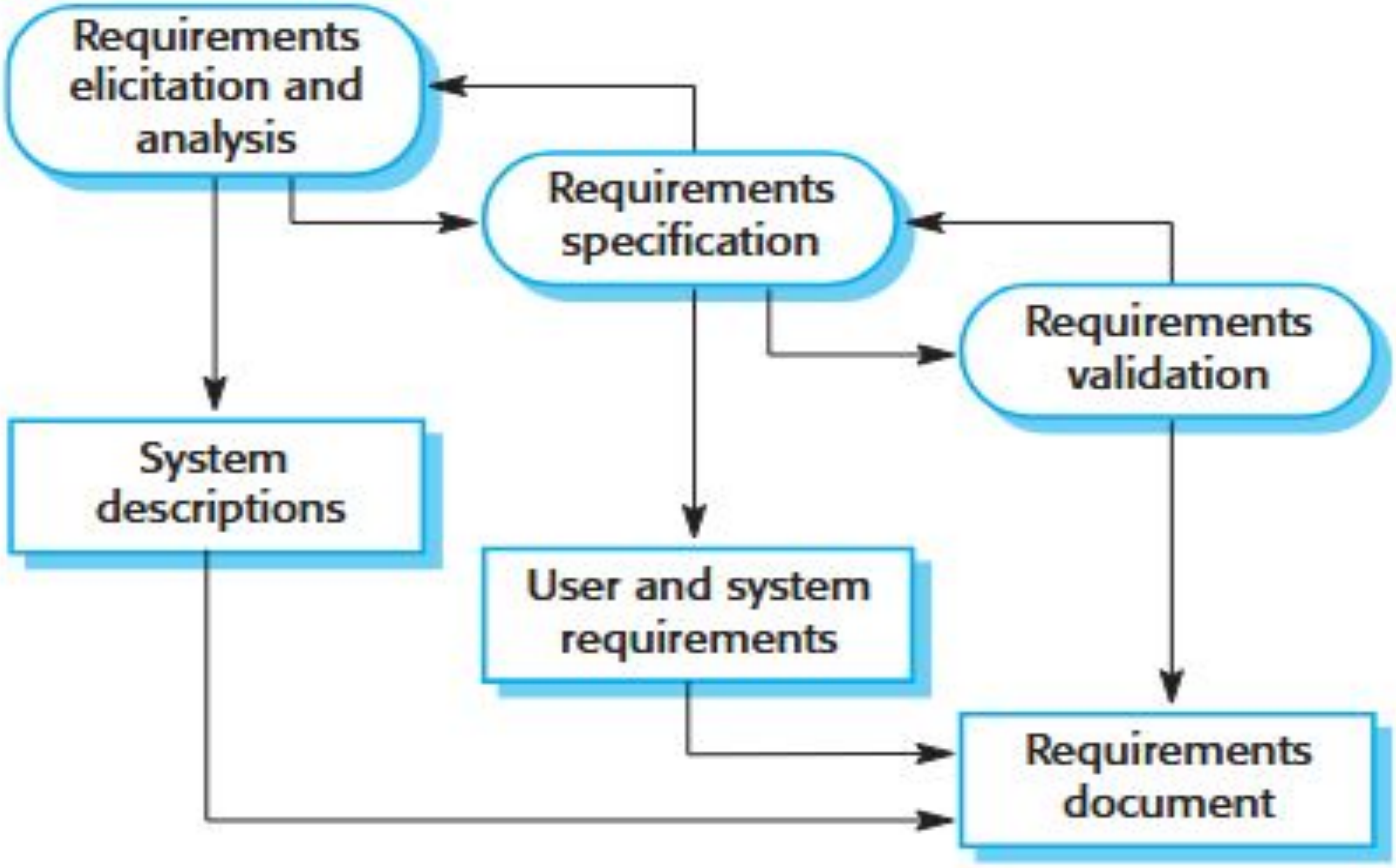

Requirements elicitation and analysis
Requirements specification
Requirements validation
System descriptions
User and system requirements
Requirements document

Figure 2.4 The requirements engineering process

Requirements engineering defines what the system should do.

- It includes:

1. High-level requirements for users
2. Detailed specifications for developers

Three main activities in requirements engineering:

1. **Elicitation & Analysis**
2. **Specification** – Write a document with:

   User requirements (simple statements for users).

   System requirements (detailed specs for developers).

3. **Validation**

- **In agile methods**:

  -Requirements are written just before each part is developed.

  -They're based on user priorities.

  -Users work closely with the development team.

Software design and implementation:

- **Software design** includes structure, data, interfaces, and algorithms.
- **Implementation** is building the working software for the customer.

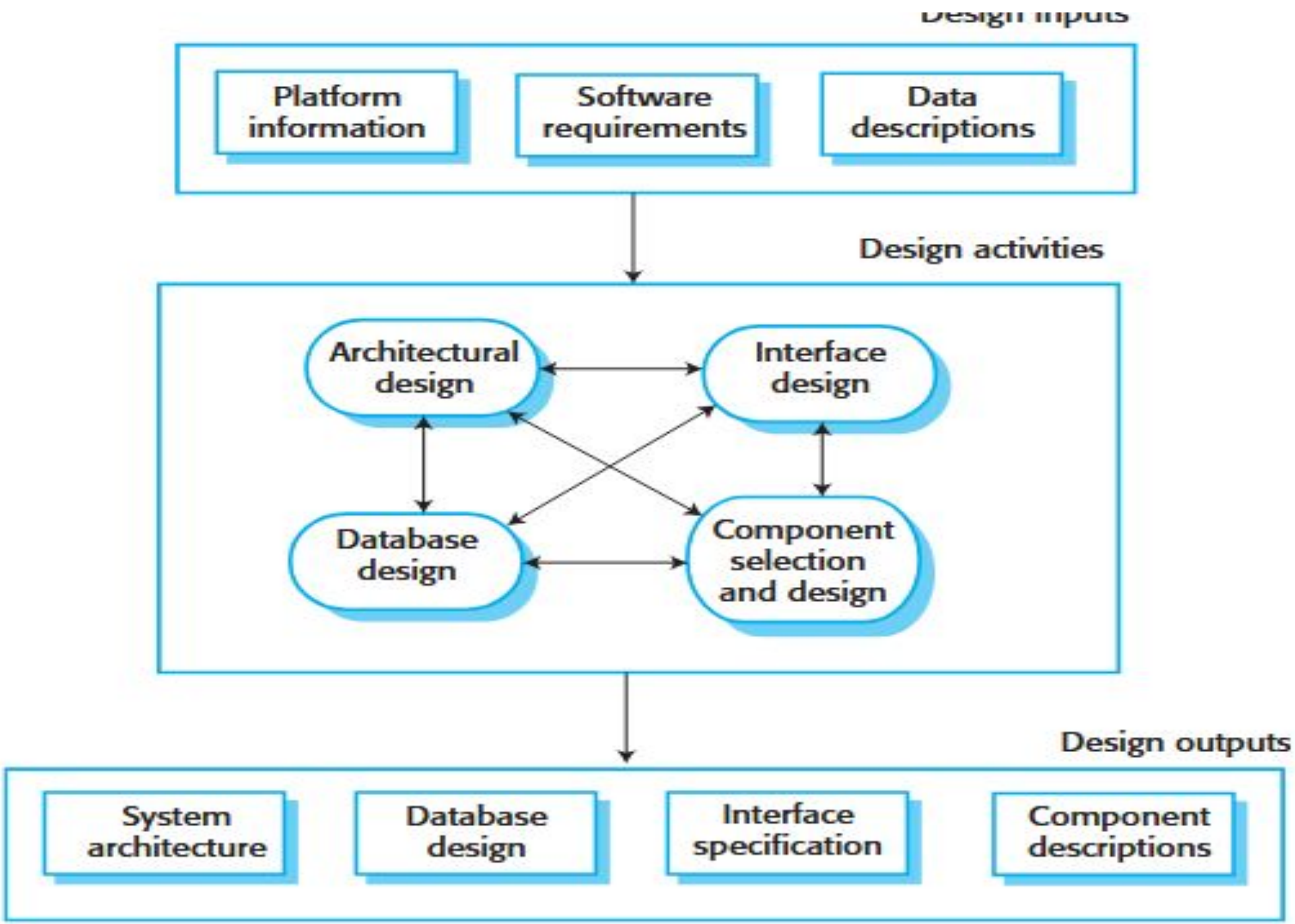


Figure 2.5 A general model of the design process

This diagram illustrates the software design process, breaking it down into three key sections: Design Inputs, Design Activities, and Design Outputs.

Design Inputs: These are the foundational elements that guide the design process:

- **Platform Information**: Details about the hardware, operating system, and environment where the software will run.
- **Software Requirements**: Functional and non-functional requirements that define what the software must do.
- **Data Descriptions**: Information about the data the system will handle—its structure, format, and flow.

These inputs ensure the design is grounded in real-world constraints and user needs.

Design Activities: This is the heart of the process, where actual design work happens. It includes four interconnected tasks:

- **Architectural Design**: Defines the overall structure of the system—how components interact and are organized.
- **Interface Design**: Specifies how users and other systems will interact with the software.

