

Software Engineering (Code: CSE3122)

B. Tech CSE 5th Sem.
(School of Computer Engineering)

Dr. Kranthi Kumar Lella
Associate Professor,
School of Computer Engineering,
Manipal Institute of Technology,
MAHE, Manipal – 576104.

SOFTWARE ENGINEERING

Module – 2 (SOFTWARE LIFE CYCLE MODELS)

CONTENTS

2.1 A few basic concepts

2.2 Waterfall model and its extensions

2.3 Rapid Application Development

2.4 Agile development models

2.5 Spiral Model

2.6 A Comparison of different Life Cycle models

2.1 A FEW BASIC CONCEPTS

Software Life Cycle

The **software life cycle** is similar to a biological life cycle—it describes the **stages** a software goes through:

1. **Inception:** A customer expresses the need for software, but requirements are unclear.
2. **Development:** Software evolves through identifiable **phases** (e.g., design, coding) based on developer actions.
3. **Operation (Maintenance):** After release, users use the software and request **bug fixes and improvements**. This phase is usually the longest.
4. **Retirement:** The software is discarded when it's no longer useful due to **changing needs or technology**.

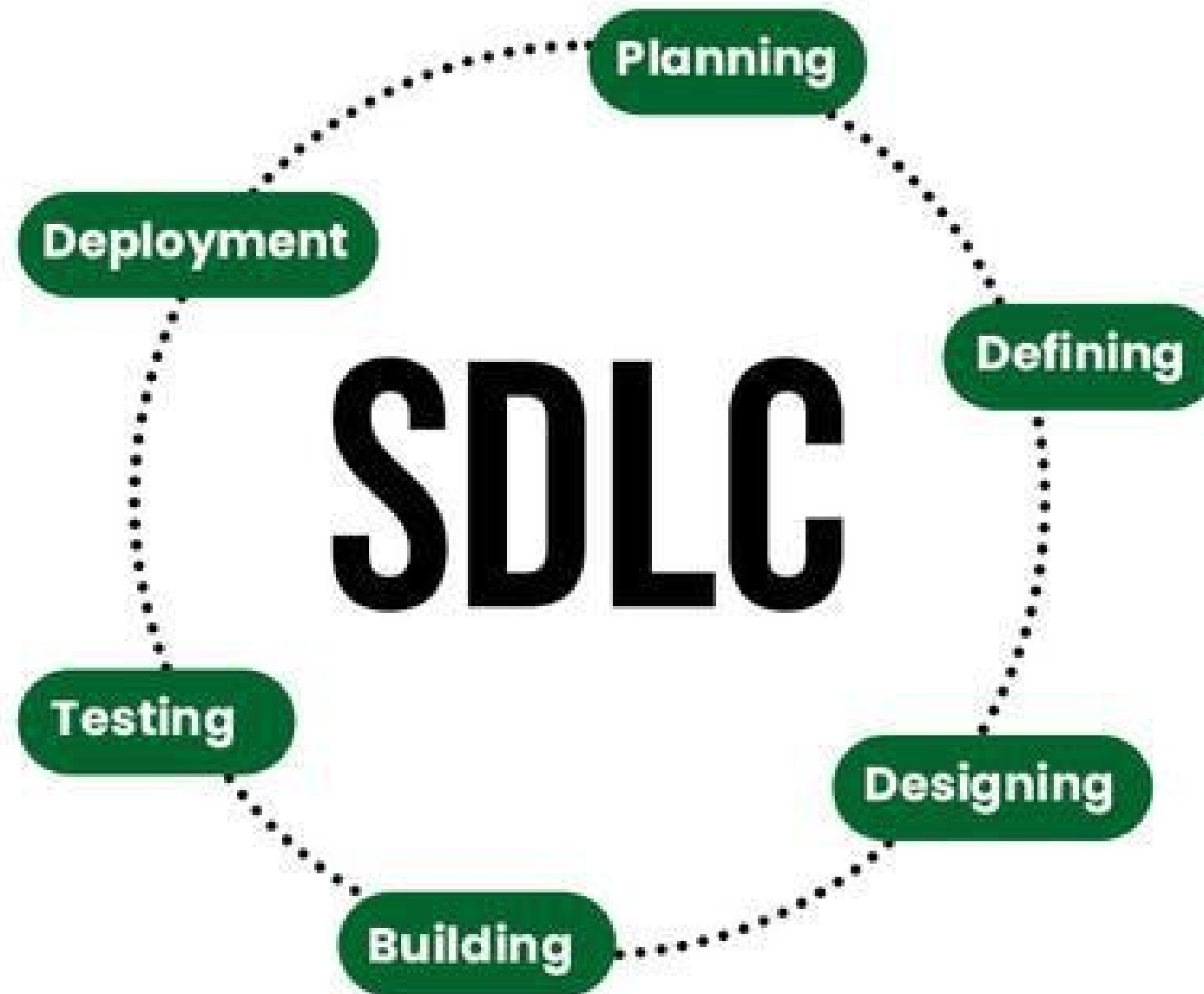
Software Development Life Cycle (SDLC) Model

An **SDLC model** (also called **software process model**) outlines **well-defined activities** needed to transition from one stage to another in the software life cycle.

- Example: Moving from requirements to design involves requirement gathering, analysis, documentation, and customer review.
- SDLC is often shown as a **graphical model** with phases and transitions, along with **textual descriptions** of activities.

Process vs Methodology

- A **process** has a **broader scope**; it covers all or major software activities (e.g., design process, testing process).
- A **methodology** is **narrower**, describing **specific steps** to perform an activity (e.g., a method for high-level design).
- A process may recommend **methodologies** for each phase.



Why Use a Development Process?

Using a well-defined development process:

- Ensures **systematic and disciplined** development.
- Is **crucial for team-based projects** (programming-in-the-large).
- Avoids problems like poor coordination, unclear responsibilities, and integration issues.
- ♦ Example: If team members develop independently without coordination, **conflicts and mismatches** will arise—leading to project failure.
- In contrast, **programming-in-the-small** (e.g., student assignments) might succeed without a formal process.

Why Document the Development Process?

Documenting the process is important because:

1. It clearly defines **activities**, their **sequence**, and **methodologies**, reducing **confusion and misinterpretation**.
2. It signals management's **seriousness**, encouraging developers to follow it strictly.
3. It allows for **tailoring** of the process to suit specific project needs.
4. It is **required by quality standards** like **ISO 9000** and **SEI CMM** for certification and client trust.
5. Helps identify **inconsistencies, omissions**, and improves **training** for new employees.

Phase Entry and Exit Criteria

A good SDLC defines:

- **Entry criteria:** Conditions that must be met to **start** a phase.
- **Exit criteria:** Conditions to **complete** a phase (e.g., reviewed and approved SRS to exit requirements phase).
- ◆ Without clear criteria:
 - Developers may **claim early completion**, causing confusion.
 - Project managers can't accurately track progress, leading to the “**99% complete syndrome**”—developers claim near-completion while work is far from done.
- ❖ Understanding and following a well-documented **SDLC model** is essential for:
 - ❖ Delivering **high-quality software**
 - ❖ Avoiding **project failures**
 - ❖ Managing **team-based development effectively**

2.2 Waterfall Model and Its Extensions

- The **Waterfall Model** was widely used in the **1970s** and still finds use today.
- It is the **most useful** model for team-based software development.
- All other life cycle models are **extensions** of the **classical waterfall model**, tailored for specific project needs.

2.2.1 Classical Waterfall Model

❖ Nature of the Model

- The **classical waterfall model** is **simple** and follows a **linear** sequence of phases.
- It is idealistic because it **assumes no mistakes** during development, but in reality, **errors are discovered in later stages**.
- It has **no provision to revisit** or correct earlier phases, making it **impractical for non-trivial projects**.
- Still, it is important to study as a **foundational model**, and it is **implicitly used for documentation purposes**.

