SUBJECT: Software Engineering & Project Management (BCS501)

CHAPTER 3 – AGILE DEVELOPEMENT

Syllabus: *Agile Development: What is Agility?, Agility and the cost of change. What is an agile Process?, Extreme Programming (XP), Other Agile Process Models, A tool set for Agile process .*

Principles that guide practice: Software Engineering Knowledge, Core principles, Principles that guide each framework activity.

INTRODUCTION

What is Agile?

Definition: Agile is a time boxed, iterative approach to software deliver that builds software incrementally from the start of the project, instead trying to deliver it all at once near the end.

- It works by breaking projects down into little bits of user functionality called user stories, prioritizing them, and then continuously delivering then in to short two cycles called iterations.
- It encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity

Who does it: Software engineers and other project stakeholders (managers, customers, end users) work together on an agile team—a team that is self-organizing and in control of its own destiny

How does it work?

- **Make a list:** Sitting down with customer, make a list of features they would like to see in their software, these are called user stories and they become the **TO DO** list for the project.
- **Size things up:** Size (estimate) stories relatively to each other, coming up with a guess as to how long you think each user story will take.
- **Set some priorities:** Like some lists, there always seems to be more to do than time allows. So ask ustomer to prioritize their list to get the most important stuff done first, and save the least important for last.
- **Start executing:** Then start delivering some value, start at the top. Work your way to the bottom. Building, iterating, and getting feedback from your customer as you go.
- **Update the plan as you go :** Then as you and your customer start delivering one of two things is going to happen .You will discover.
  1) **You're going fast enough. All is good OR**
  2) **You have too much to do and not enough time.**

SVIT

Why to use AGILE Methods

- Improves Customer involvement.
- Increase quality.
- Simplify releases.
- Drive down Risk.

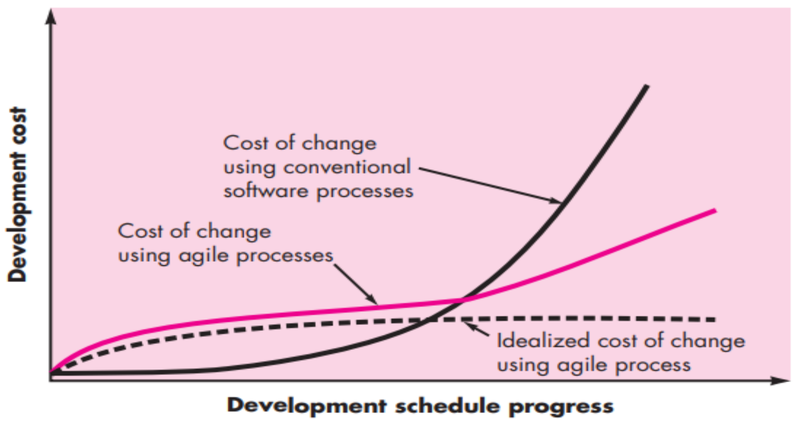
Advantages of agile model

- Customer satisfaction by rapid, continuously deliver of useful software.
- People and interactions are emphasized rather than process and tools.
- Customer, developers and testers constantly interact with each other.
- Working software is delivered frequently(weeks rather than months).
- Face to face communication is the best form of communication.
- Close, daily cooperation between business people and developers.
- Continuo’s attention to technical excellence and good design.
- Regular adaptation to changing circumstances.
- Even late changes in requirements are welcomed.

Agility and cost of change

- **Agility** refers to the **Agile methodology**, which is a set of principles and practices for software development aimed at delivering high-quality software quickly and efficiently while being responsive to changing requirements.

The graph below illustrates the concept of the "cost of change" in software development as a project progresses over time, contrasting conventional and agile development processes.



1. **Conventional Software Process (Black Curve):** In traditional, sequential development models like the Waterfall model, the cost of implementing changes increases steeply as the project progresses. Early in the project, changes are relatively inexpensive since they mainly affect the planning or initial design. However, as development moves toward testing and deployment, changes become much more complex and costly, often impacting multiple components, tests, and dependencies.
2. **Agile Process (Pink Curve):** Agile methodologies aim to keep the cost of change more manageable, even as the project progresses. Agile practices, such as incremental delivery, continuous testing, and regular feedback, help teams to accommodate changes with less disruption. This "flattens" the cost curve, reducing the steep increase seen in traditional

VIT

approaches. While the cost of change still rises over time, it is more gradual compared to the conventional curve.

