Describe the essence of software engineering practice.

2.3.1 The Essence of Practice

In the classic book, How to Solve It, written before modern computers existed, George Polya [Pol45] outlined the essence of problem solving , and consequently, the essence of software engineering practice:

1. Understand the problem (communication and analysis)

2. Plan a solution (modeling and software design).

3. Carry out the plan (code generation).

4. Examine the result for accuracy (testing and quality assurance).

Understand the problem

It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think, oh yeah, I understand, let's get on with solving this thing. Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

- Who has a stake in the solution to the problem? That is, who are the stakeholders?
- What are the unknowns? What data, functions, and features are required to properly solve the problem?
- Can the problem be compartmentalized? Is it possible to represent smaller problems that may be easier to understand?
- Can the problem be represented graphically? Can an analysis model be created?

Plan the solution

Now you understand the problem (or so you think), and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

- Have you seen similar problems before? Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- Has a similar problem been solved? If so, are elements of the solution reusable?
- Can subproblems be defi ned? If so, are solutions readily apparent for the subproblems?
- Can you represent a solution in a manner that leads to effective implementation? Can a design model be created?

Carry out the plan

The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

- Does the solution conform to the plan? Is source code traceable to the design model?
- Is each component part of the solution provably, correct? Has the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

Examine the result

You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

- Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented?
- Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?

It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

Briefly explain various specialized process models

List various prescriptive process models. Explain any two models in detail.

Waterfall Model

Description: A linear and sequential approach where each phase must be completed before the next begins.

Use Case: Suitable for projects with well-defined requirements and low likelihood of changes.

V-Model (Verification and Validation Model)

Description: An extension of the Waterfall model, with a corresponding testing phase for each development stage.

Use Case: Useful when a high degree of verification is required, such as in safety-critical systems.

Incremental Model

Description: Development and delivery are divided into increments, with each increment adding functionality to the previous release.

Use Case: Ideal for projects where requirements are expected to evolve over time.

Spiral Model

Description: Combines iterative development with systematic aspects of the Waterfall model, emphasizing risk analysis.

Use Case: Suitable for large, complex, and high-risk projects.

Agile Model

Description: Focuses on iterative and incremental development, emphasizing flexibility, customer collaboration, and rapid delivery.

Use Case: Best for projects requiring frequent changes and ongoing stakeholder involvement.

Scrum

Description: An Agile framework that uses fixed-length iterations called sprints, with roles such as Product Owner, Scrum Master, and Development Team.

Use Case: Effective for complex projects with changing requirements.

Extreme Programming (XP)

Description: An Agile methodology emphasizing customer satisfaction, continuous feedback, and flexible response to change.

Use Case: Ideal for projects with dynamic requirements and where frequent releases are needed.

Rapid Application Development (RAD)

Description: Emphasizes quick development and iteration with user feedback, often using component-based construction.

Use Case: Suitable for projects requiring rapid prototyping and user involvement.

DevOps

Description: Integrates development and operations to improve collaboration and productivity by automating infrastructure, workflows, and continuous feedback.

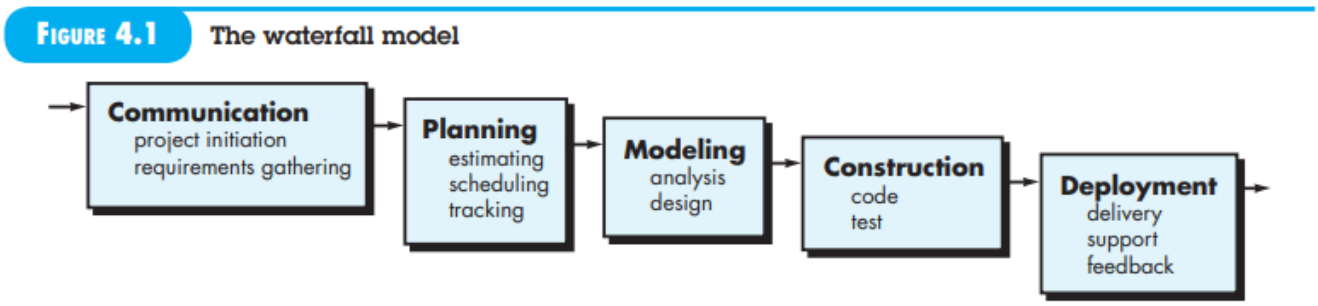
Use Case: Useful for projects needing continuous integration and delivery (CI/CD).

Lean Software Development

Description: Focuses on eliminating waste, optimizing processes, and delivering value to customers quickly.

Use Case: Beneficial for projects requiring efficiency and continuous improvement.