Phases of the Classical Waterfall Model

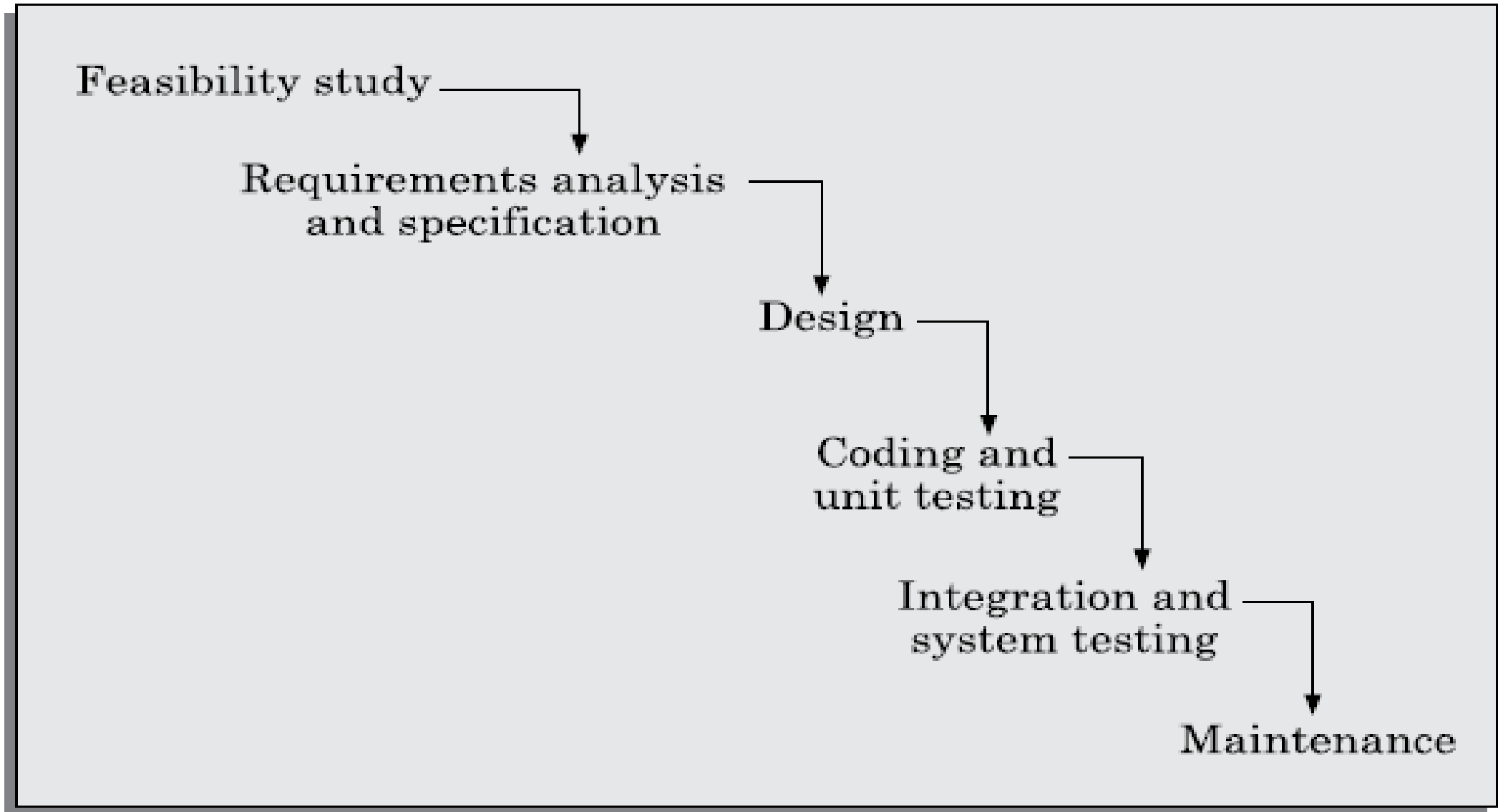


FIGURE 2.1 Classical waterfall model.

Phases of the Classical Waterfall Model (cntd..)

The model consists of **six main phases** (see Fig. 2.1):

- 1.Feasibility Study**
- 2.Requirements Analysis and Specification**
- 3.Design**
- 4.Coding and Unit Testing**
- 5.Integration and System Testing**
- 6.Maintenance**

- The first five are **development phases**; after these, the software is delivered to the customer.
- Once delivered, the **operation or maintenance phase** begins, where bug fixes and updates are made.
- **Note: Project management** is a critical activity throughout all phases but isn't treated as a separate phase.

Effort Distribution (Fig. 2.2)

- The **maintenance phase** consumes about **60%** of total project effort.
- Within development phases, **integration and system testing** requires the **most effort**.

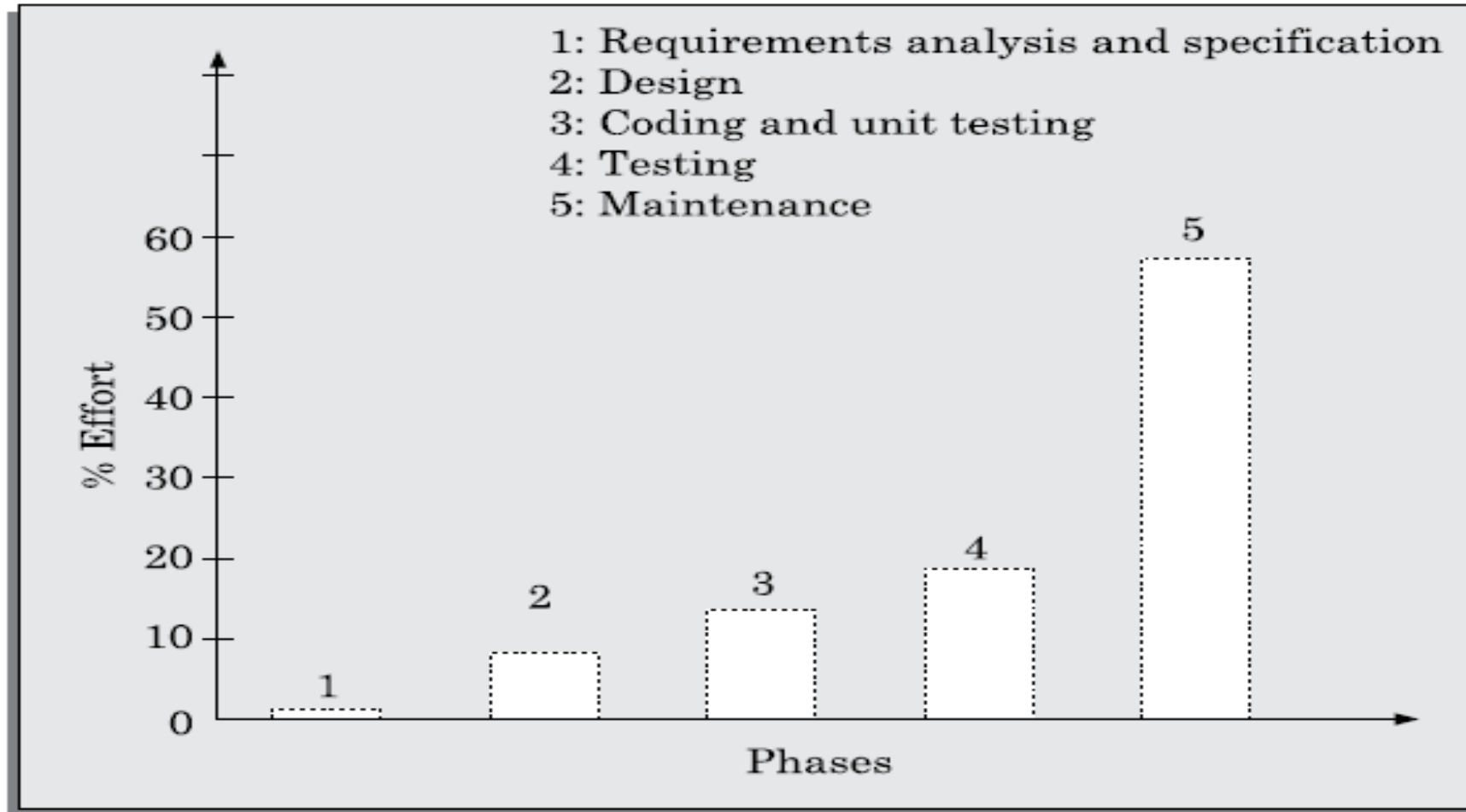


FIGURE 2.2 Relative effort distribution among different phases of a typical product.

Feasibility Study (First Phase)

The goal is to determine if the project is **technically and financially feasible**.

◆ Key Activities:

1. Collect basic information: Inputs, processing needs, expected outputs, and constraints.

2. Understand the customer's needs:

- Only focus on **key requirements**, not UI layouts, specific algorithms, or database designs.

3. Formulate possible solution strategies:

- Example: Client-server model vs standalone application.

4. Evaluate solution strategies:

- Estimate **resources, cost, and time**.
- Compare solutions to choose the best one.
- If **no solution is feasible**, the project is **abandoned**.

◆ Outcome: A **high-level decision** on whether to proceed with the project, and **which solution strategy** to follow.

Case Study 2.1 – Feasibility Study Example

Company: Galaxy Mining Company Ltd. (GMC Ltd.)

Need: A system to manage a **Special Provident Fund (SPF)** for miners, separate from the standard PF, to allow **quick compensation**.

Problem:

- GMC Ltd. has **50 mines** across **8 Indian states**.
- Monthly SPF deductions from miners need to be sent to a **Central SPF Commissioner (CSPFC)**.
- Manual record-keeping is time-consuming and delays settlements.

Solution Request:

- GMC Ltd. asked **Adventure Software Inc.** to build the SPF software.
- Budget: **₹1 million**.

Feasibility Study:

- **Two solution strategies:**
 - **Centralized database** via satellite link.
 - Fails completely if communication breaks.
 - **Local databases** at mine sites + **periodic updates** to central DB via dial-up.
 - **More fault-tolerant** and affordable.
 - Local systems work even if communication fails.
- The **second solution** is technically and financially feasible and was **approved** by GMC Ltd.