3. I**<u>dealized Agile Cost Curve (Dashed Black Line):</u>** This line represents an idealized version of agile development where the cost of change remains almost constant throughout the project. While achieving a completely flat cost curve is challenging, this illustrates the agile principle of maintaining flexibility and responsiveness to change, minimizing costs related to late-stage modifications.

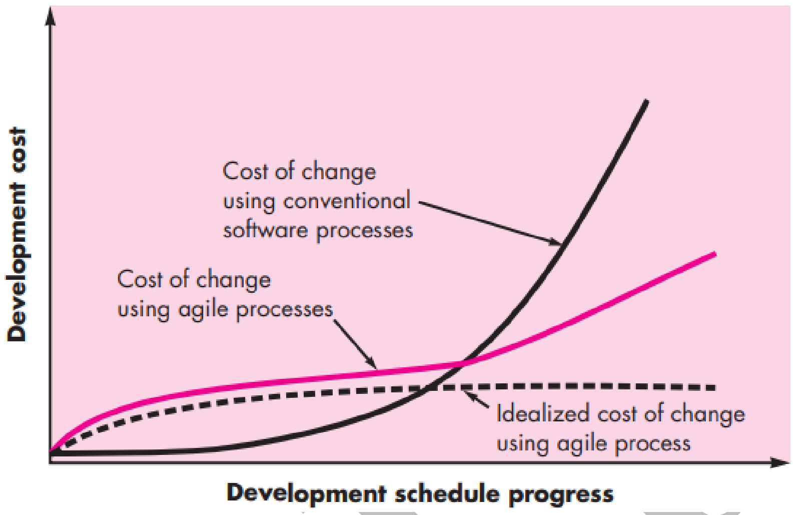


Figure: Change costs as a function of time in development.

<u>Agile Process</u>

Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects

1) 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.
2) For many types of software, design and construction are interleaved. That is, 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) Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

An agile process, therefore, must be adaptable:

- Software process must adapt incrementally.
- To accomplish incremental adaptation, an agile team requires customer feedback (so that the appropriate adaptations can be made).
- 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

SVIT

Agility Principles

The Agile defines 12 agility principles for those who want to achieve agility:

1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4) Business people and developers must work together daily throughout the project.

5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done

6) The most efficient and effective method of conveying information to and within a development team is face-to-face conversation

7) Working software is the primary measure of progress

8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely

9) Continuous attention to technical excellence and good design enhances agility

10) Simplicity—the art of maximizing the amount of work not done—is essential

11) The best architectures, requirements, and designs emerge from self– organizing teams.

12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

SVIT

Extreme Programming

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities.