4.1 The Waterfall Model

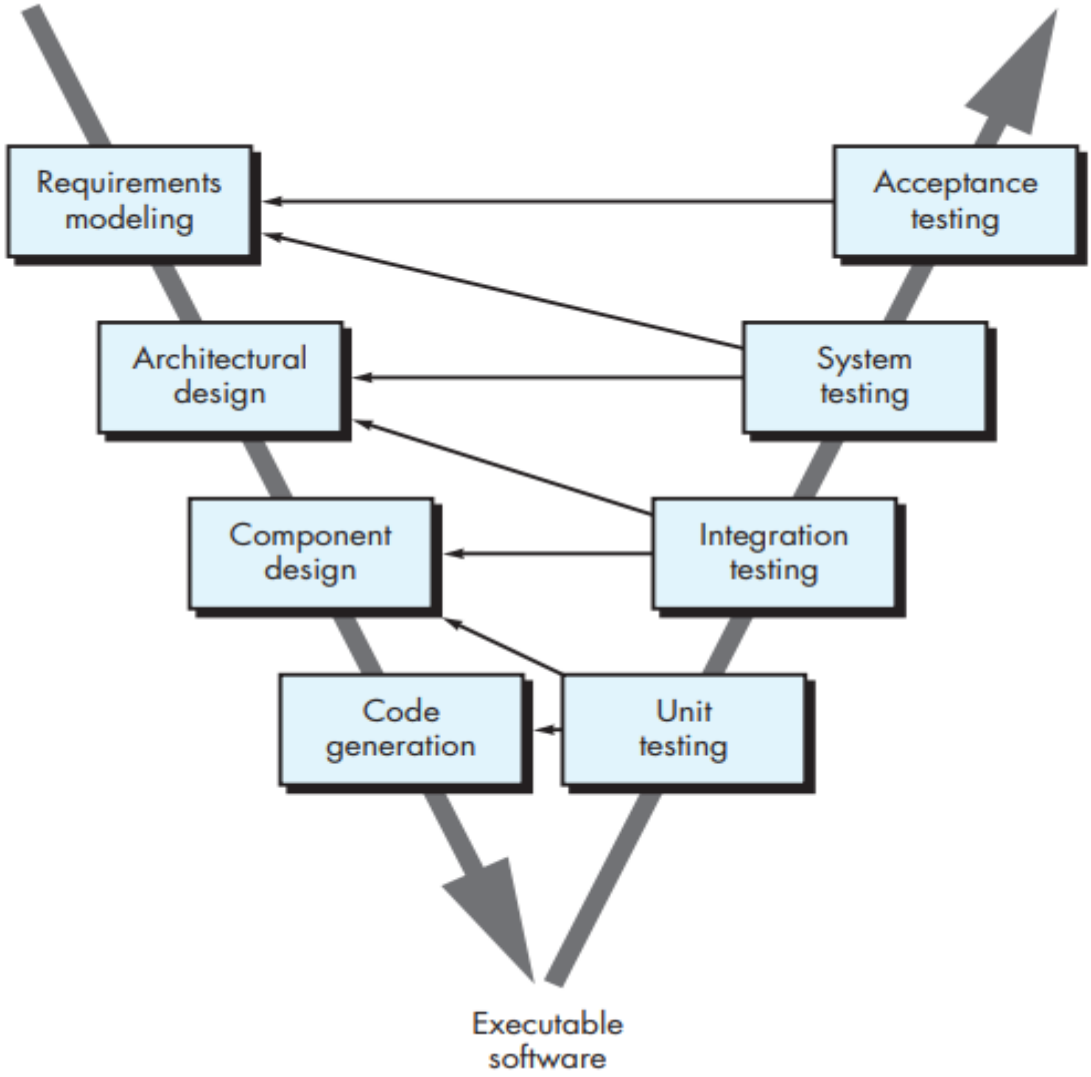


The Waterfall Model, also known as the classic life cycle, is a systematic, sequential approach to software development. It is ideal for situations where the requirements are well understood and relatively stable. This model is sometimes used for well-defined adaptations or enhancements to existing systems, such as changes mandated by new government regulations.

Phases of the Waterfall Model

1. **Communication**
    - **Project Initiation**: Define the project goals and objectives.
    - **Requirements Gathering**: Collect and document all customer requirements.
2. **Planning**
    - **Estimating**: Determine the resources and time required for the project.
    - **Scheduling**: Develop a timeline for the project activities.
    - **Tracking**: Monitor progress against the schedule.
3. **Modeling**
    - **Analysis**: Analyze the requirements to create detailed specifications.
    - **Design**: Develop the system architecture and detailed design.
4. **Construction**
    - **Code**: Write the actual code based on the design documents.
    - **Test**: Conduct unit, integration, and system testing.
5. **Deployment**
    - **Delivery**: Install the software in the production environment.
    - **Support**: Provide ongoing support and maintenance.
    - **Feedback**: Collect feedback from users for future improvements.

4.2 V-Model (Verification and Validation Model)



A variation of the Waterfall Model is the V-Model, which emphasizes the relationship between development stages and corresponding testing phases (Figure 4.2). The V-Model highlights how verification and validation actions are integrated into each phase of the development process. As the development progresses down the left side of the V, requirements are refined into more detailed representations. Moving up the right side involves testing these representations to ensure they meet the requirements.

Phases of the V-Model

1. **Requirements Modeling**:

   - **Description**: Gathering and defining the system requirements from the customer.
   - **Verification**: Acceptance Testing.
   - **Purpose**: Ensures the system meets the customer's requirements.

2. **Architectural Design**:

- **Description**: Defining the overall system architecture and high-level design.
- **Verification**: System Testing.
- **Purpose**: Ensures the complete system works as intended.

3. **Component Design**:

- **Description**: Detailed design of individual components or modules of the system.
- **Verification**: Integration Testing.
- **Purpose**: Ensures the individual components work together as expected.

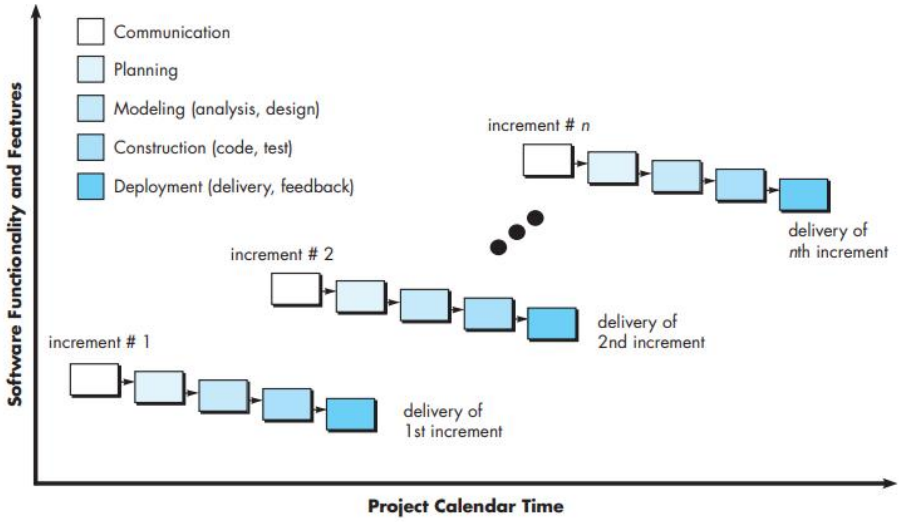
4. **Code Generation**:

- **Description**: Writing the actual code for the system based on the design specifications.
- **Verification**: Unit Testing.
- **Purpose**: Ensures each individual unit or component functions correctly.

5. **Unit Testing**
    - Description: This phase involves testing individual components or units of the software.
    - Corresponding Development Phase: Code Generation.
    - Purpose: To verify that each unit functions correctly according to its design specifications.
    - Activities:
        - Writing and executing test cases for individual units.
        - Debugging and fixing any issues found.
    - Outcome: Verified and functioning code units.
6. **Integration Testing**
    - **Description**: This phase focuses on testing the interaction between integrated units or components.
    - **Corresponding Development Phase**: Component Design.
    - **Purpose**: To ensure that different components of the system work together as intended.
    - **Activities**:
        - Combining units and testing their interactions.
        - Identifying and resolving interface and communication issues.
    - **Outcome**: Integrated components that function correctly together.
7. **System Testing**
    - **Description**: This phase involves testing the complete and
    - integrated system to verify that it meets the specified requirements.
    - **Corresponding Development Phase**: Architectural Design.
    - **Purpose**: To ensure that the entire system functions correctly as a whole.
    - **Activities**:
        - Executing test cases that cover the full system functionality.
        - Performing end-to-end testing to validate system behavior.
    - **Outcome**: A fully integrated system that operates as specified.
8. **Acceptance Testing**
    - **Description**: This phase involves testing the system in the environment in which it will be used to verify that it meets the customer's needs and requirements.
    - **Corresponding Development Phase**: Requirements Modeling.
    - **Purpose**: To ensure that the system meets the user requirements and is ready for deployment.
    - **Activities**:
        - Conducting user acceptance testing (UAT) with real users.
        - Gathering feedback and making necessary adjustments.
    - **Outcome**: A validated system that meets user expectations and requirements.