Phases of the Classical Waterfall Model (Fig. 2.1)

1. Feasibility Study

- Check **technical and financial feasibility**.
- Identify problem scope, high-level strategies, cost, time, and resource estimates.
- Outcome: Decide **whether to proceed** and **which strategy** to adopt.

2. Requirements Analysis and Specification

Objective: Understand and document customer requirements.

- **◆ Requirements Gathering & Analysis:**
 - Collect and analyze customer needs.
 - Remove **inconsistencies** and **incompleteness**.
- **◆ Requirements Specification:**
 - Create the **SRS (Software Requirements Specification)** document.
 - Written in **end-user language**, acts as a **contract**.
 - Serves as a base for further development and testing.

3. Design

Goal: Translate SRS into software architecture.

- **◆ Procedural Design Approach:**
 - Based on **Data Flow Diagrams (DFD)**.
 - Two parts:
 - **High-Level Design (Architecture):** Break into modules.
 - **Detailed Design:** Design internals like algorithms and data structures.
- **◆ Object-Oriented Design (OOD):**
 - Identify objects and relationships.
 - Easier maintenance and faster development.

4. Coding and Unit Testing

- Translate design into **source code**.
- Each module is **unit tested**:
 - Design test cases
 - Debug and verify correctness

5. Integration and System Testing

- Modules are **incrementally integrated**.
- Two levels of testing:
 - **Integration Testing**: Check module interfaces.
 - **System Testing**: Ensure full system meets **SRS**.
- **Types of system testing**:
 - **α-testing**: By development team
 - **β-testing**: By friendly customers
 - **Acceptance Testing**: By customer to approve software

6. Maintenance

- Maintenance requires **more effort** than development (approx. 60%).
- **Types:**
 - **Corrective:** Fix bugs
 - **Perfective:** Improve performance/features
 - **Adaptive:** Port to new platforms or environments

Shortcomings of the Classical Waterfall Model

- **No Feedback Paths:**
 - Assumes **no mistakes** in any phase; no option to revise earlier phases.
- **Difficult to Handle Changes:**
 - Customer **change requests** can't be easily handled after requirement phase.
- **Late Testing:**
 - Integration/testing occurs **late**, making error correction **costly**.
- **No Overlapping of Phases:**
 - Sequential execution leads to **inefficient resource use**.
 - In practice, **phases overlap** (e.g., testing starts after SRS).

Is the Waterfall Model Still Useful?

- Though **rarely used directly** for development, it is useful for:
- Understanding **foundation** of other models.
- **Documentation purposes**, as suggested by **Parnas** and **Hoare**.

❖ Like a mathematician presents a **clean proof**, even after multiple trials and errors, the final software documentation is written **as if** development followed the waterfall model exactly.

2.2.2 Iterative Waterfall Model

To overcome the **rigid limitations** of the classical waterfall model by allowing **feedback** and **error correction** during development.

Main Change:

Introduction of **feedback paths** from each phase to its **previous phases** (except feasibility phase).

📌 This means if an error is found in a later phase (e.g., testing), earlier phases (like design or coding) can be **revisited and corrected**.

📌 No feedback is allowed to **feasibility phase** because once a project is approved, it's typically not abandoned due to **legal/moral commitments**.

The feedback paths introduced by the iterative waterfall model are shown in Figure 2.3. The feedback paths allow for correcting errors committed by a programmer during some phase, as and when these are detected in a later phase.

Please notice that in Figure 2.3 there is no feedback path to the feasibility stage. This is because once a team accepts to take up a project, it does not give up the project easily due to legal and moral reasons.

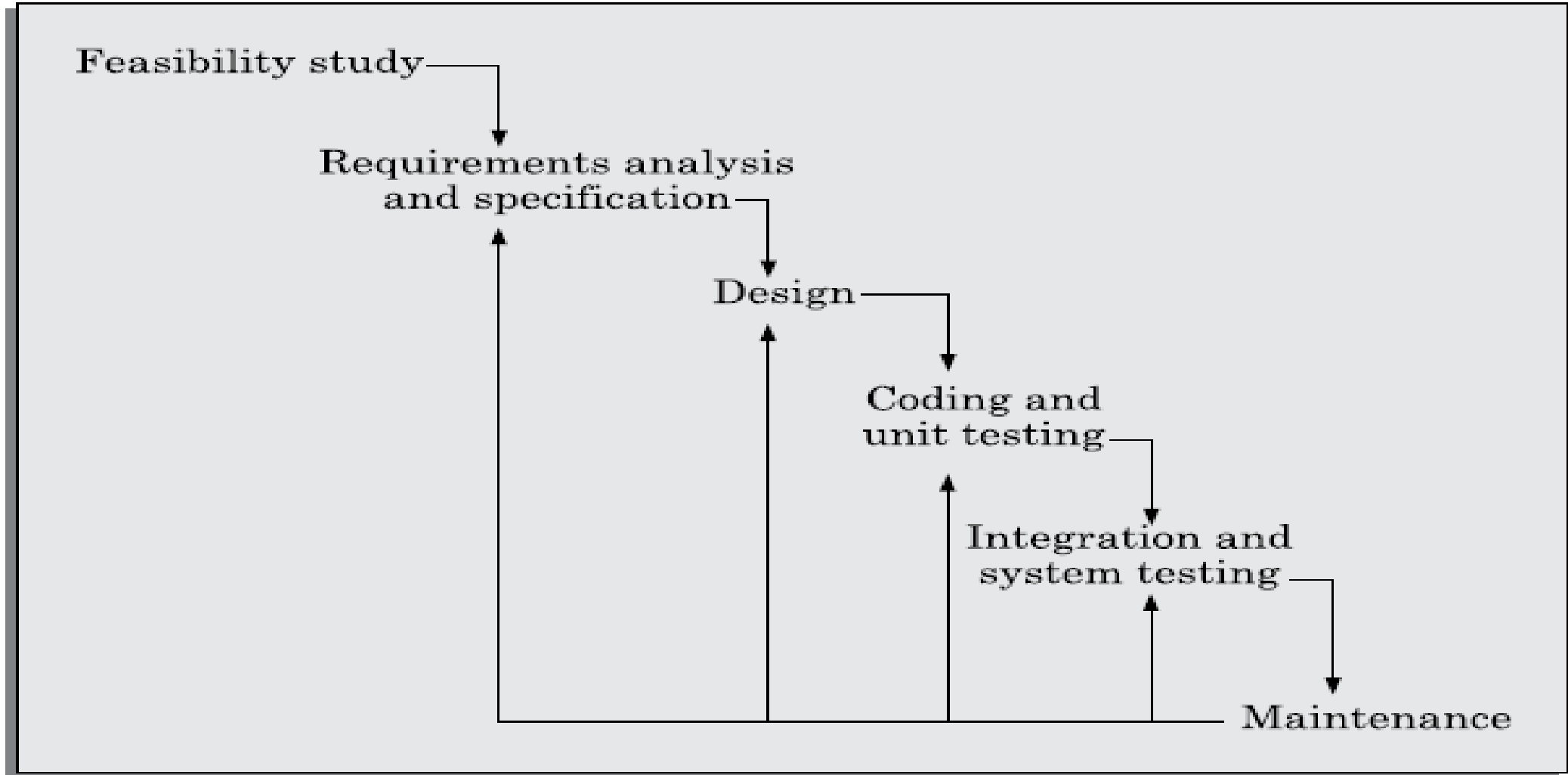


FIGURE 2.3 Iterative waterfall model.

Error Detection Strategy – Phase Containment of Errors:

Detect errors in the same phase where they were introduced.

- Early detection = lower cost and time.
- Example: Fixing a design error **during design** is cheaper than fixing it during **testing**.
- **Technique used:** Rigorous **review** of documents (like SRS, design docs, etc.).

Phase Overlap (Refer: Fig. 2.4)

- Even though the model shows a step-by-step sequence, in practice, **phases often overlap** due to:
 - ◆ **Late detection of errors** — forces rework of earlier phases.
 - ◆ **Workload differences** — team members who finish early move on to the next phase instead of waiting idly.

- If we consider such rework after a phase is complete, we can say that the activities pertaining to a phase do not end at the completion of the phase, but overlap with other phases as shown in Figure 2.4.