1) Planning
2) Design
3) Coding
4) Testing

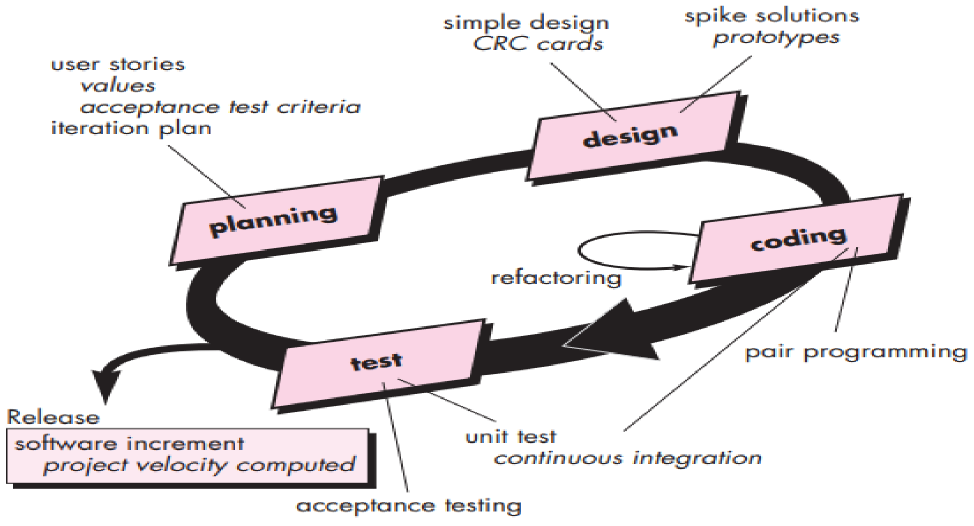


Figure: <u>Extreme Programming Process</u>

1) **Planning:** The planning activity also called the planning game begins with listening a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software.

- Listening leads to the creation of a set of "stories" also called user stories that describe required output, features, and functionality for software to be built.
- Each is written by the customer and is placed on an **index card.**
- The customer assigns a priority to the story based on the overall business value of the feature or function.
- Members of the XP team then assess each story and assign a cost measured in development week to it.
- If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again.
- It is important to note that new stories can be written at any time.
- Customers and developers work together to decide how to group stories into the next release the next software increment to be developed by the XP team.
- Once a basic commitment agreement on stories to be included, delivery date, and other project matters is made for a release, the XP team orders the stories that will be developed in one of three ways: (1) all stories will be implemented immediately (within a few weeks), (2) the stories with highest value will be moved up in the schedule and implemented first, or (3) the riskiest stories will be moved up in the schedule and implemented first.

- After the first project release also called a software increment has been delivered, the XP team computes **project velocity**.
  Project velocity: is the number of customer stories implemented during the first release. Project velocity can then be used to (1) help estimate delivery dates and schedule for subsequent releases and (2) determine whether an over commitment has been made for all stories across the entire development project. If an over commitment occurs, the content of releases is modified or end delivery dates are changed.

2) **Design:** XP design rigorously follows the KIS (keep it simple) principle
   - A simple design is always preferred over a more complex representation
   - The design provides implementation guidance for a story as it is written—nothing less, nothing more.

SVIT

- XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context.
- **CRC (class-responsibilitycollaborator)** cards identify and organize the object-oriented classes7 that are relevant to the current software increment.
- XP recommends the immediate creation of an operational prototype of that portion of the design. Called a **spike solution**, the design prototype is implemented and evaluated.

3) **Coding:** A key concept during the coding activity (and one of the most talked about aspects of XP) is pair programming.
   - **Pair Programming:** XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for realtime problem solving (two heads are often better than one) and real-time quality assurance (the code is reviewed as it is created). It also keeps the developers focused on the problem at hand.
   - As pair programmers complete their work, the code they develop is integrated with the work of others
   - The pair programmers have integration responsibility. This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment) that helps to uncover errors early.

4) **Testing:** creation of unit tests before coding commences is a key element of the XP approach.
   - The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and repeatedly).
   - This encourages a regression testing strategy whenever code is modified.
   - As the individual unit tests are organized into a "universal testing suite" [Wel99], integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go away

Industrial XP

Industrial Extreme Programming (IXP) in the following manner: "IXP is an organic evolution of XP. IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

1) **Readiness assessment**: Prior to the initiation of an IXP project, the organization should conduct a readiness assessment. The assessment ascertains whether
   (a) An appropriate development environment exists to support IXP, (b) the team will be populated by the proper set of stakeholders, (c) the organization has a distinct quality program and supports continuous improvement, (d) the organizational culture will support the new values of an agile team, and (e) the broader project community will be populated appropriately.

2) **Project community:** XP suggests that the right people be used to populate the agile team to ensure success.
   - The implication is that people on the team must be well-trained, adaptable and skilled, and have the proper temperament to contribute to a self-organizing team.
   - A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders (e.g., legal staff, quality auditors, manufacturing or sales types) who "are often at the periphery of an IXP project yet they may play important roles on the project".

SVIT

3) **Project chartering:** The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization.

4) **Test-driven management:** An IXP project requires measurable criteria for assessing the state of the project and the progress that has been made to date. Test-driven management establishes a series of measurable "destinations" [Ker05] and then defines mechanisms for determining whether or not these destinations have been reached.