4.1.2 Incremental Process Models

FIGURE 4.3
The incremental model



The Incremental Process Model is designed for situations where the initial software requirements are reasonably well-defined, but the overall project scope is too large to be handled in a purely linear fashion. This model is also useful when there's a need to deliver a basic version of the software quickly and then add more features over time.

Key Concepts

1. **Combination of Linear and Parallel Processes**:
    - The Incremental Model merges aspects of linear and parallel process flows.
    - It applies linear sequences of development in a staggered manner over time.
2. **Deliverable Increments**:
    - The development is divided into multiple phases or increments.
    - Each increment delivers a specific portion of the software's functionality.

Process

1. **Initial Increment**:
    - The first increment usually delivers a core product.
    - This core product addresses basic requirements but may lack additional features.
2. **Customer Feedback**:
    - The core product is used by the customer or undergoes evaluation.

- Based on this feedback or evaluation, a plan is made for the next increment.

3. **Subsequent Increments**:
   - The next increment refines the core product, adding new features and functionalities.
   - This process is repeated for each subsequent increment.

Example:

- **Word-Processing Software**:
  - **First Increment**: Provides basic file management, editing, and document production functions.
  - **Second Increment**: Adds more advanced editing and document production features.
  - **Third Increment**: Introduces spelling and grammar checking capabilities.
  - **Fourth Increment**: Includes advanced page layout options.

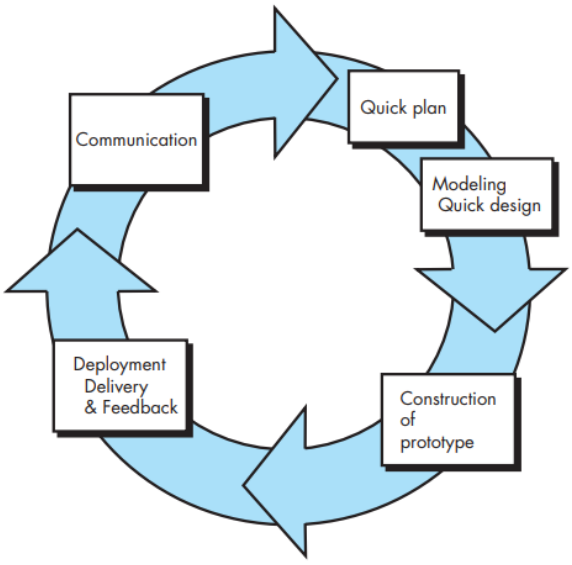
Each increment builds on the previous one, progressively enhancing the software based on user needs and feedback.

Prototyping Integration

- The process flow for any increment may involve prototyping.
- Prototyping allows for early versions of the software to be created, tested, and refined in subsequent increments.

4.1.3 Evolutionary Process Models

FIGURE 4.4
The prototyping paradigm



Overview: Evolutionary process models are designed to handle software development in scenarios where requirements and constraints evolve over time. Unlike linear models, which follow a strict sequence, evolutionary models allow the software to be developed in iterative cycles, producing increasingly refined versions.

Why Use Evolutionary Models?

- **Changing Requirements:** Business needs and user requirements may change during development, making a linear approach impractical.
- **Time Constraints:** Tight deadlines may necessitate delivering a limited version of the software first, with additional features added later.
- **Unclear Details:** Initial requirements might be well understood, but detailed extensions or additional features are not yet defined.

Prototyping Model

What is Prototyping? Prototyping is an iterative development approach where a working model of the software (a prototype) is built and refined based on feedback. It is particularly useful when requirements are vague or when there is uncertainty about certain aspects of the software.

Steps in Prototyping:

1. **Communication:**
    - Start by discussing with stakeholders to understand the overall objectives and any known requirements.
    - Identify areas where more details are needed.
2. **Quick Design:**
    - Create a rapid, simplified design focusing on visible aspects of the software, such as user interface layouts or output formats.
    - This design is not final but serves as a basis for the prototype.
3. **Prototype Construction:**
    - Build a functional version of the software based on the quick design. This prototype is an early version that may not be fully refined.
4. **Deployment and Feedback:**
    - Deploy the prototype to stakeholders for review.
    - Collect feedback on the prototype's functionality, usability, and features.
5. **Iteration:**
    - Use the feedback to make improvements. Refine the prototype through multiple iterations until it meets the needs of stakeholders and addresses any identified issues.

Prototyping Approaches:

- **Throwaway Prototyping:**
    - The prototype is built to explore requirements and is discarded once it has served its purpose. The final system is then developed from scratch based on insights gained.
- **Evolutionary Prototyping:**
    - The prototype evolves into the final product. Over time, it is refined and enhanced

until it becomes the final system. This approach allows for continuous improvement and adaptation based on ongoing feedback.

Challenges of Prototyping:

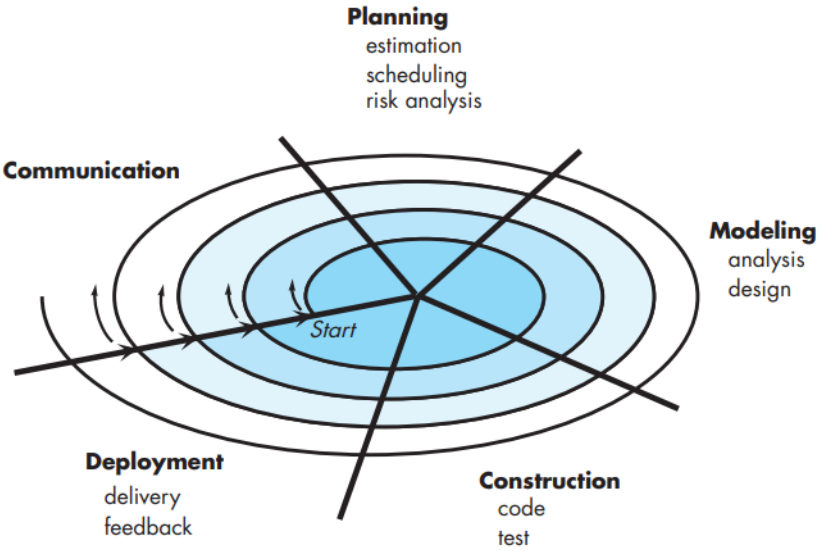
1. **Misconceptions about Prototype Quality:**
    - Stakeholders might think that the prototype is a finished product and may request quick fixes or changes that could lead to quality issues if the prototype is not rebuilt properly.
2. **Implementation Compromises:**
    - To build the prototype quickly, developers might make shortcuts or use less-than-ideal solutions. These compromises can become ingrained in the system if not properly addressed in later stages.