- **Database Design**: Structures the data storage, including tables, relationships, and access methods.
- **Component Selection and Design**: Chooses and defines the individual modules or components that will be built or reused.

These activities are interrelated, meaning decisions in one area (like architecture) influence others (like interface or database design).

Design Outputs: These are the tangible results of the design process, ready to guide development:

- **System Architecture**: A blueprint of the entire system's structure.
- **Database Design**: The finalized schema and data handling plan.
- **Interface Specification**: Detailed descriptions of user interfaces and APIs.
- **Component Descriptions**: Technical details about each software module.

This diagram shows how a structured and iterative approach to design ensures that the final software is well-planned, scalable, and aligned with user needs. It also highlights the importance of traceability where we can see how each output is directly tied to specific inputs and activities.

Software Validation and Verification

- **verification and validation (V & V)** checks if the software meets its specs and user needs.
- **Verification**: It checks if the software is built correctly according to specifications/design. It is about process quality. Its focus is does the system meet the written requirements during development (before execution). Methods includes Reviews (code reviews, design reviews), Walkthroughs, Inspections, Static analysis (checking code without running it). Example: Specification says *"Password must be at least 8 characters."* .Verification checks if the code enforces this rule.
- **Validation:** It checks if the final software meets the customer's real needs. It is about product quality. Its focus is does the system do what the user actually wants after or during development (with execution). Methods includes Testing (unit, integration, system, acceptance), Prototyping, User feedback sessions. Example: Customer wants secure login → Validation ensures the login process truly satisfies security needs.

- The main method is **program testing** with test data.
- It can also include **reviews and inspections** during development.
- Most time is spent on testing.

Process of V&V in Software Engineering (draw diagram Fig 2.7 and explain)

1. **Requirements Verification** → Check if requirements are clear, complete, consistent.

2. **Design Verification** → Check design matches requirements.

3. **Code Verification** → Static checks, code reviews.

4. **Validation Testing** → Unit, integration, system, acceptance testing.

5. **Final Validation** → Ensure delivered product meets user needs in real environment.

- Testing is **iterative**—bugs found later may require earlier tests to be repeated.
- **Component testing** is usually done by programmers during development.
- In **incremental development**, each part is tested as it's built.
- In **test-driven development** (agile), tests are written before coding to clarify requirements and avoid delays.

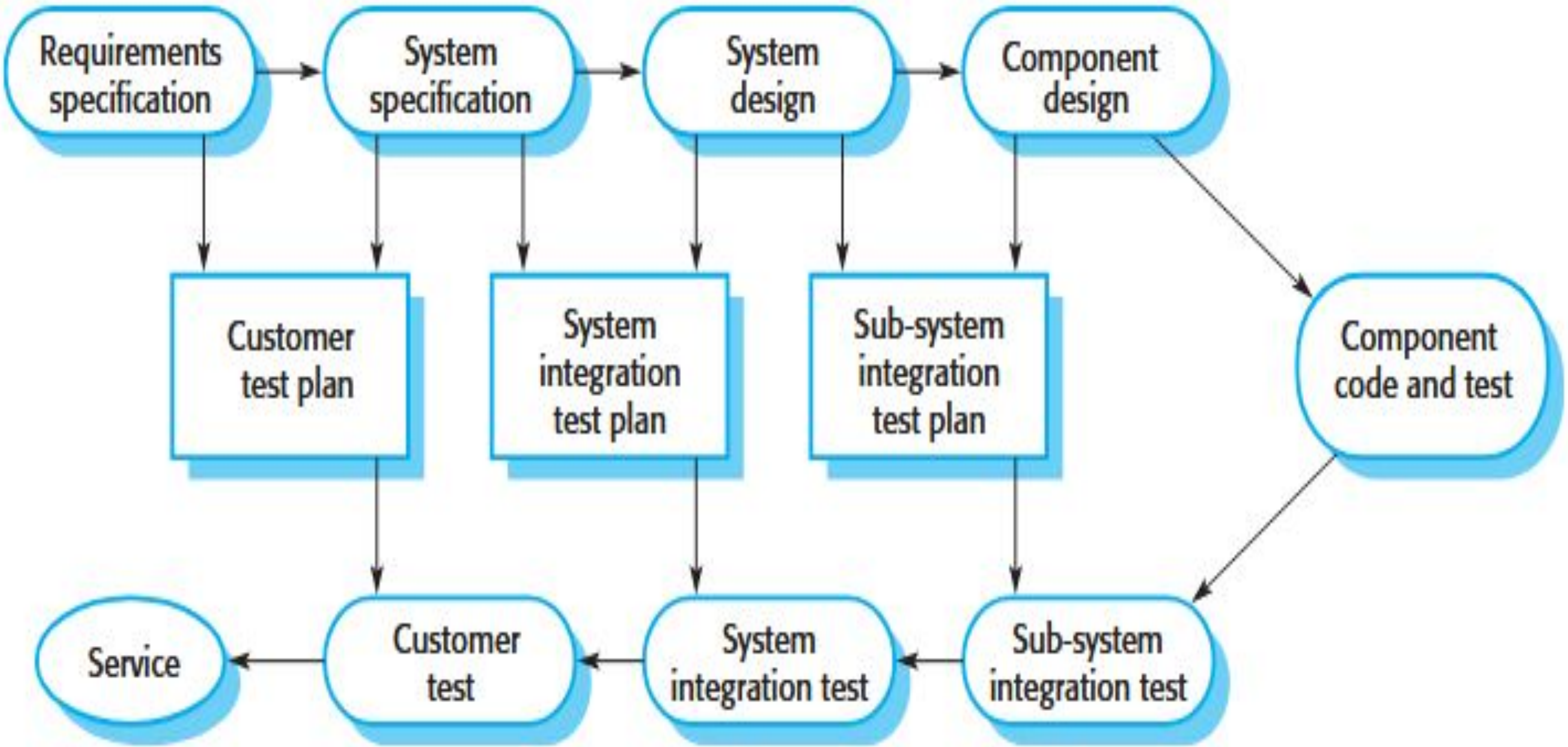


Figure 2.7 Testing phases in a plan-driven software process

This diagram represents the V-model of software Development(Verification and Validation model). It shows how the software is developed and tested step by step, with each development phase having a corresponding testing phase.

Left Side of the V: Development Phases

- These are the planning and design stages where the system is conceptualized and architected.
- **Requirements Specification**: Understanding what the customer wants. This forms the basis for all future development.
- **System Specification**: Translating customer needs into detailed system-level requirements.
- **System Design**: Designing the architecture of the system—how components will interact.
- **Component Design**: Designing individual modules or components that will be coded.

Each of these phases leads to a corresponding **test plan**:

- *Customer Test Plan*: Ensures the final product meets customer expectations.
- *System Integration Test Plan*: Validates that the entire system works together.
- *Sub-system Integration Test Plan*: Tests how smaller subsystems interact.

Right Side of the V: Testing Phases

- This side mirrors the development stages, but focuses on testing and validation.
- **Component Code and Test**: Developers write code and test individual components.
- **Sub-system Integration Test**: Tests how components work together within subsystems.
- **System Integration Test**: Ensures all subsystems integrate correctly.
- **Customer Test**: Validates the system against the original requirements.
- **Service**: Ongoing support and maintenance after deployment.

The V-shape shows how each development activity has a corresponding test activity:

- Requirements ↔ Customer Test
- System Specification ↔ System Integration Test
- System Design ↔ Sub-system Integration Test
- Component Design ↔ Component Test
- This structure reinforces the idea that testing isn’t an afterthought—it’s planned from the beginning.

- In **plan-driven** development (e.g., for critical systems), testing follows detailed test plans made from specifications and design.
- The **V-model** shows how each development stage has a matching test stage.
- **Beta testing** is used for products—real users try the software and report issues, helping developers fix problems before final release.