5) **Retrospectives:** An IXP team conducts a specialized technical review after a software increment is delivered. Called a retrospective, the review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release. The intent is to improve the IXP process.

6) **Continuous learning:** Because learning is a vital part of continuous process improvement, members of the XP team are encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher quality product.

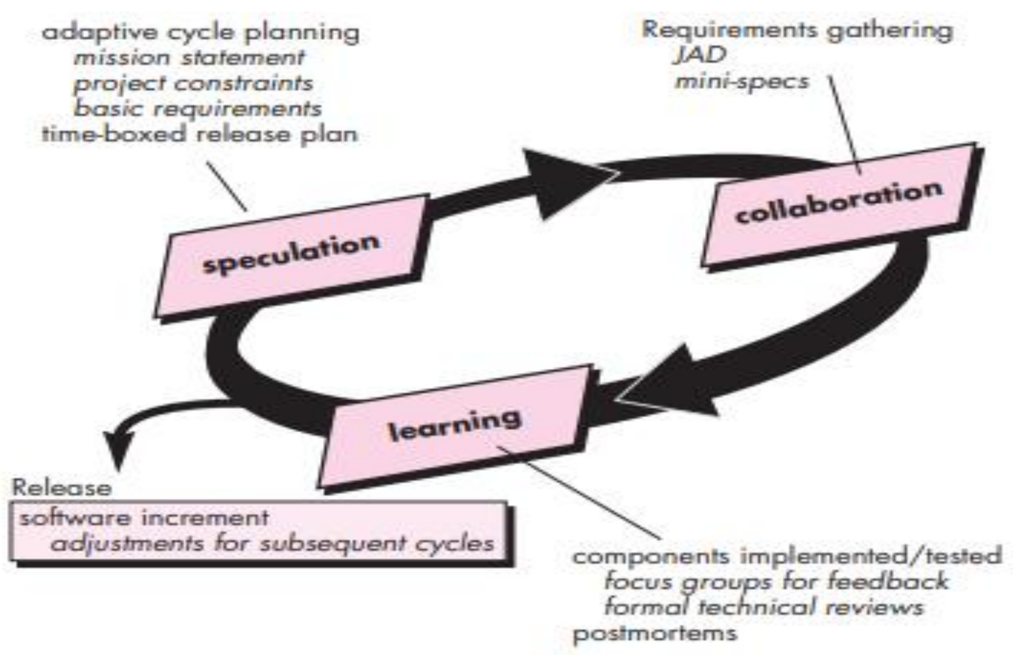
Other Agile Models

1. **Adaptive Software Development (ASD)**
2. **Scrum**
3. **Dynamic Systems Development Method (DSDM)**
4. **Crystal**
5. **Feature Drive Development (FDD)**
6. **Lean Software Development (LSD)**
7. **Agile Modeling (AM)**
8. **Agile Unified Process (AUP)**

Adaptive Software Development (ASD)

Adaptive Software Development (ASD) is an iterative and incremental approach to software development designed to handle complex, uncertain projects.
ASD is rooted in agile principles and emphasizes flexibility, collaboration, and learning as key elements for project success.
Its primary goal is to enable teams to adapt continuously to changing requirements rather than following a fixed plan.

SVIT



ASD is structured around a cycle with three main phases:

Speculate

- This phase replaces traditional planning.
- Instead of defining rigid requirements, teams focus on defining objectives, exploring possibilities, and setting the initial direction.
- It's more about hypothesizing and adjusting as new information surfaces rather than creating a strict blueprint for development.

Collaborate

- ASD promotes a high level of collaboration among team members, stakeholders, and customers.
- The emphasis is on teamwork, direct communication, and shared responsibility.
- Teams work together closely to achieve common goals and adjust based on feedback and insights gained throughout development.