Best Practices for Prototyping:

- **Clarify the Prototype's Purpose:**
    - Ensure all stakeholders understand that the prototype is a tool for discovering and refining requirements. It is not a final product but a draft that will help guide the development of the final system.
- **Focus on Quality:**
    - Resist the temptation to extend a rough prototype into a production product without proper enhancements. Develop the final system with a focus on high quality, performance, and maintainability.

The Spiral Model

FIGURE 4.5
A typical spiral model



The Spiral Model

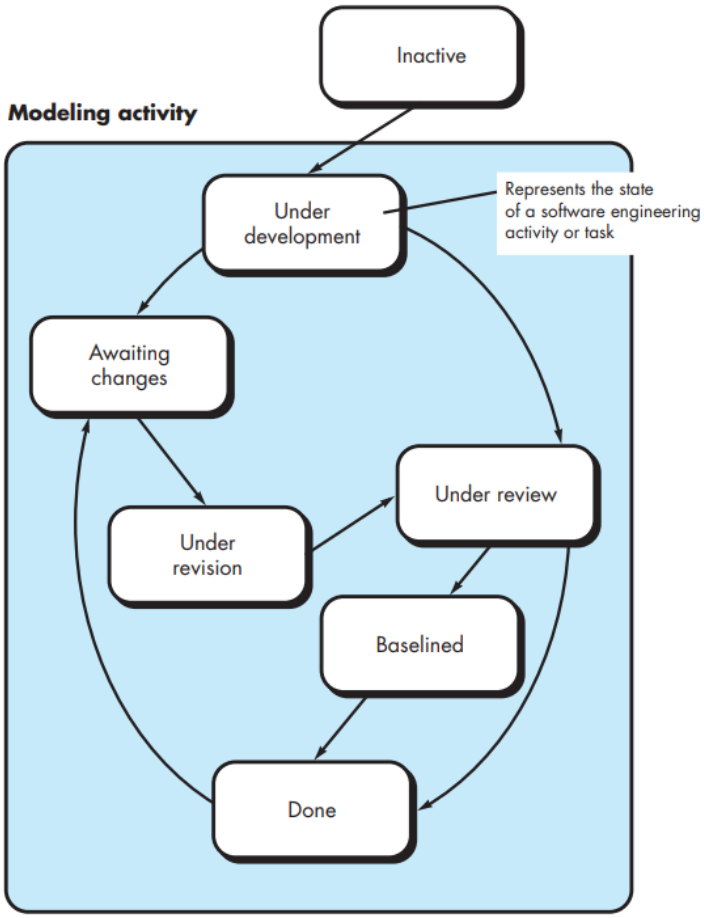
Overview: The Spiral Model is an evolutionary software process model that combines iterative development with systematic planning. It was proposed by Barry Boehm and merges the iterative nature of prototyping with the structured aspects of the waterfall model. It aims to facilitate the rapid development of increasingly complete versions of software.

The Spiral Model, proposed by Barry Boehm, is an evolutionary software process model that combines the iterative nature of prototyping with the systematic aspects of the waterfall model. Here's a summary of the text:

1. **Overview:**
    - The Spiral Model is designed for rapid and iterative development, producing increasingly complete versions of the software.
    - It integrates the cyclical approach of prototyping with structured planning and risk management, providing a framework for handling complex software projects.
2. **Features:**
    - **Risk-Driven:** The model focuses on identifying and mitigating risks at each iteration.
    - **Iterative Releases:** The software is developed in evolutionary releases, starting with early models or prototypes and progressing to more complete versions.
3. **Process Activities:**
    - **Communication:** Discuss objectives and requirements.
    - **Planning:** Develop detailed plans, including cost and schedule estimates.
    - **Modeling:** Create models or prototypes based on the plans.
    - **Construction:** Build the software components.
    - **Deployment:** Deliver the increment to stakeholders and gather feedback.
    - **Feedback and Risk Analysis:** Analyze feedback, reassess risks, and adjust the project plan for the next iteration.
4. **Framework Activities:**
    - The Spiral Model involves several framework activities around a spiral path, with each iteration addressing different aspects of the project:
        - **First Circuit:** Might result in a product specification.
        - **Subsequent Circuits:** Develop prototypes and progressively sophisticated versions of the software.
5. **Adaptability:**
    - The model applies throughout the software's life cycle, from concept development to maintenance.
    - It supports concept development, new product development, and product enhancement, continuing until the software is retired.
6. **Benefits and Challenges:**
    - **Benefits:** The model helps manage large-scale systems and evolving requirements, with a focus on risk reduction and iterative improvement.
    - **Challenges:** It may be difficult to convince customers of its control and manageability. Effective risk assessment and management are crucial for success.

4.1.4 Concurrent Models

FIGURE 4.6
One element of the concurrent process model



The concurrent development model, also known as concurrent engineering, represents a flexible approach to software development by allowing various activities and tasks to occur simultaneously, rather than sequentially. This model can be illustrated using the spiral model as an example, where iterative elements such as prototyping, analysis, and design overlap.

Key Aspects of Concurrent Modeling:

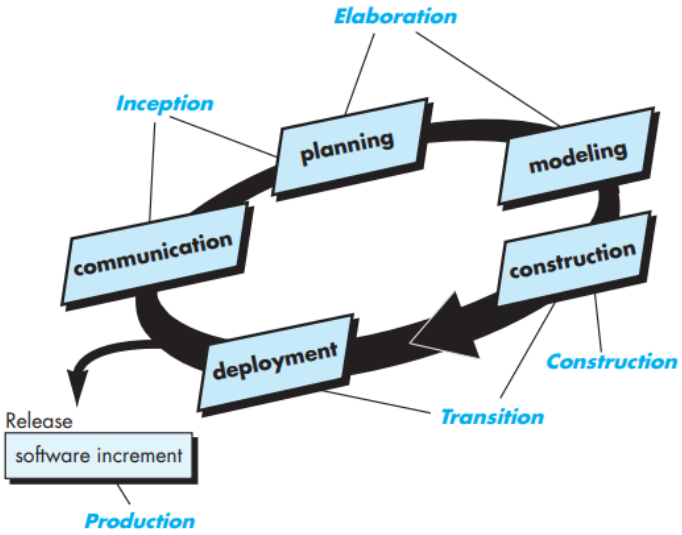
1. **Simultaneity and States:**
    - All software engineering activities, actions, or tasks exist concurrently but can be in different states at any given time. For example, the modeling activity might be in an "under development" state, while communication activities could be in an "awaiting changes" state.
2. **Transitions and Events:**
    - Activities transition between states based on events. For example, if an inconsistency in the requirements is discovered during design, this event triggers a transition in the requirements analysis activity, moving it to a different state (e.g., from "done" to "awaiting changes").