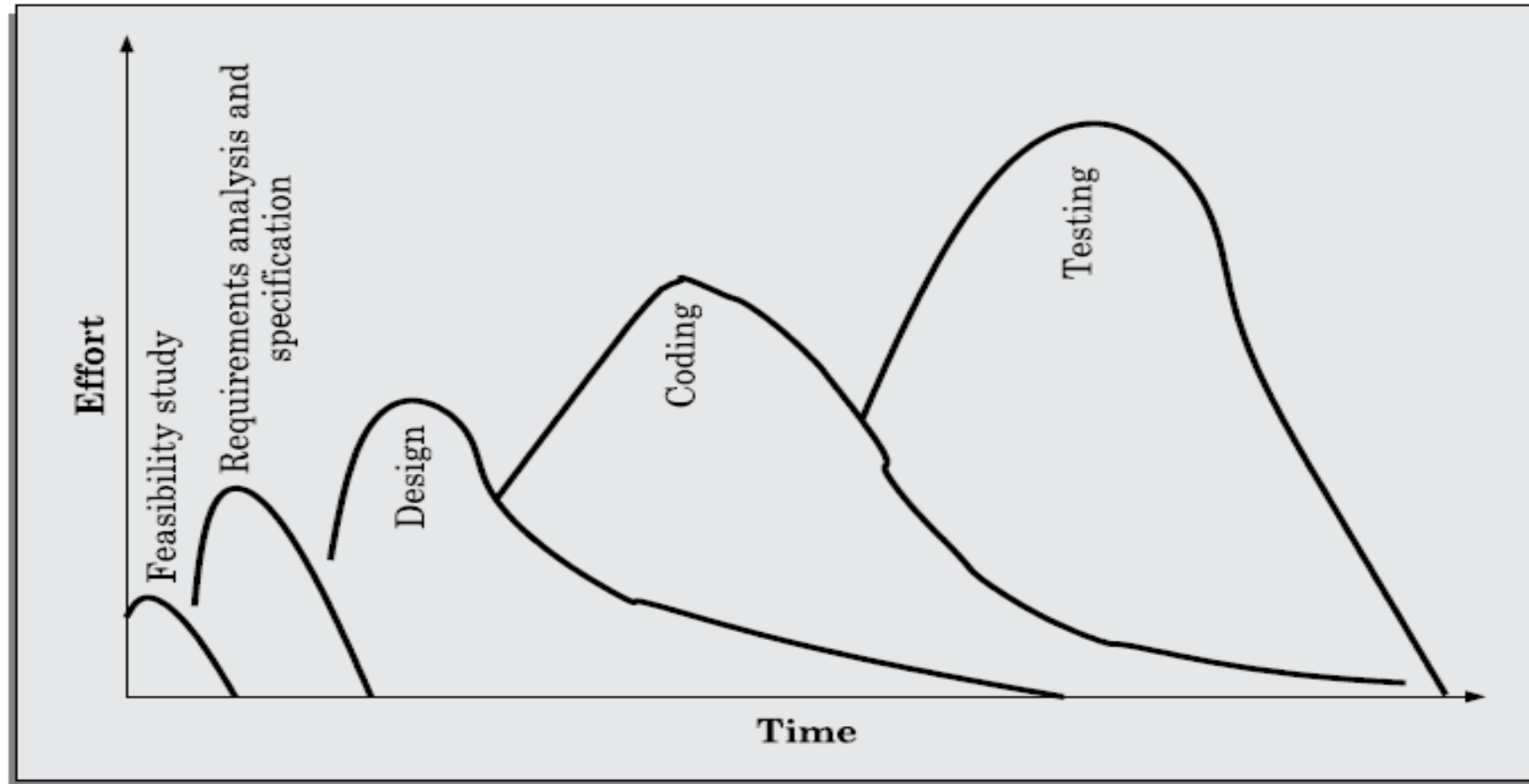


FIGURE 2.4 Distribution of effort for various phases in the iterative waterfall model.

Shortcomings of Iterative Waterfall Model

Even with feedback and overlap, the model has **major limitations**, especially in modern projects:

1. Change Requests Not Supported

- Requirements are **frozen** early.
- **Any change** needs major **replanning and rework**.
- But in real life, requirements **often change** as customers understand their needs better **during development**.

2. No Incremental Delivery

- Software is delivered **only after full development and testing**.
- If the project takes **months or years**, business needs might change, making the final product **irrelevant or outdated**.

3. Rigid Phase Sequence

- Team members may **idle** while waiting for others.
- Causes **resource wastage** and **inefficiency**.

4. Late Error Detection = High Cost

- Validation (testing) happens **after coding**.
- Bugs found late cause **expensive rework**.

5. Limited Customer Interaction

- Interaction only at **beginning** and **end**.
- No involvement during design or testing = software often **misaligned** with customer needs.

6. Heavy weight Model

Emphasizes **extensive documentation**, which:

- Consumes time
- Reduces agility
- May not directly add customer value

7. No Support for Risk or Reuse

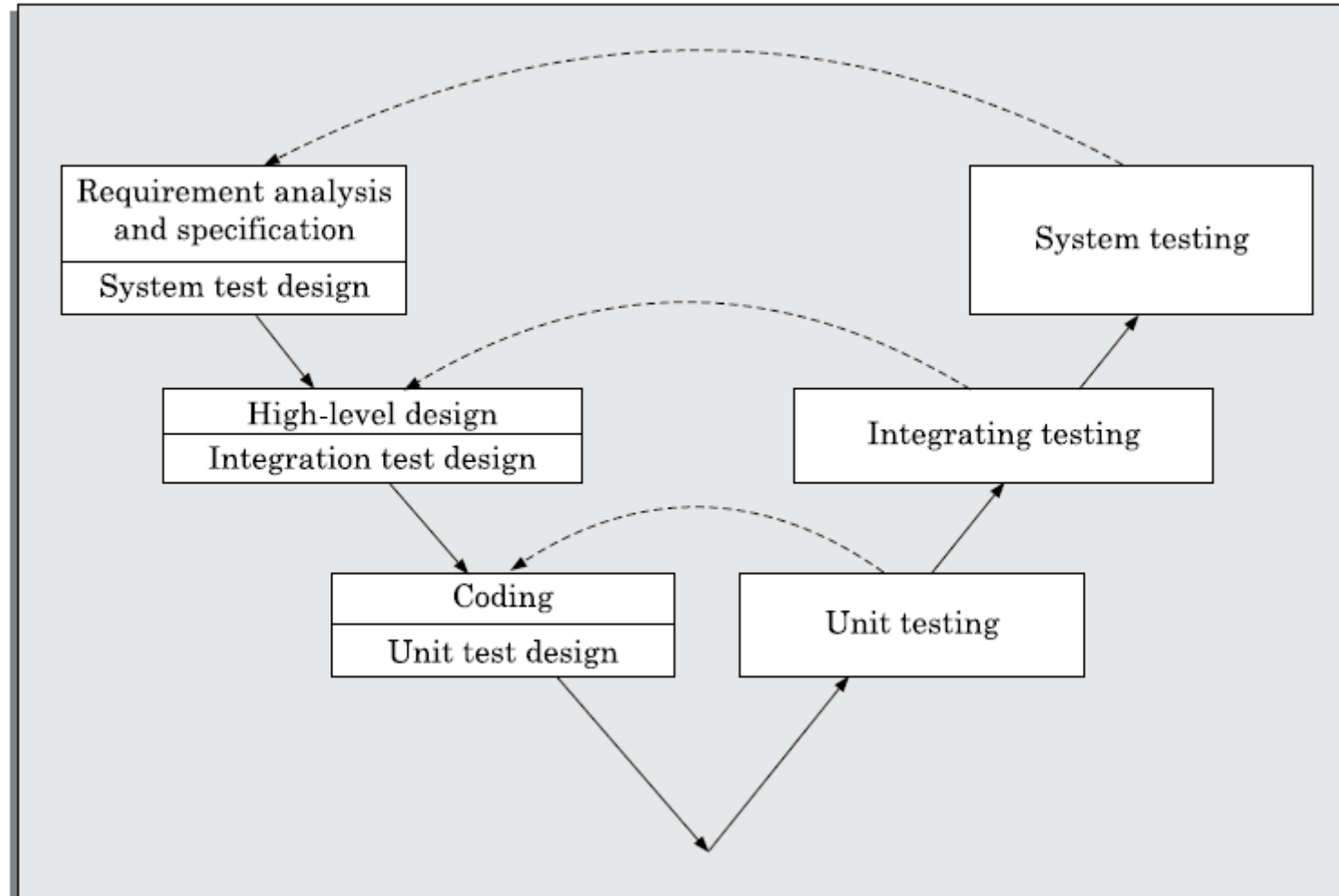
Not suited for:

- Projects with **uncertainty or risk**
- **Reuse-based development**, like using existing components or code libraries

- Iterative waterfall is **an improvement** over classical waterfall.
- However, it's still not suited for **modern, agile, fast-paced projects**.
- Hence, **Agile models** (discussed in Section 2.4) are preferred.

2.2.3 V-Model (Validation and Verification Model)


The **V-model** is a **variant of the waterfall model**, named after its **V-shaped structure** that visually represents the **development (left side)** and **validation (right side)** phases As shown in Figure 2.5 (V-model).





Key Features:

- **Verification and validation** are performed **throughout** the development life cycle.
- Suitable for **safety-critical systems** (e.g., medical, aviation, defense) where **high reliability** is essential.