Learn

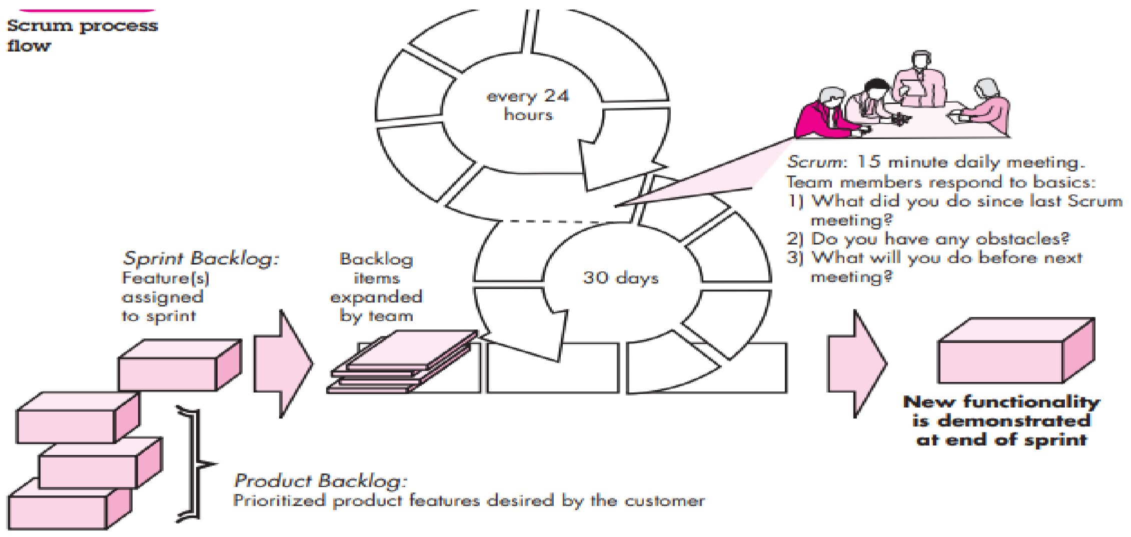
- ASD treats the development process as a continuous learning experience.
- After each cycle, the team reflects on the outcomes, collects feedback, and makes necessary adjustments.
- This phase encourages experimentation and adaptation based on real-world insights.

ASD is well-suited for projects where requirements are uncertain or likely to change, making it valuable in environments like research and development, startups, or any scenario where adaptability is crucial.

SVIT

Scrum

Scrum is a widely-used agile framework designed to help teams work collaboratively on complex projects. Scrum's structure is influenced by agile principles and is characterized by iterative development, adaptability, and regular feedback.


Scrum process flow
every 24 hours
30 days
Scrum: 15 minute daily meeting. Team members respond to basics: 1) What did you do since last Scrum meeting? 2) Do you have any obstacles? 3) What will you do before next meeting?
Sprint Backlog: Feature(s) assigned to sprint
Backlog items expanded by team
New functionality is demonstrated at end of sprint
Product Backlog: Prioritized product features desired by the customer

Core Elements of the Scrum Framework

Backlog

- The backlog is a dynamic, prioritized list of project requirements and features that add business value for the customer.
- It's managed by the Product Owner, who regularly updates priorities. New items can be added at any time, introducing flexibility to adapt to changing requirements.

Sprints

- These are short, time-boxed periods typically around 2-4 weeks in which a specific set of tasks from the backlog is completed.
- During a sprint, no new changes are introduced, allowing team members to focus on a stable set of objectives.Each sprint results in a potentially deliverable increment of the product.

Daily Scrum Meetings:

These are brief about 15 minute's daily check-ins where the Scrum team addresses three key questions:

1. What did you accomplish since the last meeting?
2. What obstacles are you encountering?
3. What do you aim to complete by the next meeting?

The Scrum Master leads these meetings to ensure smooth progress and early identification of issues. These daily scrums foster transparency and promote a self-organizing team environment.

Sprint Review (Demo)

- At the end of each sprint, the team demonstrates the completed features to stakeholders, allowing for immediate feedback.
- This demonstration may not showcase the entire planned functionality but focuses on what was accomplished during the sprint.
- Feedback from the review is used to improve the backlog and plan for the next sprint

SVIT

Dynamic Systems Development Method (DSDM)

The Dynamic Systems Development Method (DSDM) is an agile framework that focuses on delivering high-priority functionality quickly through incremental prototyping. This approach aligns with the 80/20 rule, or Pareto principle, which in DSDM suggests that 80% of an application's key functionality can be delivered in 20% of the time it would take to complete the entire application.
Its phases and components are:

Feasibility Study: This phase identifies the basic business needs and constraints of the project, assessing whether it is suitable for the DSDM approach.