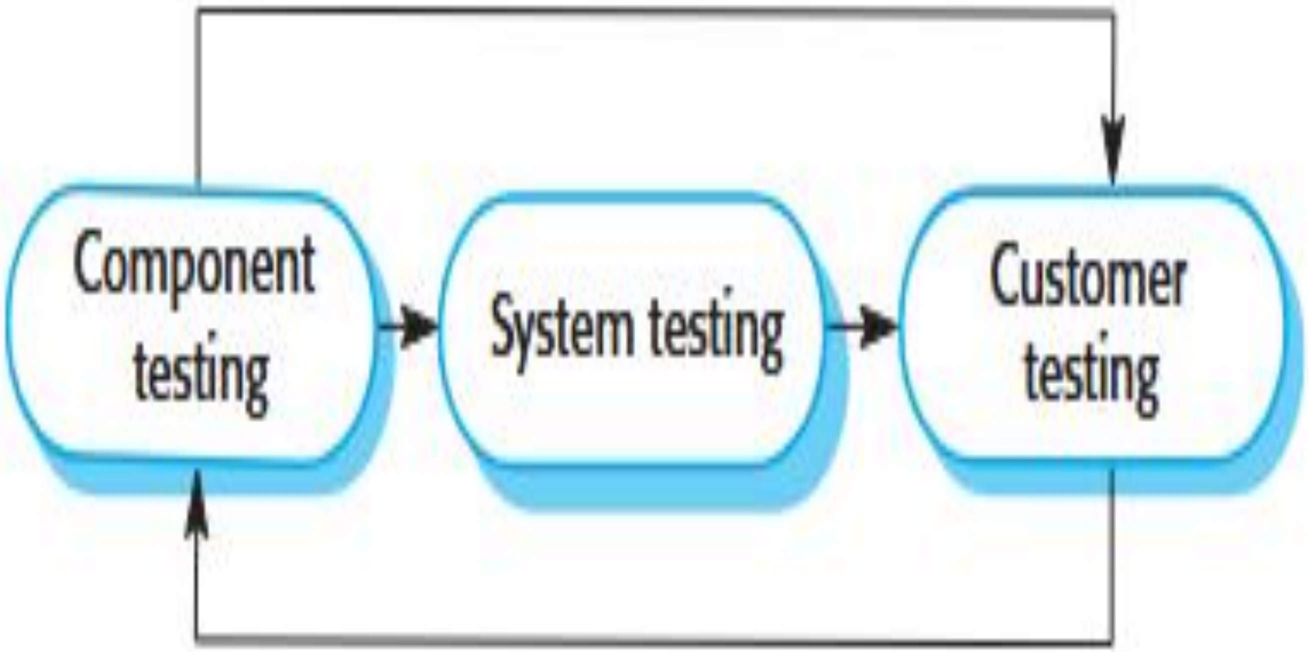


Figure 2.6 Stages of testing

The stages in the testing process are:

1. **Component Testing** – Each part is tested alone by developers using tools like JUnit.
2. **System Testing** – All parts are combined and tested for interaction issues and requirements.
3. **Customer Testing** – Real users test the system with real data to find missing or unclear requirements.

Software evolution:

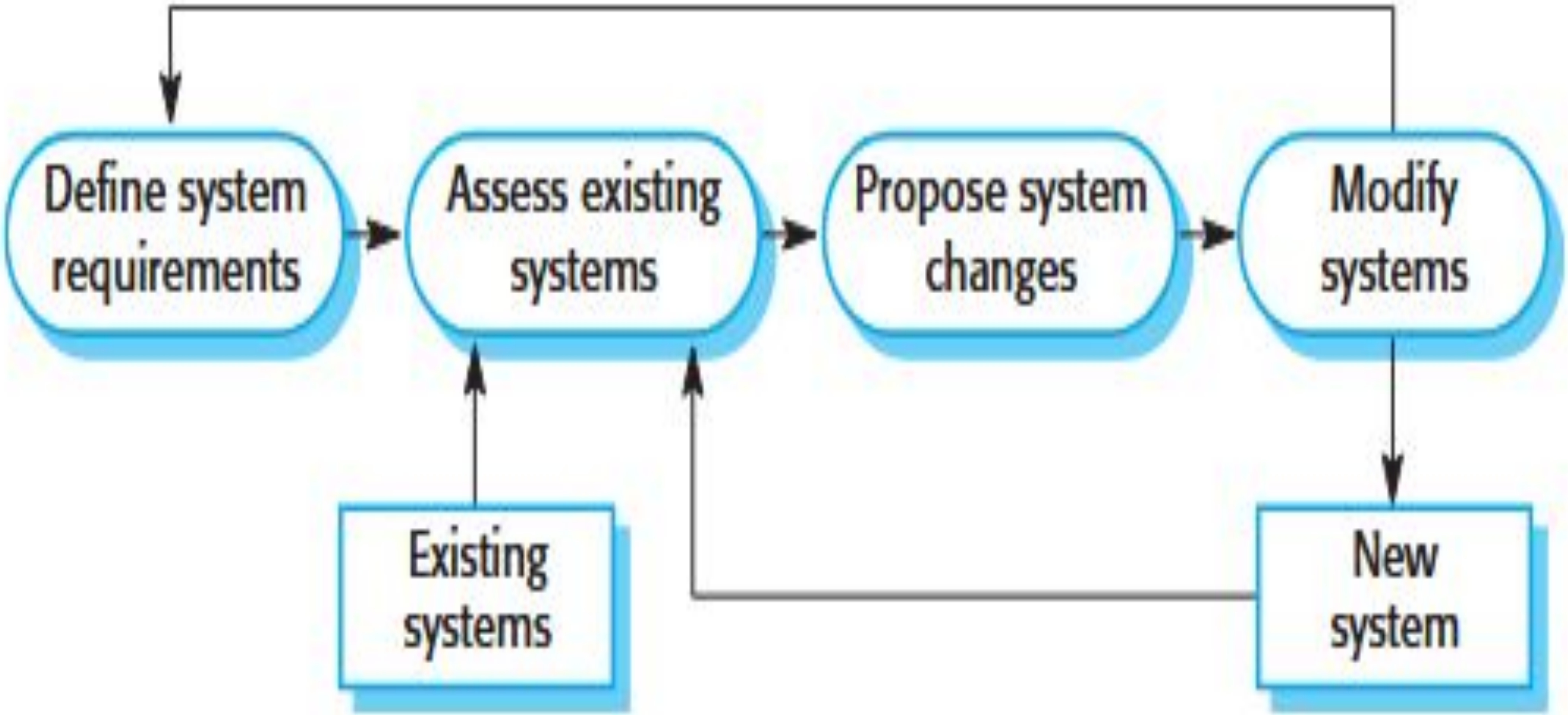


Figure 2.8 Software system evolution

This diagram represents a System Development or Improvement Process.

1. Define System Requirements

- Identify what the system needs to do.
- Gather information about user needs, business goals, and technical requirements.

2. Assess Existing Systems

- Review the current system(s) in place.
- Find out what works well and what needs improvement.

3. Propose System Changes

- Suggest ways to improve or upgrade the system.
- These changes should fix issues or add new features based on requirements.

4. Modify Systems

- Implement the proposed changes.
- This can include updating the existing system or creating a new one.

5. Existing Systems & New System

- **Existing systems** feed into the assessment stage for evaluation.
- **New system** is created or updated after modifications and may later become the "existing system" for future improvements.

Cycle Nature: It’s a continuous process, once a new system is created, it can be reassessed in the future for more improvements.

- **Software is flexible**, making changes easier and cheaper than hardware.
- **Development** and **maintenance** were once seen as separate.
- Now, they're seen as one **continuous, evolving process**, as most systems are updated over time to meet new needs.

Thank You

Requirements Analysis

Vaishnavi
Assistant
Professor

- **Requirements** describe the services a system must provide and its constraints, reflecting customer needs.

- **Requirements Engineering (RE)** is the process of discovering, analyzing, documenting, and validating these requirements.

- The term *requirement* varies:

- User requirements.

- System requirements.

- **Stakeholders** include anyone affected by the system, such as:
  - Patients and their families
  - Doctors and nurses
  - Medical receptionists and IT staff
  - Ethics managers and health care managers
  - Medical records staff

4.1 - Functional and non-functional requirements

Software system requirements are often classified as:

1. **Functional requirements**

2. **Non-functional requirements**

4.1.1 - Functional requirements:

- Define **what** the system should do (functions, inputs, outputs).
- Written for users in **clear, natural language**; detailed for developers.
- Ambiguity can lead to **misunderstandings**, delays, and higher costs.
- Large systems may have **inconsistent** or **incomplete** requirements.

4.1.2 - Non-functional requirements:

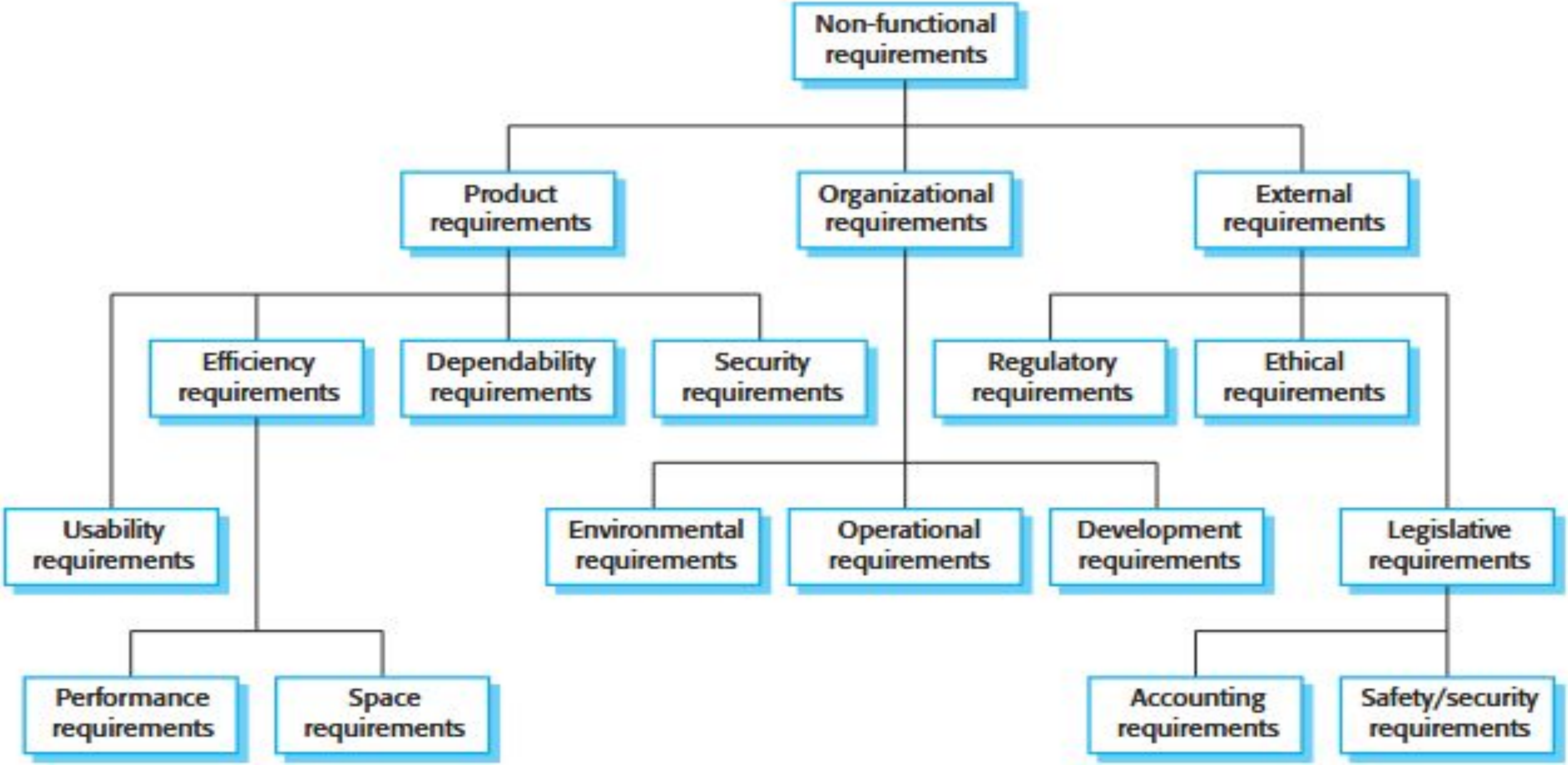
- **Non-functional requirements** focus on system characteristics (e.g., reliability, performance, security) rather than specific services.
- They may affect the **overall system architecture**
- Sources of non-functional requirements:

 - **Product requirements**: Performance, reliability, security, usability.

 - **Organizational requirements**: Policies, development standards, environment.

 - **External requirements**: Regulatory, legislative, and ethical constraints.

Fig 4.3 Types of non-functional requirements



The fig 4.3 above, offers a structured breakdown of **non-functional requirements (NFRs)**—the qualities and constraints that define how a system performs rather than what it does.

Main Categories of Non-Functional Requirements:

1. Product Requirements These relate directly to the system's behavior and performance from a technical standpoint.

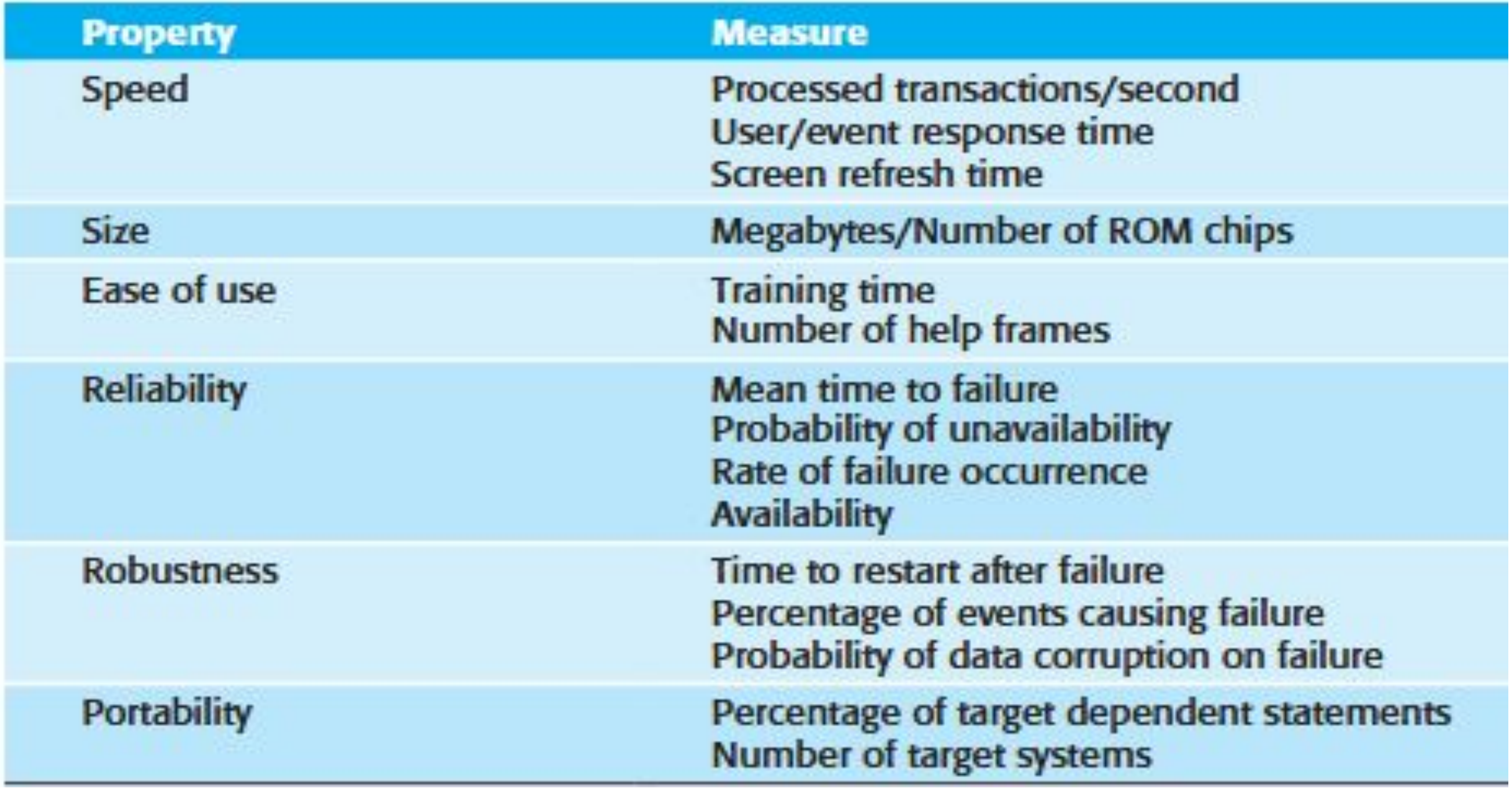
- **Efficiency**: How well the system uses resources like CPU, memory, and bandwidth.
- **Dependability**: Includes reliability, availability, and fault tolerance.
- **Security**: Protection against unauthorized access, data breaches, and vulnerabilities.
- **Usability**: Ease of use, user interface design, and user experience.
- **Performance**: Speed, response time, throughput under various conditions.
- **Space**: Physical or digital space constraints (e.g., memory footprint, storage)

2. Organizational Requirements These are shaped by the company’s internal policies, processes, and infrastructure.

- **Environmental**: Compatibility with existing hardware/software environments.
- **Operational**: Requirements for system operation, including maintenance and support.
- **Development**: Constraints on development tools, languages, or methodologies.
- **Accounting**: Budgetary constraints, cost tracking, and financial reporting.

3. External Requirements These come from outside the organization and often involve compliance and ethics.

- **Regulatory**: Adherence to industry standards and government regulations.
- **Ethical**: Ensuring the system aligns with moral principles (e.g., fairness, transparency).
- **Legislative**: Legal obligations like data protection laws (e.g., GDPR).
- **Safety/Security**: Ensuring the system does not pose risks to users or the environment.



Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Megabytes/Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Figure 4.5 Metrics for specifying non-functional requirements

4.2 - Requirements engineering Processes

- **Three key activities**:

1. **Elicitation & Analysis** – Discovering requirements with stakeholders
2. **Specification** – Documenting them in standard form
3. **Validation** – Ensuring the system meets customer needs.

- **Iterative process**: Not strictly sequential—activities repeat and overlap in a **spiral model**.
- **Early stages**: Focus on business, user, and high-level non-functional requirements.
- **Later stages**: Focus shifts to detailed system requirements.
- **Supports agile**: Requirements and system development can happen together.
- **Changing needs**: As requirements evolve, managing changes is important to understand the impact on the system.