Structure:

- **Left side (Development Phases):**
 - Each phase (e.g., requirements, design, coding) includes:
 - Creation of the work product.
 - Designing corresponding **test plans and test cases**.
- **Right side (Validation Phases):**
 - Each validation phase corresponds to a development phase.
 - Actual testing is performed:
 - ♦ **Unit testing** → Validates **coding**
 - ♦ **Integration testing** → Validates **design**
 - ♦ **System testing** → Validates **requirements**
-  This ensures **early detection** of defects and better product quality.

V-Model vs. Waterfall Model:

Feature	Waterfall Model	V-Model
Testing timing	Only in the testing phase	Throughout the life cycle
Parallel test planning	 No	 Yes (during development)
Team involvement	Testers join late	Testers involved from start
Error detection	Late (expensive to fix)	Early (less cost, better quality)

Advantages of V-Model:

- 1.Faster development:** Because test planning happens early, the final testing phase is shorter.
- 2.Better test quality:** Test cases are written early, with more care (less time pressure).
- 3.Efficient manpower use:** Testers work **throughout** the project, not just at the end.
- 4.Improved testing effectiveness:** Testers understand the system deeply as they are involved from the beginning.

Disadvantages of V-Model:

Since it's based on the **classical waterfall model**, it inherits its **limitations**, such as:

- **Strict Phase Sequencing**
- **Limited flexibility**
- **Poor support for changing requirements**
- **Heavy documentation**
- **Not suitable for iterative, agile, or reuse-based projects**

❖ The **V-model improves** upon the classical and iterative waterfall models by emphasizing **early test planning** and **parallel development-validation**. However, it is still **not suitable for dynamic or modern agile projects** due to its waterfall-based nature.

2.2.4 Prototyping Model

The **Prototyping Model** is a **popular life cycle model** and can be considered an **extension of the waterfall model**. It involves building a **working prototype** of the system **before** the actual software is developed.

A prototype is a basic, early version of the final software that has:

- **Limited functionality**
- **Low reliability**
- **Inefficient performance**

It is built **quickly using shortcuts**, such as:

- Dummy functions
- Table lookups instead of real computations
- This is also called **rapid prototyping**, often using tools like **4GLs** for GUI construction.

When to Use the Prototyping Model

This model is useful in three main situations:

➤ **Developing GUI (Graphical User Interface):**

- Helps demonstrate **input formats, dialogs, and outputs** to users.
- Users can give feedback by interacting with the prototype instead of imagining a UI.

➤ **When Technical Solutions Are Unclear:**

- Used to **test uncertain technologies** (e.g., compiler development, response time of hardware).
- Helps the team **learn and resolve technical risks** before real development.

➤ **When Perfection Isn't Possible Initially:**

- As Brooks (1975) stated: *“Plan to throw away the first version.”*
- Helps to refine requirements and design before developing an **efficient final product**.

Life Cycle Activities in Prototyping Model

As shown in Figure 2.6, the model consists of two main phases:

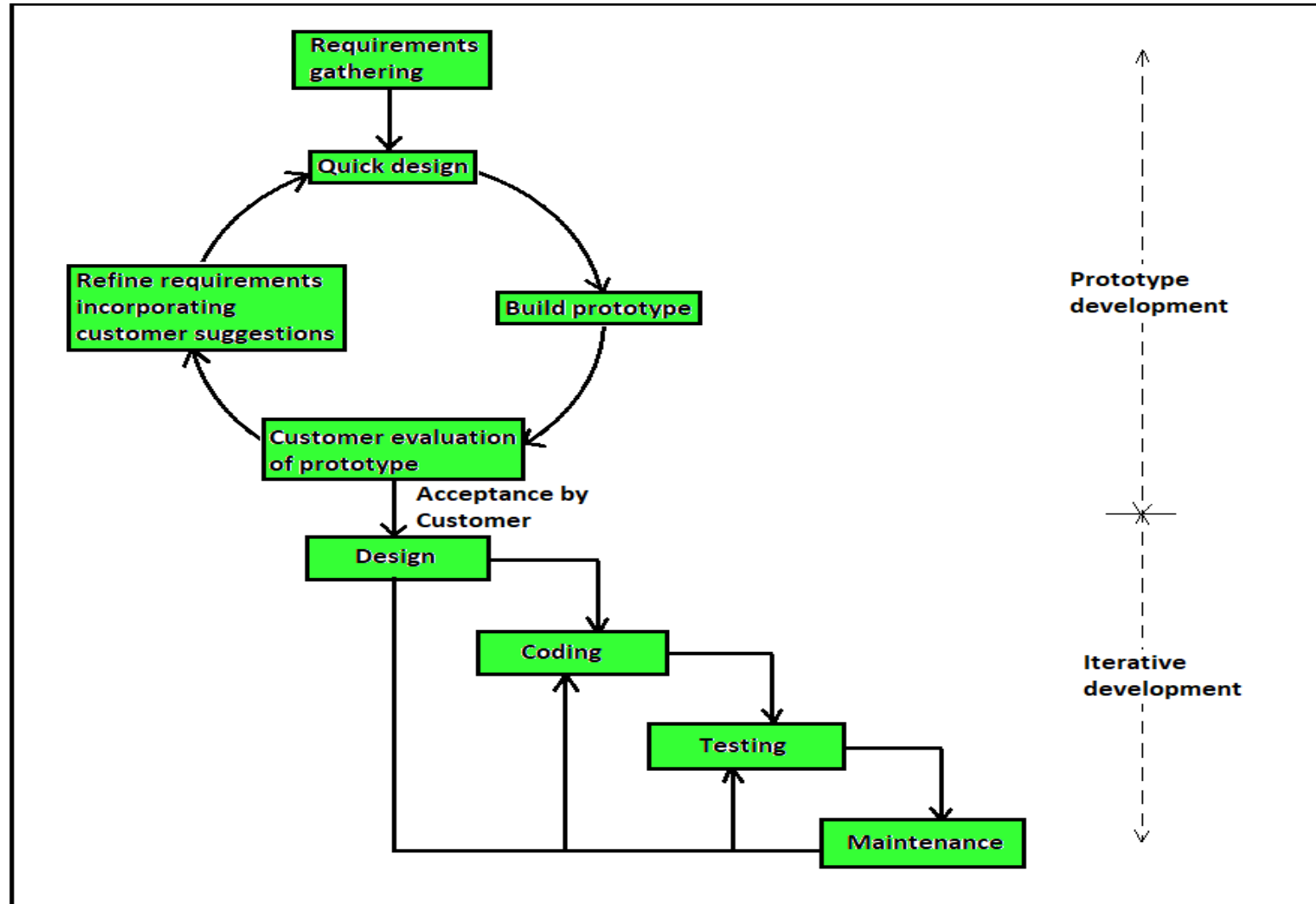



FIGURE 2.6 Prototyping model of software development.

1. Prototype Development:

- Start with **initial requirement gathering**
- Do a **quick design** and build the prototype
- Submit it to the **customer for feedback**
- Refine requirements and modify prototype **iteratively** until the customer **approves**

2. Iterative Waterfall Development:

- Once prototype is approved, use **iterative waterfall** model to develop the actual software
- Even if a prototype exists, an **SRS (Software Requirements Specification)** is needed for:
 - **Traceability**
 - **Verification**
 - **Test planning**
- For GUI parts, SRS may not be needed, as the **approved prototype serves as animated specifications.**
-  **Prototype code is usually discarded, but the experience gained helps in real development.**

Advantages of Prototyping Model

- Excellent for projects with:
 - **Unclear requirements**
 - **Technical uncertainties**
- Helps clarify requirements early
- Reduces **future changes** and **redesign costs**

Disadvantages of Prototyping Model

- **Increases cost** if used for:
 - Routine projects with no significant risks
- Only helps if **risks are known upfront**
- **Ineffective** for risks identified **later** during development (e.g., staff leaving midway)

- The **Prototyping Model** is best suited for projects involving **unclear requirements** or **technical risks**, especially **GUI development**. It helps refine requirements early but may add **unnecessary cost** for well-understood, routine projects.

2.2.5 Incremental Development Model

In the **incremental life cycle model**, software is developed and delivered in **increments**—small parts that gradually build up the full system.

- **First**, the overall requirements are broken into **smaller, manageable units** called *increments*.
- The **first increment** delivers a **basic working version** with **core features**.
- In **each iteration**, a **new increment** adds **additional features** or **refines existing ones**, until the **complete system** is built.

📌 **Figure 2.7** illustrates the concept of incrementally building a system.