Business Study: Focuses on defining the application's functional and informational requirements to ensure business value. It also outlines the preliminary architecture and maintainability needs.

Functional Model Iteration: Develops incremental prototypes showcasing functionality. Users interact with these prototypes, providing feedback to guide further development and refine requirements.

Design and Build Iteration: Revisits prototypes to ensure they are engineered to meet operational standards. This phase may overlap with the Functional Model Iteration, emphasizing concurrent development.

Implementation: Deploys the latest prototype increment into the operational environment. As DSDM is iterative, any requested changes or uncompleted tasks loop back to previous stages.

Crystal

The Crystal family of agile methods, created by Alistair Cockburn and Jim Highsmith, aims to provide a flexible approach to software development that emphasizes "maneuverability."

Maneuverability refers to the ability of a team or process to quickly adapt and adjust to changing requirements, challenges, or feedback throughout the software development lifecycle.

Cockburn describes this as a "resource-limited, cooperative game of invention and communication," with a primary goal of delivering functional, valuable software, and a secondary goal of preparing effectively for future work. It emphasizes flexibility and responsiveness, allowing the team to shift direction as needed to better meet the evolving needs of the project or the client.

SVIT

Feature Driven Development (FDD)

Feature Driven Development (FDD), initially developed by Peter Coad and colleagues, is an agile model designed specifically for object-oriented software engineering.FDD as a method for developing software through manageable, client-valued functions or "features."

FDD is particularly well-suited for moderately sized and large projects, providing a structured yet adaptive framework that emphasizes.Key Principles: collaboration, feature-based decomposition, and clear communication.

FDD, a feature is defined as a "client-valued function that can be implemented in two weeks or less.
FDD template for feature definition is structured as:

<action> the <result> <by for of to> a(n) <object>

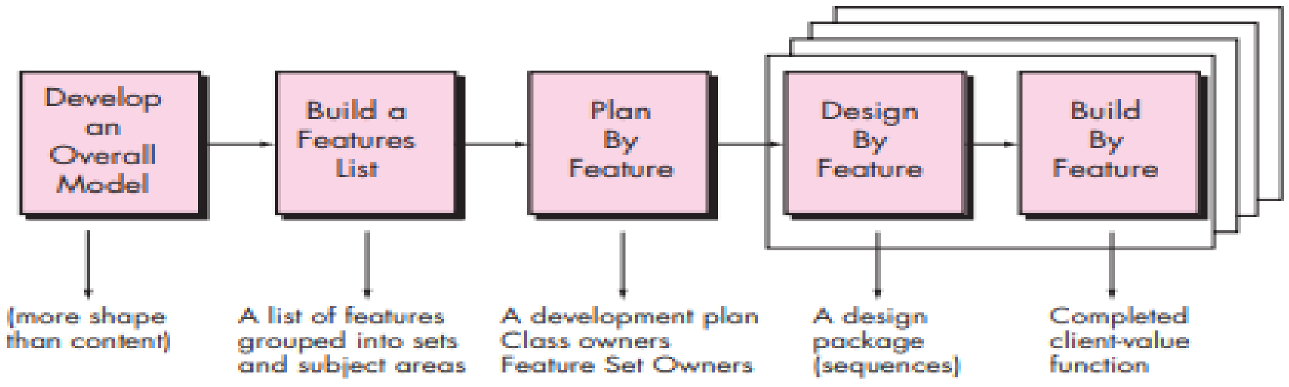
<action>: The verb describing what will be done Example: "create," "authorize".
<result>: The outcome expected from the action Example: "payment," "account setup".
<by for of to>: A connecting preposition that gives additional context.

<object>: The target or entity involved in the feature, such as a person, place, or thing.
For example: Authorize the payment for an order.