3. **Process Network:**
    - Instead of a linear sequence of events, concurrent modeling defines a process network where activities and tasks are interrelated. Events occurring in one part of the network can affect and trigger changes in other parts.
4. **Applicability:**
    - This approach is applicable to all types of software development and provides a real-time, accurate representation of the project's status by reflecting ongoing and concurrent activities.

4.3 THE UNIFIED PROCESS

4.3.2 Phases of the Unified Process

FIGURE 4.7
The Unified Process



4.4.1 Personal Software Process

PSP is a structured approach that helps individual software engineers improve their performance by providing a disciplined framework for planning, designing, managing, and measuring their work. It's essentially a personal roadmap for software development that emphasizes self-improvement.

The Five Framework Activities of PSP

1. **Planning:** This is the initial phase where the developer outlines the project. It involves:
    - Defining the project's requirements.
    - Estimating the project's size and resource needs.
    - Predicting the number of potential defects (defect estimation).
    - Creating a detailed project schedule.
    - Recording all these details for future analysis.
2. **High-level design:** Here, the developer starts creating a blueprint of the software. It includes:

    - Defining the components of the software.
    - Designing the overall structure of each component.
    - Creating prototypes for uncertain parts of the design.
    - Documenting any issues or challenges encountered.
3. **High-level design review:** This is a quality check point. The developer:
    - Uses formal verification techniques to find errors in the design.
    - Tracks the time spent and the number of errors found.
4. **Development:** This is where the actual coding takes place. It involves:
    - Refining the component-level design.
    - Writing, testing, and debugging the code.
    - Again, tracking time spent and errors found.
5. **Postmortem:** Once the project is complete, the developer analyzes the entire process:
    - Reviewing the collected data on time spent, defects, and other metrics.
    - Identifying areas for improvement.
    - Adjusting the personal process based on the analysis.

Key Benefits of PSP

- **Improved planning and estimation skills:** Developers learn to accurately predict project timelines and resource needs.
- **Enhanced quality:** By focusing on defect prevention and early detection, developers produce higher quality software.
- **Increased productivity:** A well-planned and executed process leads to more efficient development.
- **Personal growth:** PSP encourages self-awareness and continuous improvement.

Provide three examples of software projects that would be amenable to the component-based model. Explain your answer with justification.

1. E-commerce Platform

Justification:

- **Modularity:** E-commerce platforms typically have a variety of features, such as product catalogs, shopping carts, user authentication, and payment processing. These features can be developed as separate components, allowing for independent development, testing, and deployment.
- **Reusability:** Components like payment gateways or user authentication modules can be reused across different e-commerce platforms or different parts of the same platform. This reduces redundancy and speeds up development.
- **Scalability:** As the platform grows, additional components can be added to support new features or integrate with third-party services without disrupting existing functionality.

2. Enterprise Resource Planning (ERP) System

Justification:

- **Modularity:** ERP systems integrate various business processes, such as finance, human resources, supply chain management, and customer relationship management. Each of these modules can be developed as separate components that interact with one another.

- **Reusability:** Components such as financial reporting or inventory management can be adapted for different ERP systems or customized for specific industries.
- **Scalability:** An ERP system can start with core modules and expand over time to include additional functionalities or integrate with other systems, making the component-based model ideal for accommodating growth.

3. Content Management System (CMS)

Justification:

- **Modularity:** CMS platforms include components such as content editors, user management, and media libraries. These components can be developed and updated independently, allowing for easier maintenance and upgrades.
- **Reusability:** CMS components like themes, plugins, and modules can be reused across different websites or content management scenarios, enhancing development efficiency.
- **Scalability:** A CMS can be extended with new components or plugins to add features like SEO tools, analytics, or social media integration, making it adaptable to changing needs.

4.2.1 Component-Based Development

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defi ned interfaces that enable the component to be integrated into the software that is to be built. The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature [Nie92], demanding an iterative approach to the creation of software. However, the componentbased development model comprises applications from prepackaged software components.

Explain the following agile process models: Scrum, DSDM and Agile Modelling.

5.1 WHAT I S AGILITY?

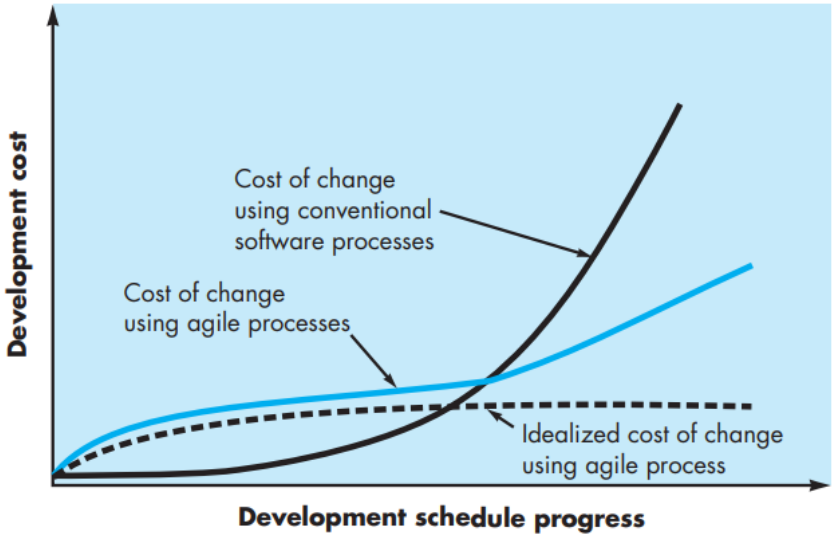
Agility in software engineering refers to the ability of a team and its processes to adapt to changes quickly and effectively. This concept is grounded in the idea that software development is inherently dynamic and subject to frequent changes due to various factors like evolving requirements, new technologies, and shifting team dynamics.

Ivar Jacobson highlights several key aspects of agility:

1. **Response to Change**: Agility is primarily driven by the need to handle changes smoothly. Since software projects often face unexpected changes, agile teams must be able to quickly adjust their processes and goals to accommodate these changes.
2. **Team Collaboration**: Agility emphasizes the importance of collaboration among team members. Effective communication and teamwork are crucial for addressing changes and ensuring the success of the project.
3. **Agile Philosophy**: The Agile Manifesto, which underpins the concept of agility, promotes values such as:
    - Prioritizing working software over comprehensive documentation.
    - Encouraging collaboration between business stakeholders and development teams.
    - Emphasizing flexibility and iterative progress over rigid planning.
4. **Incremental Delivery**: Agile approaches focus on delivering functional software in small, incremental releases. This allows for rapid feedback and continuous improvement based on real-world use and changing requirements.
5. **Flexibility in Planning**: Agile methodologies accept that planning is inherently uncertain and must be adaptable. The project plan should be flexible to accommodate new information and shifting priorities.