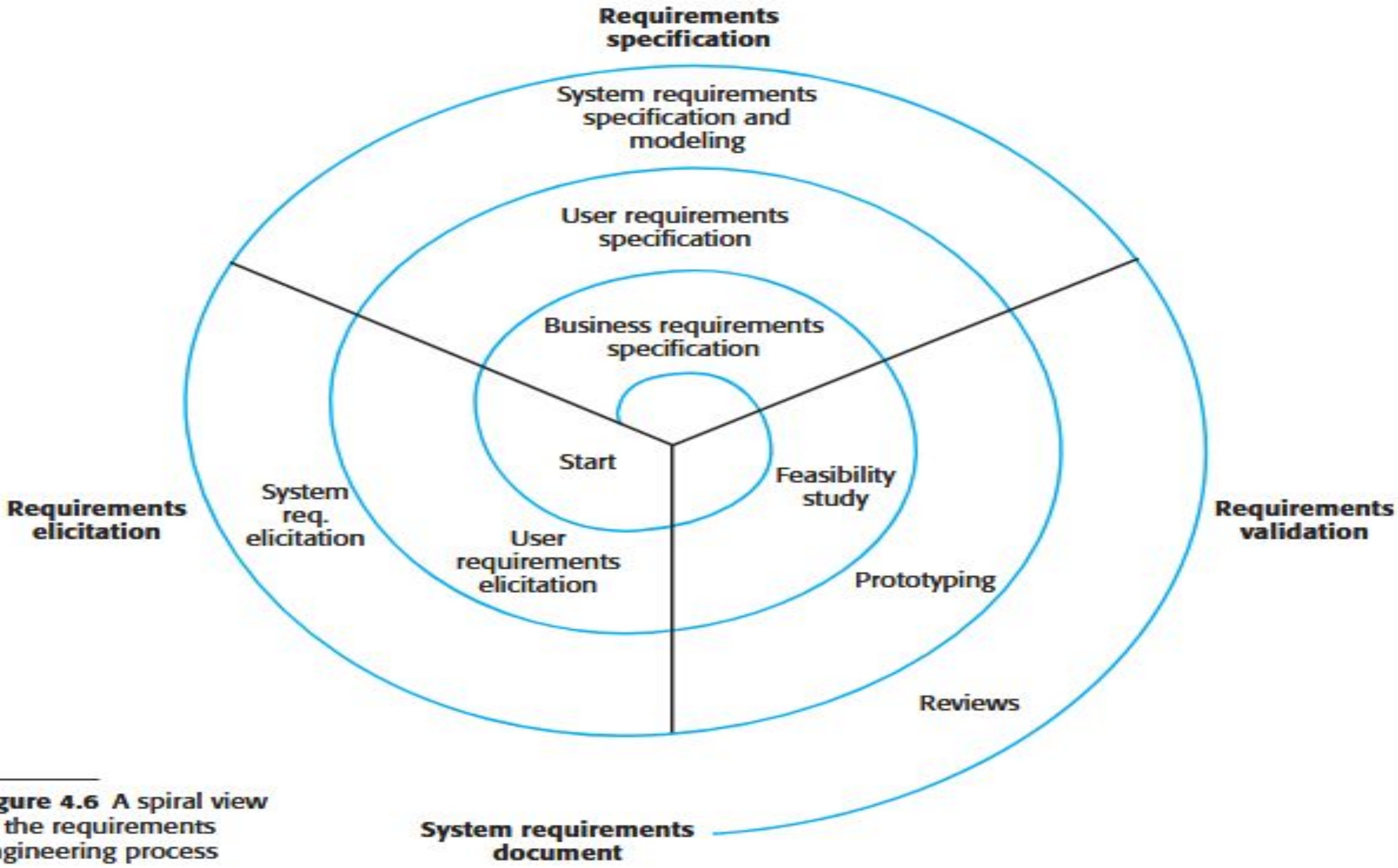

Requirements specification
System requirements specification and modeling
User requirements specification
Business requirements specification
Start
Feasibility study
Requirements elicitation
System req. elicitation
User requirements elicitation
Prototyping
Requirements validation
Reviews
System requirements document

Figure 4.6 A spiral view of the requirements engineering process

The above diagram is called **“A Spiral View of the Requirements Engineering Process”**. It represents how requirements are gathered, refined, validated, and documented in a **cyclical and iterative** manner. Let’s break it down:

1. Spiral Structure

- The process starts at the **center** (Start).
- Each loop of the spiral represents a **stage of requirements engineering**.
- As you move outward, requirements become more detailed and complete.
- It emphasizes that requirements engineering is **not linear**, but **iterative** — we revisit steps multiple times to refine requirements.

2. Three Main Activities

- The spiral is divided into three main sectors:

i. **Requirements Elicitation (Left side)**

- This is about **gathering requirements** from stakeholders.
- Stages include:
    - User requirements elicitation: Collecting needs from end-users.
    - System requirements elicitation: Identifying system-level requirements

ii. Requirements Specification (Top side)

- Involves **documenting and modeling requirements** clearly.
- Stages include:
    - Business requirements specification: High-level business goals.
    - User requirements specification: What the users expect from the system.
    - System requirements specification and modeling: Technical and functional details of the system.

iii. Requirements Validation (Right side)

- Ensures requirements are **correct, complete, and feasible**.
- Stages include:
    - Feasibility study: Can the requirements be realistically implemented?
    - Prototyping: Building early versions to validate user needs.
    - Reviews: Checking documents and prototypes with stakeholders.

3. Output

- At the outer edge of the spiral, we end up with the **System Requirements Document (SRD)**.
- This document serves as a formal reference for designers, developers, and testers.

4.3 - Requirement elicitation

- **Goal**: Understand stakeholders’ work and how a new system can support it.

Challenges:

- Stakeholders may not know exactly what they want.
- Language and knowledge gaps between stakeholders and engineers.
- Conflicting requirements from different stakeholders.
- Political influences on system demands.
- Changing requirements due to new stakeholders or circumstances.

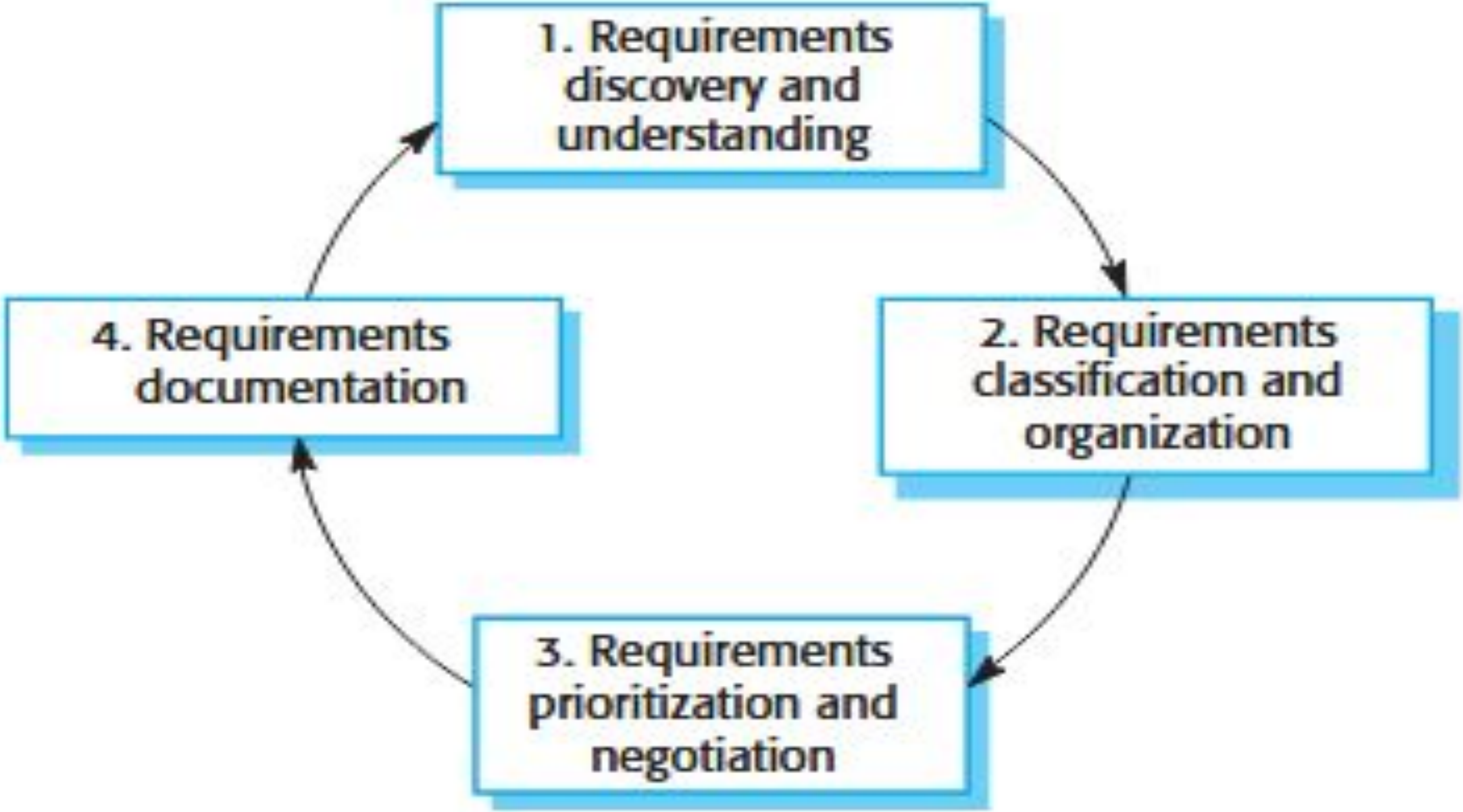

1. Requirements discovery and understanding
2. Requirements classification and organization
3. Requirements prioritization and negotiation
4. Requirements documentation

Figure 4.7 The requirements elicitation and analysis process

The diagram 4.7, works as a **visual representation of the continuous process** used in requirement elicitation and analysis. It’s designed to show how each stage feeds into the next—and how the cycle repeats to refine and improve the requirements over time.