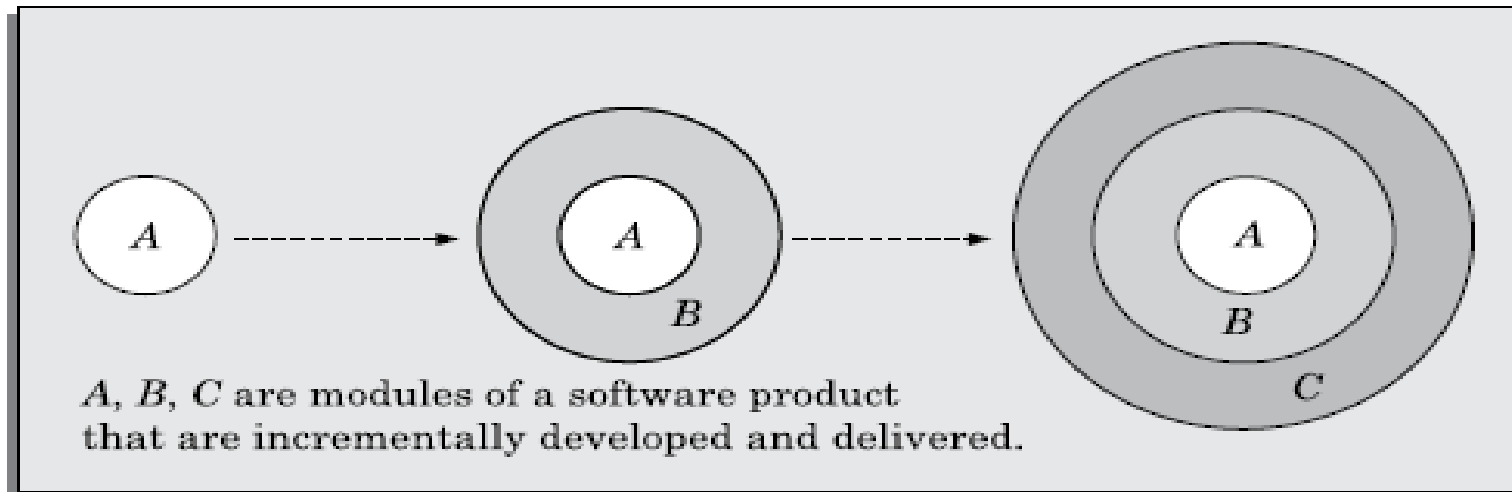


FIGURE 2.7
Incremental software development.

Life Cycle Activities in Incremental Model

➤ Requirements Breakdown:

- All software requirements are split into **modules/features**, called *increments*.

➤ No Long-Term Plan:

- At any point, the team **plans only for the next increment**, making it easier to handle **change requests**.

➤ Core Features First:

- Development starts with **core features**, which do **not depend on other parts**.
- **Non-core features**, which depend on core services, are added in later increments.

➤ Iterative Construction:

- Each increment is developed using the **iterative waterfall model**.
- After each version is delivered, **customer feedback** is collected and used in the next iteration.

➤ Successive Delivery:

- Each version delivered to the customer:
 - Adds **new features**
 - **Improves existing features**
- After the final increment (**increment n**), the full software system is complete and deployed.

📌 This process is shown schematically in **Figure 2.8**.

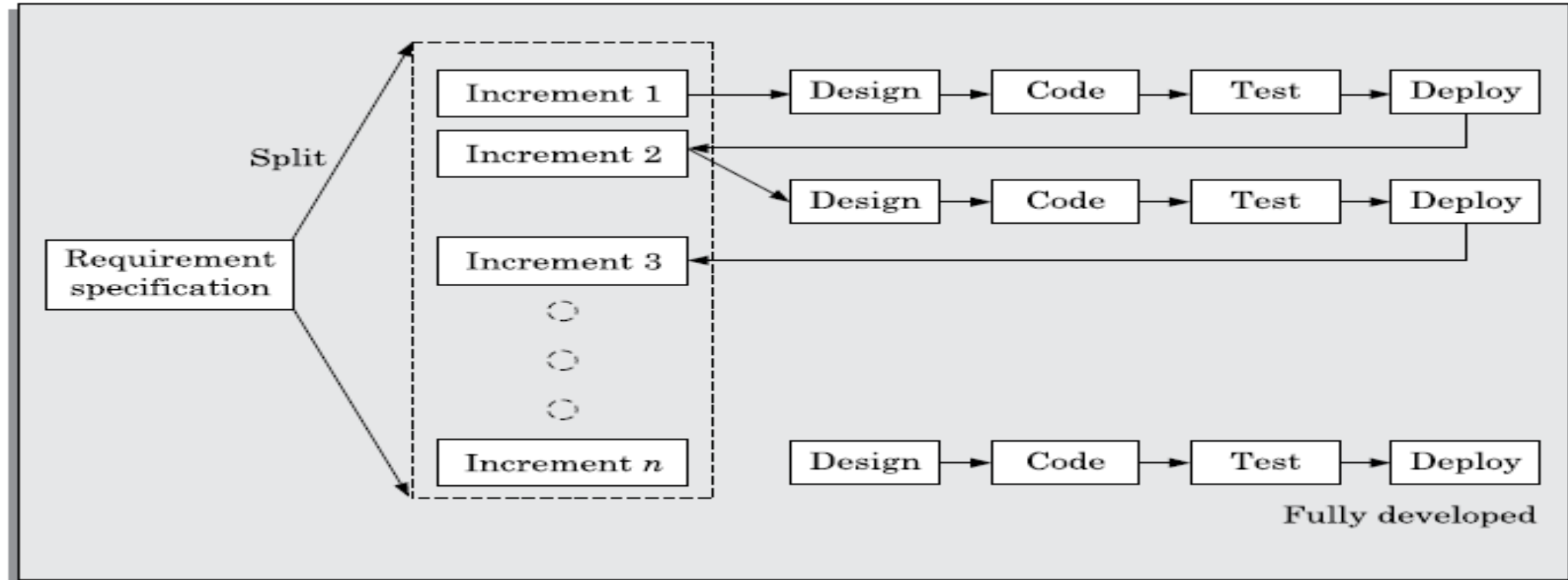


FIGURE 2.8 Incremental model of software development.

Advantages of Incremental Model



Error Reduction:

- Core modules are delivered and used early.
- They get **thorough testing** through repeated use.
- This increases **reliability** of the final product.



Incremental Resource Deployment:

- Customers don't need to commit **large resources at once**.
- Developers also **avoid bulk resource usage**, making the process **cost-effective and flexible**.

The **Incremental Model** breaks development into manageable chunks and delivers value early. It allows:

- **Better handling of changes**
- **Frequent customer feedback**
- **Reduced errors**
- **Gradual deployment of resources**
- It's ideal for projects where requirements evolve over time or where early delivery of parts of the system is beneficial.

2.2.6 Evolutionary Model

- The **Evolutionary Model** is a software development life cycle model where the system is built **incrementally**—one feature at a time—and refined over time based on **user feedback**. It shares similarities with the **incremental model** but introduces a key difference:
- ❖ **Requirements, plans, and solutions evolve during the development**, rather than being finalized upfront.

Key Idea: Evolve As You Go

- The model follows a "**design a little, build a little, test a little, deploy a little**" approach.
- After getting a **rough idea** of what is needed, development starts immediately.
- New features and refinements are added with **each version**, based on customer feedback.



A visual representation is shown in **Figure 2.9** (Evolutionary development diagram).