5.2 AGILITY AND THE COST OF CHANGE

FIGURE 5.1
Change costs as a function of time in development



The conventional wisdom in software development, supported by decades of experience, is that the cost of change increases nonlinearly as a project progresses. It is relatively easy to accommodate a change when a software team is gathering requirements early in a project. A usage scenario might have to be modified, a list of functions may be extended, or a written specification can be edited. The costs of doing this work are minimal, and the time required will not adversely affect the outcome of the project. But what if we fast-forward a number of months? The team is in the middle of validation testing, something that occurs relatively late in the project, and an important stakeholder is requesting a major functional change. The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs escalate quickly, and the time and cost required to ensure that the change is made without unintended side effects is nontrivial. Proponents of agility, such as Beck and Ambler, argue that a well-designed agile process flattens the cost of change curve, allowing a software team to accommodate changes late in a software project without dramatic cost and time impact. Agile processes encompass incremental delivery, and when this is coupled with other agile practices such as continuous unit testing and pair programming, the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence, as noted by Cockburn, to suggest that a significant reduction in the cost of change can be achieved.

5.3 WHAT I S AN AGILE PROCESS?

An agile software process is characterized in a manner that addresses several key assumptions about the majority of software projects:

1. **Unpredictability of Requirements**: It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds. An agile process reduces the cost of change because software is released in increments, and change can be better controlled within an increment.
2. **Interleaved Design and Construction**: For many types of software, design and construction are interleaved. Both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. **Unpredictable Analysis, Design, Construction, and Testing**: These activities are not as predictable (from a planning point of view) as we might like.

Given these assumptions, the question arises: How do we create a process that can manage unpredictability? The answer lies in process adaptability to rapidly changing project and technical conditions. An agile process, therefore, must be adaptable.

However, continual adaptation without forward progress accomplishes little. Therefore, an agile software process must adapt incrementally. To accomplish incremental adaptation, an agile team requires customer feedback so that the appropriate adaptations can be made. An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an incremental development strategy should be instituted. Software increments (executable prototypes or portions of an operational system) must be delivered in short time periods so that adaptation keeps pace with change (unpredictability). This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.

5.3.1 Agility Principles

The Agile Alliance defines 12 principles for achieving agility in software development:

1. **Customer Satisfaction**: Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. **Embracing Change**: Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. **Frequent Delivery**: Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.
4. **Daily Collaboration**: Business people and developers must work together daily throughout the project.
5. **Motivated Individuals**: Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. **Face-to-Face Communication**: The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. **Working Software as Progress**: Working software is the primary measure of progress. While working software is crucial, it must also exhibit quality attributes such as reliability, usability, and maintainability.

8. **Sustainable Development**: Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. **Technical Excellence**: Continuous attention to technical excellence and good design enhances agility.
10. **Simplicity**: Simplicity—the art of maximizing the amount of work not done—is essential.
11. **Self-Organizing Teams**: The best architectures, requirements, and designs emerge from self-organizing teams.
12. **Reflect and Adjust**: At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

5.3.2 The Politics of Agile Development

The debate over agile software development versus conventional software engineering processes has been intense, sometimes even strident. Jim Highsmith [Hig02a] humorously captures the extremes of this debate. He describes the pro-agility camp ("agilists") as believing that “traditional methodologists are a bunch of stick-in-the-muds who’d rather produce flawless documentation than a working system that meets business needs.” In contrast, he portrays the traditional software engineering camp as seeing “lightweight, er, ‘agile’ methodologists as a bunch of glorified hackers who are going to be in for a heck of a surprise when they try to scale up their toys into enterprise-wide software.”

As with many technology debates, this discussion risks devolving into a "religious war," where entrenched beliefs overshadow rational thought and objective facts. The real issue is not whether agility is beneficial—there is widespread agreement on its value—but rather the best way to achieve it. Additionally, how can software be built to meet current customer needs while also being extendable and scalable to address future needs?

There are no absolute answers to these questions. Within the agile community itself, various process models (as detailed in Section 5.4) offer different approaches to achieving agility. Each model introduces concepts that mark a departure from traditional software engineering practices. However, many agile concepts are adaptations of established good practices in software engineering.

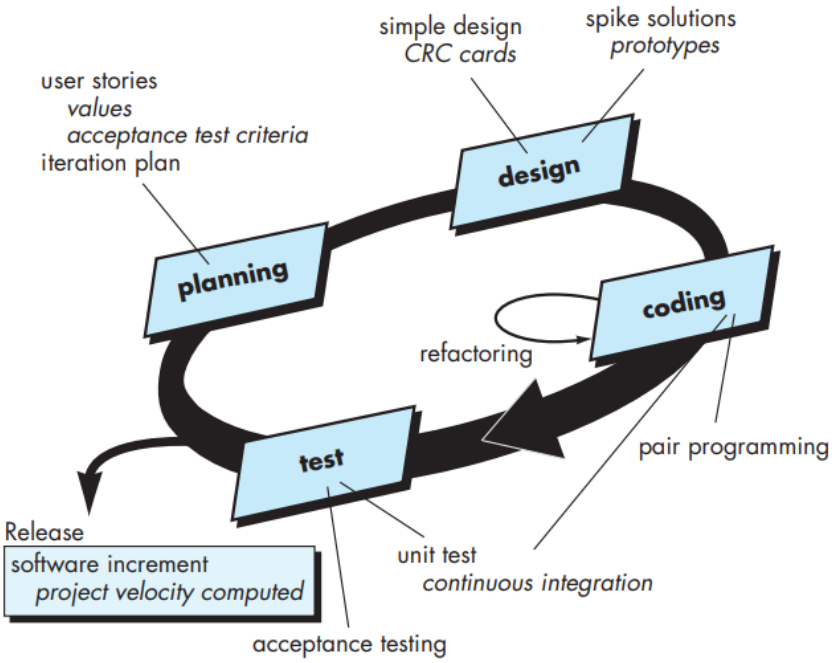
Ultimately, it is not necessary to choose between agility and traditional software engineering. Instead, it is possible to define a software engineering approach that incorporates agility. For those interested in exploring these issues further, see [Hig01], [Hig02a], and [DeM02] for an engaging overview of other significant technical and political aspects.