1. Requirements discovery and understanding

- Talk to stakeholders.
- Learn what they do and what they need.

2. Requirements classification and organization

- Group similar requirements together.
- Organize them (e.g., user needs, business needs, system features).

3. Requirements prioritization and negotiation

- Decide which requirements are most important.
- Solve conflicts when people want different things.
- Agree on what to do first.

4. Requirements documentation

- Write down the requirements clearly.
- Create a document everyone can understand and use.
- And also this document will become as an input to the next cycle.

Why a cycle?

Because requirements often change or new needs are found. If something is unclear at one step, you go back and improve earlier steps.

- The process is **iterative**, with continual feedback and improvement.
- Use **viewpoints** to organize requirements by stakeholder group or subsystem.
- Regular stakeholder meetings help **resolve conflicts** and ensure all concerns are addressed.
- **Simple language and diagrams** are crucial for clear documentation and easy feedback.

4.3.1 - Requirements elicitation techniques

- Requirements elicitation involves gathering information about a proposed system by engaging with stakeholders
- Involves understanding users’ work, outputs, and current system use.
- Two main approaches:

  1. **Interviewing** – talking to stakeholders about their roles and needs.
  2. **Observation (Ethnography)** – watching users perform tasks to understand workflows and tools.

- Combining both methods helps identify accurate system requirements.
- These requirements form the basis for further discussions and system design.

Interviewing

- Interviews are a common part of requirements engineering, used to gather information from stakeholders.

Types of interviews:

1. **Closed interviews** – predefined questions.
2. **Open interviews** – flexible discussion without a fixed structure.

In practice, interviews usually combine both types.

- Interviews help:

- Understand stakeholder roles and interactions with current and future systems.
- Identify problems with existing systems.

Challenges:

- Stakeholders use domain-specific jargon that can be misunderstood.
- Some knowledge is so familiar it's not mentioned.
- Difficult to uncover organizational politics or real decision-making structures.

Effective interviewing tips:

- Stay open-minded and ready to adjust your understanding.
- Use prompts (e.g., prototypes or specific questions) to guide discussion.
- Avoid vague questions like “tell me what you want.”

Ethnography

- Software systems operate within social and organizational environments, which influence their requirements.
- Failing to consider these factors often leads to systems that are delivered but not used.

What is Ethnography?

- An observational technique where analysts immerse themselves in the user’s work environment.
- Helps uncover **implicit requirements** based on how work is actually done—not just how it's documented.

Benefits:

- Reveals unspoken, routine practices users may not mention in interviews.
- Identifies cooperative behaviors and real workflows (e.g., air traffic controllers adapting processes).
- Can be combined with prototyping to refine system design more efficiently.

Limitations:

- Focuses on existing practices—not ideal for discovering innovations or high-level organizational/domain requirements.
- Best used alongside other elicitation techniques for a complete view.

Examples:

- ATC systems: Ethnography exposed the need for visibility into adjacent sectors, not noted in formal procedures.
- Nokia used ethnography to refine products, while Apple innovated by not relying on current use.

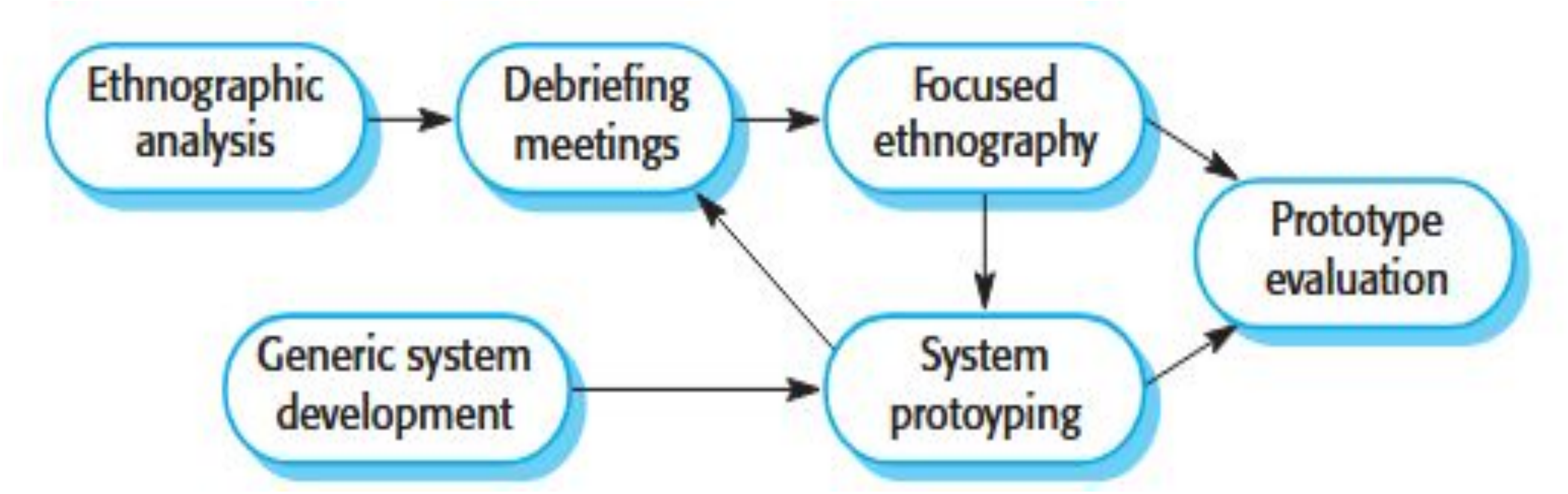


Fig 4.8 Ethnography and prototyping for requirements analysis

4.3.2 - Stories and scenarios

- People relate better to real-life examples than to abstract descriptions.
- Stakeholders may not specify requirements directly but can describe how they handle situations or imagine new workflows.

What Are Stories and Scenarios?

- **Stories**: High-level, narrative descriptions of how the system might be used.
- **Scenarios**: More structured, detailed descriptions including inputs, outputs, and system interactions.

Purpose:

- Help stakeholders visualize system use.
- Support discussions during interviews.
- Aid in identifying and refining specific system requirements.

Use:

- Start with stories to present the "big picture.“
- Break stories into detailed scenarios for deeper analysis.

- **Stories** are easy to relate to and help gather input from a broader audience.
- Sharing stories (e.g., via a wiki) encourages feedback from users like teachers and students.
- Stories give a high-level view, while **scenarios** add detail and structure.

Scenarios in Detail:

- Describe specific user-system interactions.
- Best presented in a structured format, not just as narratives.
- Often used in **Agile** (e.g., user stories in Extreme Programming).

Typical Scenario Structure:

1. Initial system/user conditions
2. Normal sequence of events
3. Possible errors and how they’re handled
4. Concurrent activities
5. Final system state

4.4 - Requirement specification

- **Requirements specification** involves documenting user and system requirements.

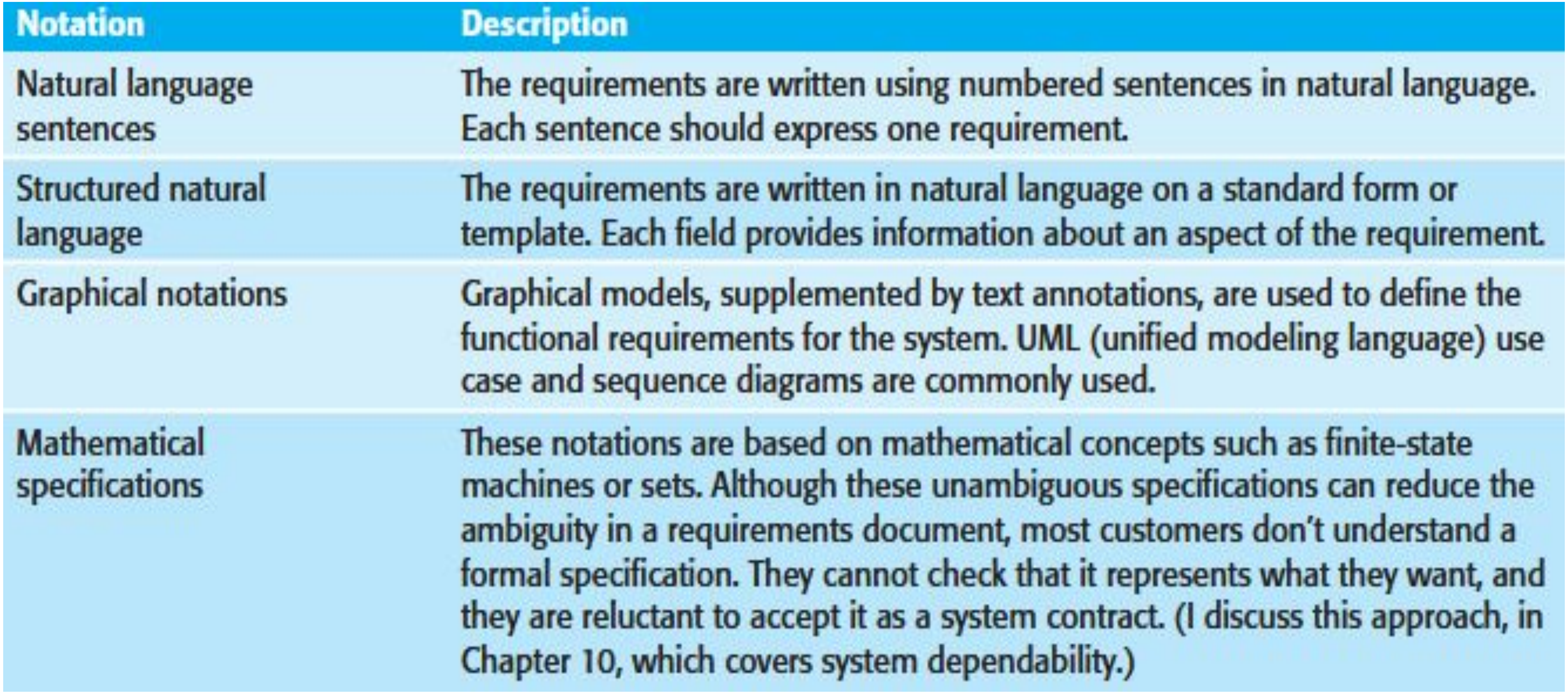
- These should be **clear, unambiguous, complete, and consistent**, though this is hard to fully achieve.