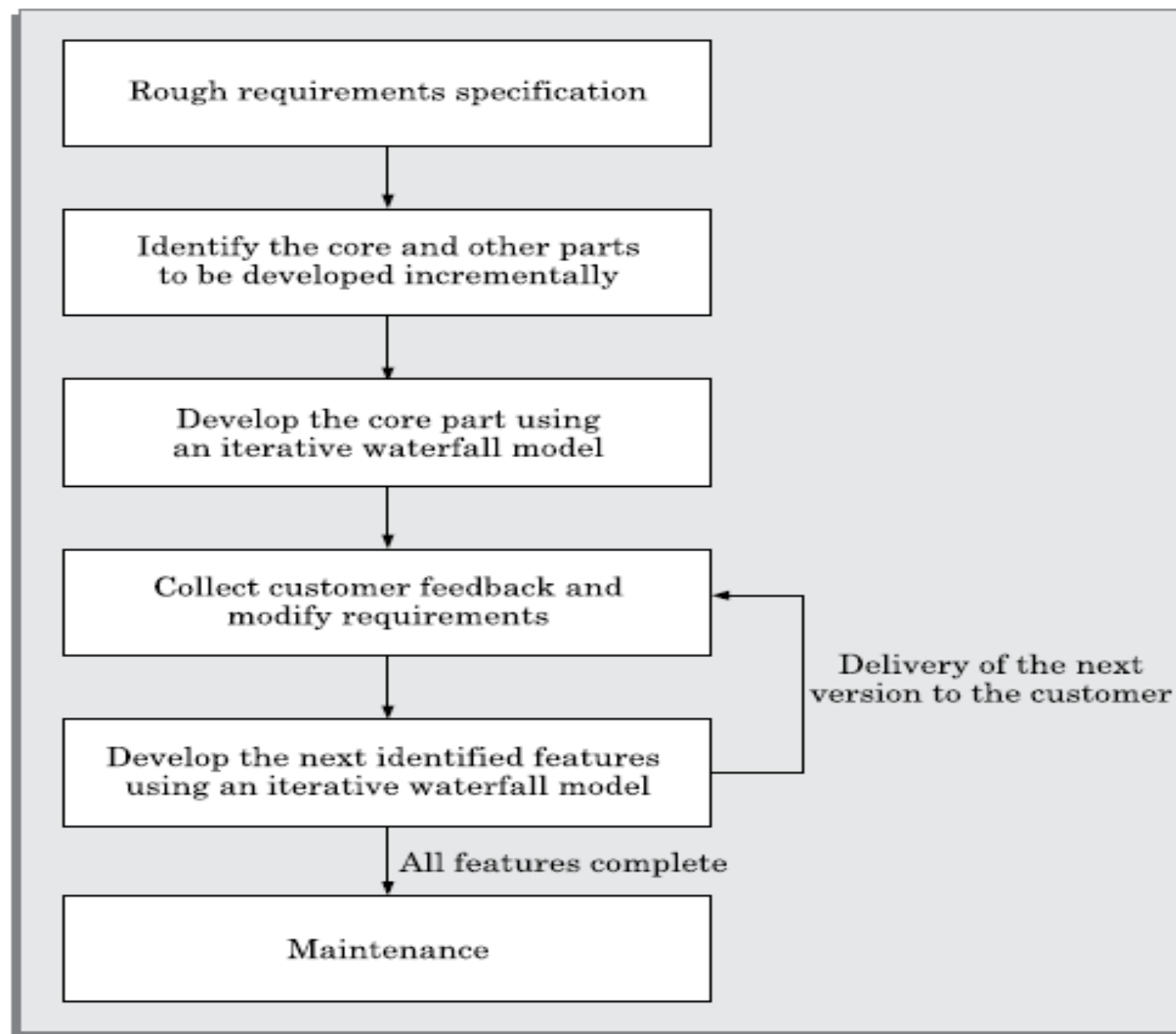


FIGURE 2.9 Evolutionary model of software development.

Comparison with Incremental Model

Aspect	Incremental Model	Evolutionary Model
Requirements	Fully defined <i>before</i> development begins	Evolve during development iterations
Start of development	After complete SRS document is ready	Starts with rough requirements
Feedback integration	Limited (feedback mainly after increments)	Strong emphasis on continuous feedback
Nature	Plan-driven, sequential increments	Adaptive and evolving

Advantages

1. ✓ Better understanding of requirements:

1. Users interact with early, partial versions of the software.
2. Feedback helps refine and clarify actual needs early on.
3. Result: **Fewer change requests** after final delivery.

2. ✓ Easier change handling:

1. Since there is a change in **long-term plans**, adapting to changes is easier.
2. **Rework effort is minimal** compared to sequential models.

Disadvantages

➤ **Difficult to divide features:**

- For some projects, especially **small or tightly coupled systems**, splitting features into incremental parts can be **complex and time-consuming**.

➤ **Ad-hoc design risk:**

- Since only the **current increment** is designed at a time, overall system architecture can become **unstructured or inefficient**, harming **maintainability**.

• The **evolutionary model** allows building and refining software through **multiple iterations** based on real-time feedback.

• It is ideal when **requirements are unclear or likely to change**.

• However, it may not suit **small, well-defined projects** where complete planning is feasible from the beginning.

❖ Modern agile methods combine both incremental and evolutionary principles to get the best of both worlds.

2.3 Rapid Application Development (RAD)

RAD is a software development model introduced in the early 1990s to overcome the **rigid, change-resistant nature** of the **waterfall model**. It combines key elements of the **prototyping** and **evolutionary** models and promotes **faster development, early user feedback, and easier incorporation of changes**.

Main Goals of RAD

- **Reduce development time and cost**
- **Easily handle change requests**
- **Minimize communication gaps** between developers and customers.

How RAD Works

- **Development happens in short cycles** (called **time boxes**)
- Each cycle focuses on a **small functionality**
- A **prototype** is quickly created, shown to the customer, and **refined** based on feedback
- **Prototypes are not discarded** (unlike in the prototyping model); instead, they are **enhanced** to become the final product
- A **customer representative** is part of the development team to ensure clarity of requirements.

How RAD Supports Change and Speed



Change Accommodation:

- Customers suggest changes soon after using each increment.
- Since each feature is developed independently, changes can be made without reworking the entire system.



Fast Development:

- Minimal long-term planning
- **Heavy reuse** of existing components
- Emphasis on **rapid prototyping** using tools that support:
 - Visual development
 - Reusable components

When RAD is Suitable

RAD works well when:

- **Customized software** is needed
 - e.g., adapting existing educational software for different institutions
- **Non-critical software**
 - Performance and reliability are not mission-critical
- **Tight project schedules**
 - Aggressive deadlines make RAD attractive
- **Large software systems**
 - Easier to break into incremental deliveries

When RAD is Unsuitable

Avoid RAD if the project has:

- **Generic products (widely distributed)**
 - Need high performance/reliability in competitive markets
- **Requirement for optimal performance or reliability**
 - e.g., OS or flight simulator
- **Lack of similar past projects**
 - Limited scope for reuse makes RAD ineffective
- **Monolithic design**
 - If the software can't be split into parts, incremental development is hard

RAD vs. Other Models

Comparison	RAD Model	Other Models
vs. Prototyping	Prototype evolves into final product	Prototype is thrown away
vs. Iterative Waterfall	Small features are incrementally developed with feedback	Entire system is developed before delivery; harder to change
vs. Evolutionary	Quick-and-dirty prototypes with fast delivery	Each version is carefully developed using waterfall steps; more systematic

- RAD emphasizes **speed, user involvement, reuse, and incremental delivery**
- Effective for projects needing **quick turnaround and custom solutions**
- Not ideal for **performance-critical, generic, or tightly coupled** software

2.4 AGILE DEVELOPMENT MODELS

Why Agile was Introduced:

- Waterfall models are rigid and not suitable for modern, dynamic projects.
- Around **40% of requirements arrive after development begins.**
- Agile responds better to changing customer needs, fast delivery demands, and customization.

Agile Model Highlights:

- **Incremental & Iterative:** Breaks down work into small parts delivered in short cycles (time-boxed).
- **Customer collaboration:** Customer representatives are involved throughout.
- **Flexibility:** Activities not needed for a specific project can be skipped.
- **Minimal documentation:** Focuses on working software and face-to-face communication.
- **Team Size:** Small (5–9 members) for effective collaboration.

Agile Principles (Agile Manifesto, 2001):

- Working software over comprehensive documentation.
- Welcomes requirement changes.
- Regular, frequent delivery (every few weeks).
- Emphasizes **people and communication** over tools and processes.
- Encourages **pair programming** and **customer collaboration**.

Extreme Programming (XP):

- Proposed by **Kent Beck (1999)**.
- Based on doing known good practices *to the extreme*.
- **Key Practices:**
 - Pair programming
 - Test-driven development (TDD)
 - Continuous integration
 - Simple design with **daily refactoring**
 - Focus on **user stories**, not formal use cases
 - Frequent feedback

Scrum Model

- Project is divided into small parts of work.
- Incrementally developed and delivered over time boxes that are called sprints.
- s/w developed over a series of manageable chunks.
- Team members assume three fundamental roles.
 - ✓ software owner - Communicates the customer's vision and priorities.
 - ✓ scrum master - Facilitates the process between the Product Owner and the team, ensuring smooth workflow and removing obstacles.
 - ✓ team member - Developers who design, build, and test the product increments.

Lean Software Development:

- Originated from Toyota manufacturing.
- Focus: **eliminate waste**, improve flow, and reduce delays.
- **Kanban Board:** Visualizes workflow using cards/stages.
 - Limits WIP (Work In Progress)
 - Makes bottlenecks and delays visible
 - Improves cycle time and predictability

Model	Key Traits
Waterfall	Heavy documentation, rigid, good for stable, critical systems
Agile	Lightweight, flexible, customer-focused, iterative
RAD	Fast prototyping, but lacks Agile's discipline
Exploratory	Similar informality, but lacks Agile's structure
XP	Emphasizes testing, feedback, simplicity
Lean + Kanban	Efficiency-focused, bottleneck visualization, WIP limits