5.4 EXTREME PROGRAMMING

In order to illustrate an agile process in a bit more detail, we'll provide you with an overview of Extreme Programming (XP), the most widely used approach to agile software development. Although early work on the ideas and methods associated with XP occurred during the late 1980s, the seminal work on the subject has been written by Kent Beck [Bec04a]. A variant of XP, called Industrial XP (IXP), refi nes XP and targets the agile process specifi cally for use within large organizations [Ker05].

5.4.1 The XP Process

FIGURE 5.2
The Extreme Programming process



Extreme Programming (XP) employs an object-oriented approach and integrates a set of rules and practices within four primary framework activities: planning, design, coding, and testing. The XP process is illustrated in Figure 5.2, which highlights key ideas and tasks associated with each activity. Here is a summary of the key XP activities:

Planning: The planning phase, also known as the planning game, begins with requirements gathering through a listening process. This helps the XP team understand the business context and major features required for the software. The requirements are then captured as "stories" (user stories) on index cards, each assigned a priority value by the customer based on business value. The XP team estimates the development cost of each story in weeks. If a story is too large (i.e., estimated to take more than three weeks), it is split into smaller stories. New stories can be added at any time. The team then groups stories into the next release and decides their order based on one of three strategies: implementing all stories immediately, prioritizing stories with the highest value, or tackling the riskiest stories first. After delivering the first release, the XP team calculates project velocity— the number of stories implemented—which helps

in estimating delivery dates and assessing any overcommitment. Adjustments are made as needed based on new or modified stories.

Design: XP emphasizes simplicity with the Keep It Simple (KIS) principle. The design should be straightforward and only cover what is necessary for each story. The use of CRC (class-responsibility-collaborator) cards is encouraged to organize object-oriented classes relevant to the current increment. Difficult design issues are addressed with spike solutions—prototypes that help reduce risk and validate estimates. Refactoring is a key practice in XP, where the code is continuously improved without altering its external behavior. Although XP minimizes design documentation, refactoring helps maintain a clean and efficient codebase. Continuous design modifications occur as construction progresses.

Coding: Coding in XP starts after developing unit tests for each story. These tests guide the implementation, ensuring that the code meets the required functionality. Pair programming is a central practice where two developers work together at one workstation. This approach enhances real-time problem solving, quality assurance, and adherence to coding standards. As code is completed, it is integrated with other code on a daily basis, either by an integration team or by the pair programmers themselves. This strategy helps prevent compatibility issues and supports continuous integration.

Testing: XP uses automated testing frameworks to facilitate regression testing whenever code is modified. A universal testing suite allows for daily integration and validation testing, providing continuous feedback on progress and early detection of issues. Acceptance tests, specified by the customer, focus on overall system features and functionality that are visible and reviewable by the customer. These tests are derived from user stories and help ensure that the software meets customer expectations.

5.4.2 Industrial XP

Industrial Extreme Programming (IXP) is an evolved version of Extreme Programming (XP) that adapts XP's principles to better suit large-scale projects and organizations. As described by Joshua Kerievsky, IXP retains XP’s core values but introduces several new practices and modifies existing ones to improve its applicability to significant projects. Here are the six new practices introduced in IXP:

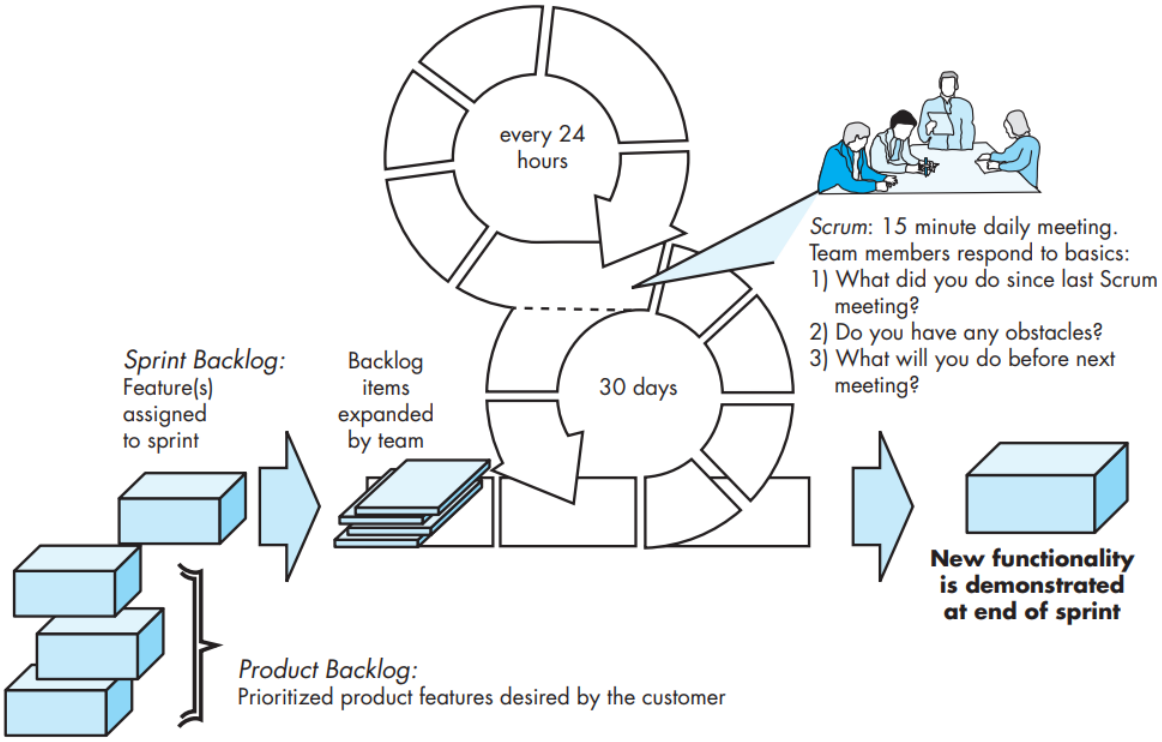
1. **Readiness Assessment:** This practice ensures that all members of the project community (stakeholders, developers, management) are prepared for the project. It involves confirming that the proper environment is established and that team members possess the necessary skills and understanding.
2. **Project Community:** IXP emphasizes the importance of having the right people with the appropriate skills and training involved in the project. This “community” includes both technologists and other stakeholders who are essential for project success.

3. **Project Chartering:** This practice assesses whether the project has a valid business justification and aligns with the organization's overall goals and objectives. It ensures that the project will provide value and contribute to the strategic objectives of the organization.
4. **Test-Driven Management:** IXP introduces the concept of defining measurable "destinations" to assess progress. Mechanisms are established to determine whether these destinations have been achieved, ensuring that the project stays on track and meets its goals.
5. **Retrospectives:** After delivering a software increment, IXP conducts a specialized technical review called a retrospective. This review examines issues, events, and lessons learned across the increment or entire software release to improve future processes and practices.
6. **Continuous Learning:** IXP encourages and possibly incentivizes the team to continually learn new methods and techniques that can enhance the quality of the product. This practice helps teams stay updated with evolving best practices and technologies.