User Requirements:

- Written in **natural language**, with simple **tables and diagrams**.
- Should focus on **functional** and **non-functional** needs.
- Must be understandable to non-technical users.
- Should describe **what the system should do**, not **how** it does it.

System Requirements:

- May use **natural language**, **forms**, **graphical**, or **mathematical models**.
- Can be more technical and detailed than user requirements.



Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system. UML (unified modeling language) use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want, and they are reluctant to accept it as a system contract. (I discuss this approach, in Chapter 10, which covers system dependability.)

Fig 4.11 – Notations for writing system requirements

- **System requirements** are detailed versions of user requirements, used for system design and contracts.
- They describe **external behavior** and **operational constraints**, but some design details are often included.

Why Design Info May Be Included:

- To structure requirements around **system architecture** or subsystems.
- To meet **integration needs** with existing systems.
- To fulfill **non-functional requirements** (e.g., safety, reliability) that require specific architectural choices.

Natural language specification

- It is widely used for writing requirements due to its simplicity, but it can be **vague** or **ambiguous**.

Guidelines to Improve Clarity:

- **Use a standard format** for all requirements (1–2 clear sentences).
- Use **"shall"** for mandatory and **"should"** for desirable requirements.
- **Highlight key text** (bold, italic, color) for clarity.
- **Avoid technical jargon** and unclear abbreviations.
- Include a **rationale** for each requirement—why it's needed and who requested it.

Structured specifications

- It uses templates to write requirements in a consistent, organized way—more precise than free-form text but still readable.
- Often uses forms, standardized fields, and formatting (e.g., fonts, shading) to highlight key elements.

Template Fields Typically Include:

1. Function or entity description
2. Inputs and their sources
3. Outputs and their destinations
4. Required data or system components
5. Actions taken
6. Preconditions and post conditions (if functional)
7. Side effects (if any)

Benefits:

- Reduces ambiguity and improves consistency.
- Helps organize complex requirements clearly.
- Still may require **tables, diagrams, or models** to clarify complex logic or computations.

Example Method:

- **VOLERE** uses requirement cards with fields like rationale, dependencies, and source.

Use cases

- It describe interactions between users (actors) and a system using **diagrams and structured text**.
- Part of **UML** (Unified Modeling Language), originally from the **Objectory method**.

Components:

- **Actors** (users or other systems) – shown as stick figures.
- **Use cases** (interactions) – shown as ellipses.
- Lines connect actors to use cases; arrows can show who initiates the interaction.

Use Case Example:

Setup Consultation allows multiple doctors to view a patient record together, with only one able to edit, plus a chat window for coordination.

- **Pros:** Good for modeling system interactions; useful in **system design**.
- **Cons:** Often **too detailed for early requirements discussions**; non-technical stakeholders may not understand them.

Scenarios:

- Each use case can include **multiple scenarios**—normal and exception flows.

The software requirements document

- The **Software Requirements Document (SRS)** defines what developers should implement.
- It may include both **user** and **system requirements**, either combined or in separate sections.

When SRS Is Essential:

- In **outsourced projects**
- When **different teams** handle different parts
- Where **formal analysis** is required

Agile Perspective:

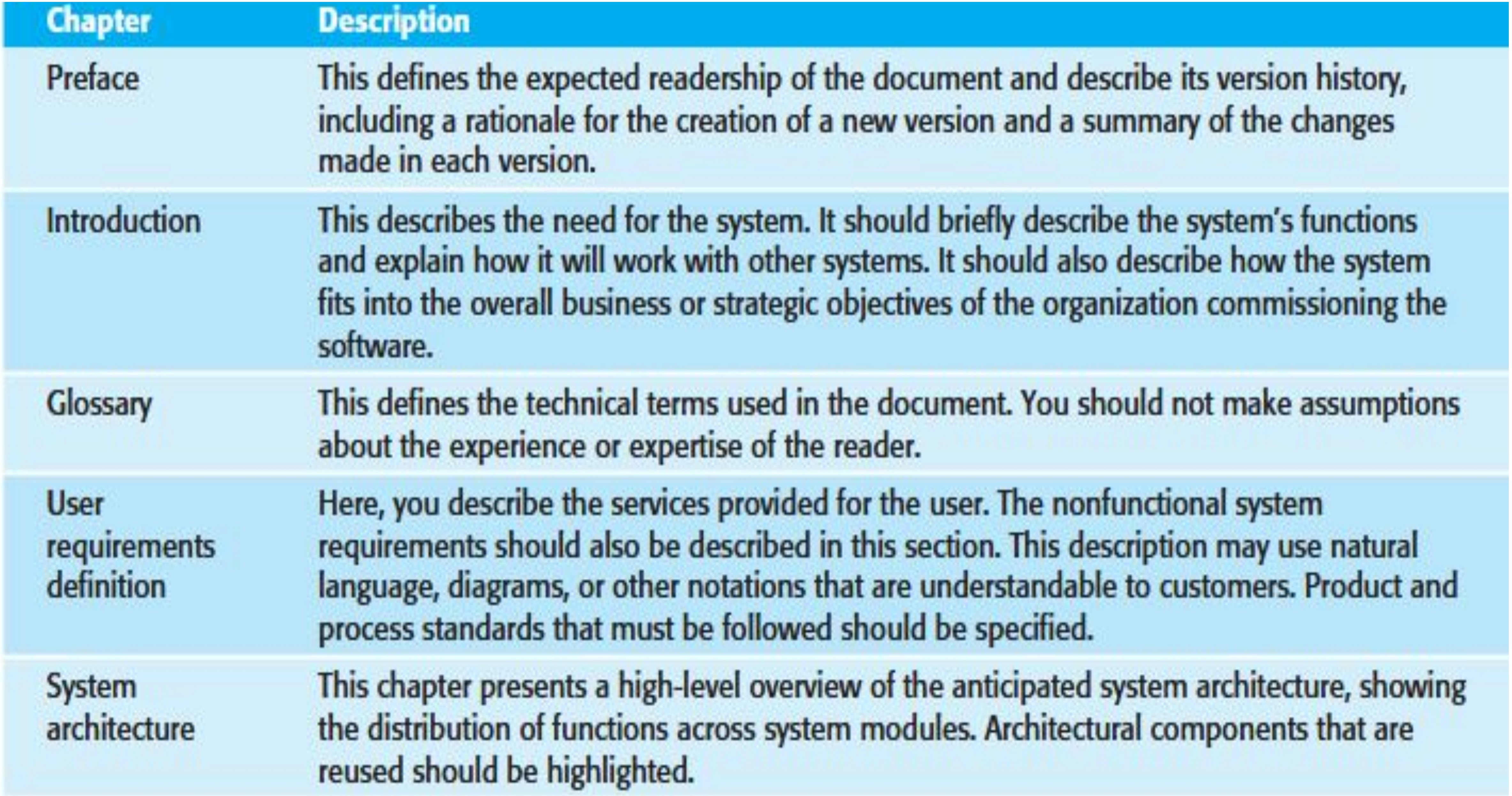
- Agile methods often **avoid detailed SRS**, using short, evolving **user stories** instead.
- However, it's still helpful to document key **business** and **dependability requirements**.