2.5 SPIRAL MODEL

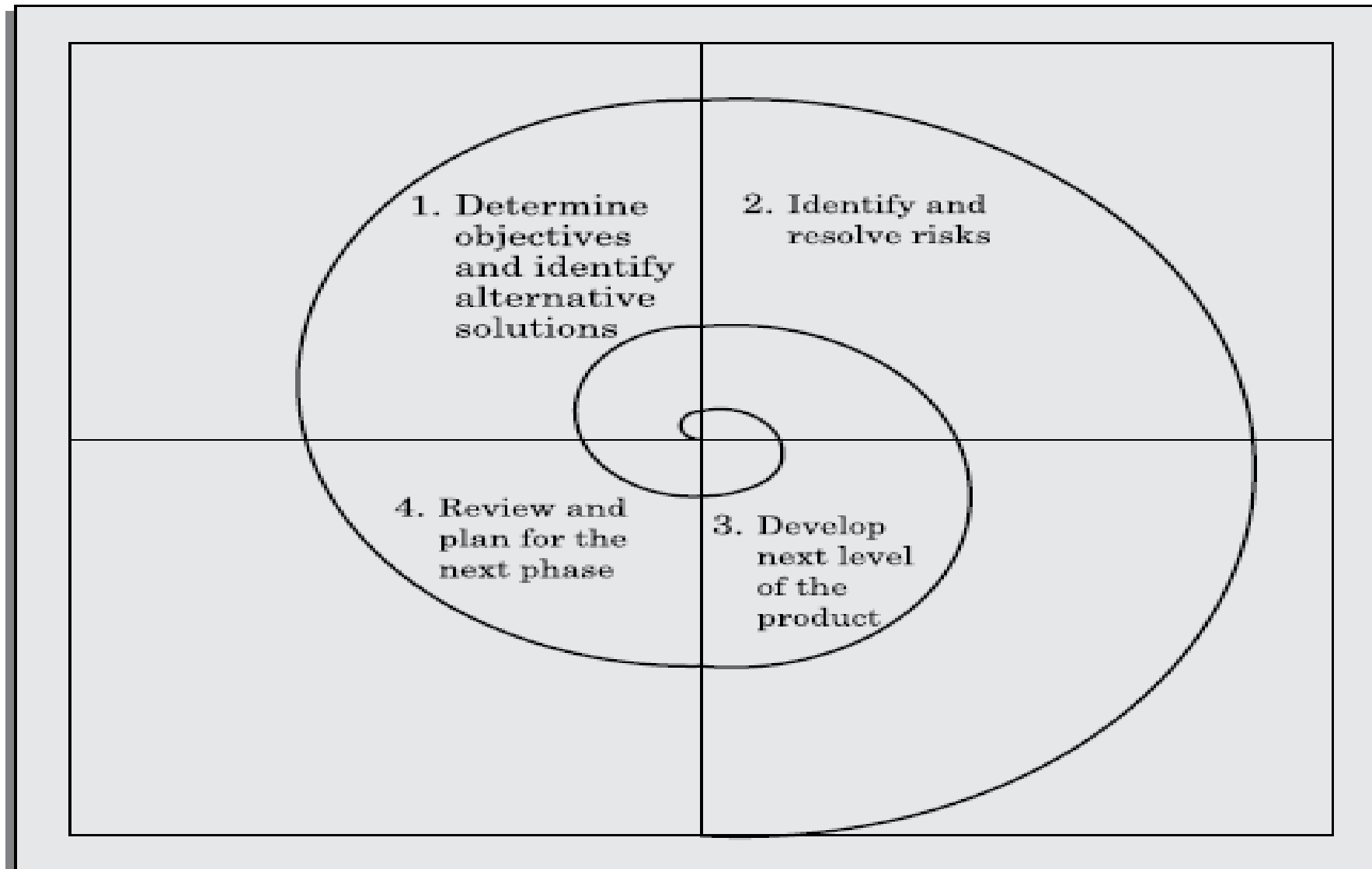


FIGURE 2.10 Spiral model of software development.

- The **spiral model** which resembles a spiral with **multiple loops**.
- Each **loop** is considered a **phase** of development.
- The **number of loops is not fixed** and depends on project-specific risks.

Key Feature: Risk Handling

- Unlike the **prototyping model**, which assumes risks are known **before** development starts, the spiral model **identifies and resolves risks throughout the project**.
- In **every phase**, a **prototype** may be built to analyze risks and test solutions.
- Example: If **remote data access** is slow, a prototype can test performance and suggest improvements like caching or faster communication.

2.5.1 Phases of Spiral Model

Each **phase (loop)** is divided into **4 quadrants**:

➤ Quadrant 1:

- Define objectives and identify possible **risks**.
- Propose **alternative solutions**.

➤ Quadrant 2:

- **Evaluate** the alternatives.
- **Develop prototype(s)** to compare solutions.

➤ Quadrant 3:

- **Develop and test** the selected solution.
- Build the **next version** of the software.

➤ Quadrant 4:

- **Review** the results with the **customer**.
- **Plan the next iteration** of the spiral.

- Each iteration moves **outwards** from the center, gradually building the complete system.

Key Role: Project Manager

- **Decides** the number of phases based on emerging risks.
- **Adapts** the model dynamically for the project.

Parallel Cycles

- Features that can be developed independently may go through **parallel spirals** to speed up development.

Advantages

- Excellent for **risk-driven** projects.
- Supports **prototyping in every phase**.
- More **flexible** than waterfall or prototyping.
- Suitable for **large, high-risk** projects.

Disadvantages

- **Complex** to manage and implement.
- Needs **experienced staff**.
- Not suitable for **outsourced** or low-risk projects.

Spiral Model as a Meta Model

- It **includes** features from:
 - **Waterfall model** (systematic approach)
 - **Prototyping model** (risk resolution via prototypes)
 - **Evolutionary model** (software evolves across iterations)
- ❖ **A single spiral loop behaves like the waterfall model.**

Case Study: Galaxy Inc.

- Project: Satellite-based mobile communication across Earth.
- **High risk** due to unknown challenges (e.g., satellite handoff).
- Used **spiral model** because:
 - Risks couldn't be fully known upfront.
 - Project required **millions of lines of code**.
 - Used **parallel spirals** to speed up development.
 - Project successfully completed in **5 years** after multiple refinements in **cost and schedule**.

The spiral model is ideal for:	It's not suitable for:
Large, complex, high-risk projects	Small or low-risk projects
Projects where new risks emerge during development	Teams with inexperienced staff
Teams with experienced developers	Outsourced projects

2.6 Comparison of Software Life Cycle Models

1. Waterfall Model (Classical)

- Acts as the **basic model**.
- Does **not allow correction** of errors found in later phases.
- Hence, **not suitable** for practical projects - E-Commerce, Mobile Apps, Healthcare, AI/M, Banking System.

2. Iterative Waterfall Model

- Adds **feedback paths** to correct earlier errors.
- Widely used due to its **simplicity** and ease of understanding.
- Best for **well-understood, low-risk** problems.
- **Not suitable** for large or highly risky projects - Smart City, Autonomous Vehicle, National Healthcare Records, Spacecraft Navigation, Online Stock Trading.

3. Prototyping Model

- Ideal when **requirements or technical solutions** are not well understood.
- All **risks must be identifiable** at the beginning.
- Popular for designing **user interfaces**.
- Not Suitable for - Payroll systems, calculator software, Embedded systems, Defense systems, Backend enterprise systems, etc.,

4. Evolutionary Model

- Suitable for **large problems** that can be broken down into **modules**.
- Supports **incremental development and delivery**.
- Commonly used in **object-oriented** development.
- Only applicable if **incremental delivery is acceptable** to the customer.
- Not suitable - Compiler Design, Banking Transaction, Microcontrollers, Navigation and Control Systems, Tax Filing and Regulatory Reporting, etc.,

5. Spiral Model

- A **meta-model** that encompasses all other models.
- Built for **flexibility** and **risk handling**.
- Best for **large, complex, high-risk** projects where risks emerge over time.
- More **complex** than other models, which can deter its use in simple projects.
- Not suitable - Simple CRUD Applications (Create, Read, Update, Delete), Educational or Academic Mini Projects, Time-Critical Projects , Small Utility Tools, Maintenance Projects or Bug Fixes, etc.,

Prototyping vs Spiral Model

- **Prototyping Model:** Use when risks are few and known early.
- **Spiral Model:** Use when risks are high or appear during development.

Customer Perspective

- Initially, **customer confidence** is high across all models.
- As time passes and no working software appears, confidence drops, leading to **frustration**.
- **Evolutionary/incremental models** show working versions early, **maintaining customer trust**.
- Also help customers **adjust gradually** to new software.
- From a financial view, **incremental models** reduce upfront cost — software can be purchased **in parts**.

2.6.1 Selecting a Suitable Model

Factors to Consider:

➤ **Software Characteristics:**

- **Agile model:** Best for small, service-based projects.
- **Iterative waterfall:** Suitable for **product or embedded** software.
- **Evolutionary model:** Preferred for **object-oriented** systems.

➤ **Team Characteristics:**

- **Experienced team:** Can use simpler models like iterative waterfall even for complex systems.
- **Inexperienced team:** May need prototyping even for basic applications.

➤ **Customer Characteristics:**

- **Inexperienced customers:** Tend to **change requirements frequently**.
- In such cases, **prototyping** helps avoid later change requests.

- ❖ **Waterfall (Classical)**: Theoretical, not used in real projects.
- ❖ **Iterative Waterfall**: Simple and common, but limited to well-defined problems.
- ❖ **Prototyping**: For unclear requirements but known risks.
- ❖ **Evolutionary**: For large systems needing gradual delivery.
- ❖ **Spiral**: Most flexible and robust, but complex; best for large, risk-prone projects.