5.5 OTHER AGILE PROCESS MODELS

5.5.1 Scrum

FIGURE 5.3 Scrum process flow



Scrum is an agile software development methodology that emerged in the early 1990s, initially developed by Jeff Sutherland and later refined by Schwaber and Beedle. It emphasizes iterative progress through short, time-boxed development cycles known as sprints, typically lasting 30 days or less. The Scrum process revolves around several key components:

1. **Backlog**: A prioritized list of project requirements or features that deliver business value. This list is continuously updated by the product manager as new items are added and priorities shift.
2. **Sprints**: Fixed-duration work cycles within which specific backlog items are addressed. Changes to the backlog are not permitted during a sprint, ensuring a stable work environment.
3. **Scrum Meetings**: Daily 15-minute stand-up meetings where team members discuss:
    - What they accomplished since the last meeting
    - Any obstacles they are facing
    - What they plan to do before the next meeting

    The Scrum master facilitates these meetings to uncover potential issues early and promote team cohesion.

4. **Demos**: At the end of each sprint, the team presents the completed work to the customer. This allows the customer to review and evaluate the delivered functionality, though it may not include all planned features.

Scrum is designed to handle tight timelines and evolving requirements, leveraging these structured patterns to manage uncertainty and enable successful project outcomes. The overall flow of Scrum includes defining requirements, analysing, designing, evolving, and delivering in a cyclical manner.

5.5.2 Dynamic Systems Development Method

The Dynamic Systems Development Method (DSDM) is an agile software development approach focused on delivering systems within tight time constraints through incremental prototyping. It is based on a modified Pareto principle, where 80% of an application can be delivered in 20% of the time it would take to complete the full application.

Key Aspects of DSDM:

1. **Iterative Process**: DSDM uses iterative cycles, with each iteration aimed at delivering an increment that provides partial but functional value. The focus is on delivering enough to move to the next increment, with further details addressed as more requirements become known.
2. **DSDM Life Cycle**: The DSDM process consists of several stages:
    - **Feasibility Study**: Establishes basic business requirements and constraints.
    - **Business Study**: Identifies functional and information requirements.
    - **Functional Model Iteration**: Develops incremental prototypes to demonstrate functionality. Feedback from users is collected to refine requirements.

- **Design and Build Iteration**: Refines prototypes to ensure they are engineered to deliver operational business value. This stage may overlap with the functional model iteration.
- **Implementation**: Deploys the latest software increment into the operational environment. The increment might not be fully complete, and further changes may be requested, leading back to the functional model iteration.

3. **Combining with XP**: DSDM can be integrated with Extreme Programming (XP) to blend a robust process model (DSDM) with the specific practices required to build software increments effectively.

5.5.3 Agile Modelling

Agile Modeling (AM) is a practice-based methodology designed to provide effective, lightweight modeling and documentation of software systems. It emphasizes flexibility, simplicity, and relevance over traditional, more rigid modeling approaches. Key principles and practices of Agile Modeling include:

1. **Model with a Purpose**: Each model should have a specific goal, such as communicating information to stakeholders or understanding a particular aspect of the system. This ensures that the model serves a clear function and uses appropriate notation and detail.
2. **Use Multiple Models**: Different models and notations can describe various aspects of a system. Agile Modeling advocates using only those models that provide value to their intended audience, rather than creating exhaustive or unnecessary models.
3. **Travel Light**: Maintain only those models that offer long-term value. This principle advocates for discarding models that no longer contribute to the project, to avoid slowing the team down with maintenance work.
4. **Content Over Representation**: The value of a model lies in the content it provides, not in its syntactic perfection. A model that delivers useful insights, even if imperfectly represented, is more valuable than a flawlessly formatted model that lacks substantive content.

5. **Know Your Models and Tools**: Understanding the strengths and weaknesses of both the models and the tools used to create them is crucial. This knowledge allows for better adaptation and application of modeling practices.
6. **Adapt Locally**: The modeling approach should be tailored to fit the specific needs of the agile team and the project context. This flexibility ensures that modeling practices align with the team's workflow and project goals.

Unified modelling Language (UML) is a widely adopted method for representing analysis and design models in software engineering. Scott Ambler has developed a simplified version of the Unified Process (UP) that integrates Agile modelling principles, aiming to combine effective modelling with agile practices.

5.5.4 Agile Unified Process

The Agile Unified Process (AUP) is an agile adaptation of the classic Unified Process (UP) methodology, blending traditional phased activities with iterative development. It aims to offer a structured approach while maintaining flexibility and rapid delivery. Here's a breakdown of AUP:

Key Principles:

- **"Serial in the Large, Iterative in the Small"**: AUP follows a sequential process at a high level, with phases such as inception, elaboration, construction, and transition. Within each phase, iterative cycles are employed to deliver software increments.

Core Activities in AUP Iterations:

1. **Modeling**: Develop UML models to represent the business and problem domains. These models should be "just barely good enough" to facilitate progress without becoming a bottleneck.
2. **Implementation**: Convert models into source code, focusing on incremental development.
3. **Testing**: Design and execute tests to identify and fix errors, ensuring that the code meets its requirements. This aspect aligns with practices from Extreme Programming (XP).
4. **Deployment**: Deliver software increments to end users and gather feedback. This is crucial for validating the functionality and making necessary adjustments.
5. **Configuration and Project Management**: Manage change, risks, and persistent work products (e.g., models, documents). Track and control the team's progress and coordinate activities.
6. **Environment Management**: Oversee the process infrastructure, including standards, tools, and support technologies, to ensure the team operates efficiently.

Connection with UML:

While AUP incorporates UML modeling, it is designed to be compatible with various agile processes. UML can be used with other agile methodologies to support modeling and documentation needs.

For further details and resources on AUP, you might find it helpful to refer to Scott Ambler’s work and the broader agile community.

2.2.1 The Process Framework

A *process framework* establishes the foundation for a complete software engineering process by identifying a small number of *framework activities* that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of *umbrella activities* that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

Communication. Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders).[3] The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning. Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a software project plan—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling. Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction. What you design must be built. This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment. The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of

2.2.2 Umbrella Activities

Software engineering process framework activities are complemented by a number of *umbrella activities*. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

Software project tracking and control—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance—defines and conducts the activities required to ensure software quality.

Technical reviews—assess software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management—manages the effects of change throughout the software process.

Reusability management—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

Work product preparation and production—encompass the activities required to create work products such as models, documents, logs, forms, and lists.