Users of the SRS:

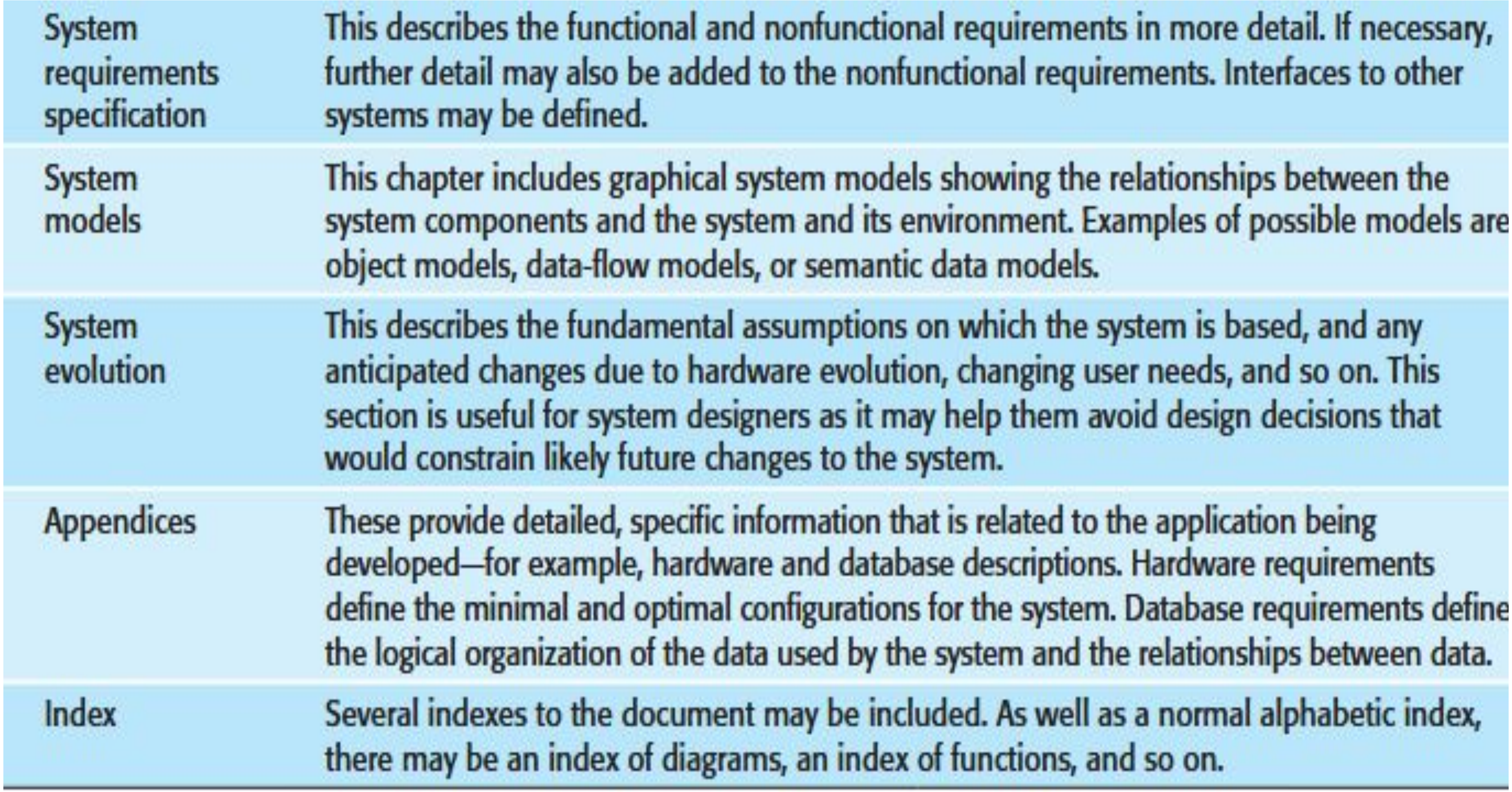
- Includes **management, developers, testers, maintainers**, and **system designers**.
- Must balance being understandable for customers and detailed for technical staff.

Detail Level Depends On:

- **Critical systems** require detailed, precise specs.
- **Outsourced projects** need clear specifications.
- **In-house or iterative development** can use less detailed, evolving documents.



Chapter	Description
Preface	This defines the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This describes the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This defines the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter presents a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.



System requirements specification	This describes the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This chapter includes graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This describes the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These provide detailed, specific information that is related to the application being developed—for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Fig 4.17 – The structure of a requirements document

4.5 - Requirement Validation

Requirements validation is the process of checking that requirements define the system that the customer really wants.

Different types of checks:

1**. Validity checks:** These check that the requirements reflect the real needs of system users. Because of changing circumstances, the user requirements may have changed since they were originally elicited.

2. **Consistency checks:** Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.

3. **Completeness checks** : The requirements document should include requirements that define all functions and the constraints intended by the system user.

4. **Realism checks :** By using knowledge of existing technologies, the requirements should be checked to ensure that they can be implemented within the proposed budget for the system. These checks should also take account of the budget and schedule for the system development.

5. **Verifiability :** To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

Validation techniques :

1. **Requirements reviews :** The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.

2. **Prototyping :** This involves developing an executable model of a system and using this with end-users and customers to see if it meets their needs and expectations. Stakeholders experiment with the system and feed back requirements changes to the development team.

3. **Test-case generation :** Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of test-driven development.

Challenges:

- It's hard to fully validate requirements before development.
- Users may struggle to imagine how a system will fit into their workflow.
- Changes are often needed even after initial agreement.

4.6 – Requirements change

- Large systems face constant requirement changes due to evolving understanding and external factors.
- These systems often solve **“wicked” problems**, which can’t be fully defined upfront.

Reasons for Change:

1. **Changing business/technical environment** – new hardware, regulations, or integration needs.
2. **Conflicting needs** – system buyers and users often differ in priorities.
3. **Diverse stakeholders** – different goals require compromises that evolve over time.

Requirements Management:

- Track and link requirements to assess change impact.
- Start managing changes early using a formal process.

Agile Approach:

- Handles change flexibly with **user-prioritized iterations**.
- **Downside:** Users may not assess cost-effectiveness well.
- **Best practice:** Use an **independent decision-maker** to balance stakeholder needs.

Requirements management planning:

Defines how evolving requirements will be tracked and managed.

Key Planning Decisions:

1. **Requirements Identification:** Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.
2. **Change Management Process** – This is the set of activities that assess the impact and cost of changes.
3. **Traceability Policies** – These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.
4. **Tool Support** – Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to shared spreadsheets and simple database systems.

Tool Support Should Include:

- **Requirements Storage** – The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
- **Change Management** – The process of change management is simplified if active tool support is available. Tools can keep track of suggested changes and responses to these suggestions.
- **Traceability Management**– Tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

What is Requirement Change Management?

- It is the process of **handling changes to requirements** during the software development lifecycle.
- Since requirements may change due to new customer needs, business goals, technology updates, or errors in earlier requirements, this process ensures changes are properly analyzed, costed, and implemented without disrupting the system.
- Goal: To make sure that changes are **evaluated, approved, and controlled systematically**

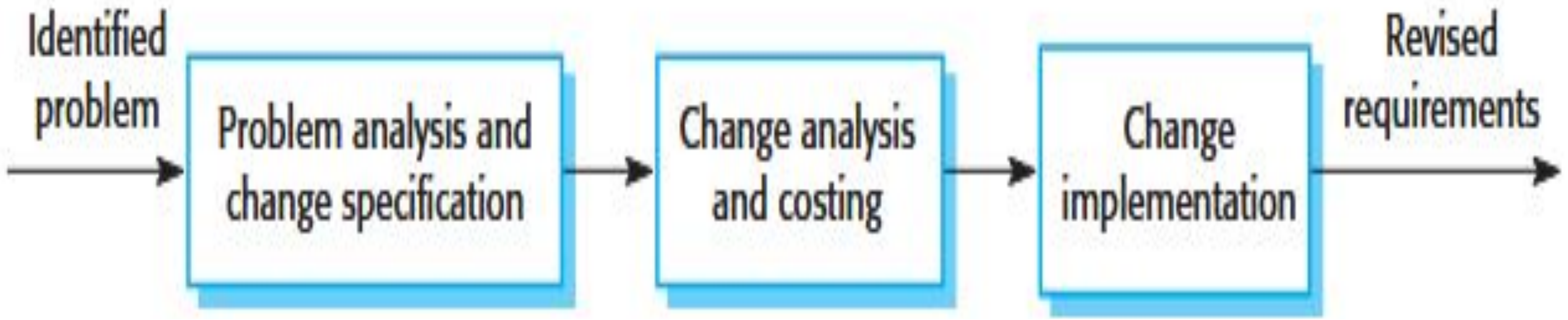

Identified problem
Problem analysis and change specification
Change analysis and costing
Change implementation
Revised requirements

Figure 4.19
Requirements change management

1. Problem Analysis and Change Specification - The process starts with an **identified requirements problem** or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

2. Change Analysis and Costing - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made as to whether or not to proceed with the requirements change.

3. Change Implementation - The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document. This almost inevitably leads to the requirements specification and the system implementation getting out of step. Once system changes have been made, it is easy to forget to include these changes in the requirements document. In some circumstances, emergency changes to a system have to be made. In those cases, it is important that you update the requirements document as soon as possible in order to include the **revised requirements**.

THANK YOU