Framework Activities in FDD

FDD comprises five key activities or "processes":

1. Develop an overall model.
2. Build a feature list.
3. Plan by feature.
4. Design by feature.
5. Build by feature.


Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
A list of features grouped into sets and subject areas
A development plan Class owners Feature Set Owners
A design package (sequences)
Completed client-value function

<u>Develop an Overall Model</u>

To create a high-level understanding of the system, capturing the shape and scope of the project. This model gives general insights but is not heavily detailed.

<u>Build a Features List</u>

To identify and document all the features required in the system. Features are grouped into sets and subject areas based on functionality. This list organizes the work in terms of client-valued functions.

<u>Plan By Feature</u>

To create a development plan that includes assigning ownership. The development team assigns class owners and feature set owners, organizing the work and assigning responsibilities for efficient execution.

<u>Design By Feature</u>

To design each feature in detail before implementation.

<u>Build By Feature</u>

To develop and integrate each feature into the system. Features are coded, tested, and integrated as complete units. This process is repeated for each feature, resulting in incremental, client-valued software functionality.

<u>Agile Unified Process (AUP)</u>

This an agile adaptation of the traditional Unified Process (UP) that combines structured phases with agile, iterative practices.

It uses a "*serial in the large*" and "*iterative in the small*" approach to develop computer-based systems refers to a hybrid approach in software development where the overall structure of the project is organized in a sequential manner, but within each major phase , work is done iteratively.

SVIT

Serial in the Large

AUP follows the classic UP phases—Inception, Elaboration, Construction, and Transition.

These phases are executed in a linear sequence, providing a high-level, structured view of the entire project. This serial flow helps teams visualize the overall project path, clarifying the steps needed to progress from concept to delivery.

Iterative in the Small

Within each UP phase, the team works iteratively to enhance agility. By repeatedly cycling through smaller activities, the team can adapt to change, deliver useful software increments more quickly, and refine the product based on feedback.

Flexibility with UML Modeling

AUP is rooted in the Unified Process and uses UML for modeling, UML can also be applied with other agile methodologies.This flexibility allows teams to incorporate UML where it adds value, regardless of the specific agile approach they choose.

Agile Modeling (AM)

Agile Modeling (AM) offers an adaptive, practical, and lightweight approach to creating models in software engineering. It aiming to make the modeling process less cumbersome, more accessible, and still highly effective.

Scott Ambler, the creator of AM, describes it as a collection of values, principles, and practices meant to streamline the process of documenting software-based systems. This is achieved by using "just barely good" models rather than aiming for perfection, embracing agility and adaptability as core components.

The main principles that define Agile Modeling include:

Model with a Purpose: Models should have a clear goal, such as communicating information to stakeholders or clarifying an aspect of the software system.

Use Multiple Models: Different aspects of a system may require different models or notations, and only the essential ones should be used. This allows AM to offer insight by presenting the necessary perspectives without overloading the development process.

Travel Light: In Agile Modeling, only models that provide long-term value are kept, while others are discarded as development progresses.

Content Over Representation: AM values the clarity and relevance of the model's content for its intended audience over strict adherence to notation standards.

Know the Models and Tools: Developers should understand the capabilities and limitations of both the models they use and the tools they employ to create them.

Adapt Locally: The modeling approach should be tailored to meet the specific needs of the agile team, promoting flexibility and enhancing productivity.

SVIT

Principles That Guide Process

Regardless of whether a model is linear or iterative, prescriptive or agile, it can be characterized using the generic **process framework** that is applicable for all process models.
The following set of core principles:
Principle 1-Be agile: Whether the process model is prescriptive or agile, the basic tenets of agile development should govern the approach. Every aspect of the work should emphasize economy of action keep technical approach as simple as possible, keep the work products as concise as possible, and make decisions locally whenever possible.
Principle 2-Focus on quality at every step: The exit condition for every process activity, action, and task should focus on the quality of the work product that has been produced.
Principle 3- Be ready to adapt: Process is not a religious experience, and dogma has no place in it. When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.
Principle 4-Build an effective team: Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team that has mutual trust and respect. 98 PART TWO MODELING uote: "In theory there is no difference between theory and practice. But, in practice, there is." Jan van de Snepscheut Every project and every team is unique. That means that you must adapt your process to best fit your needs.
Principle 5. Establish mechanisms for communication and coordination. Projects fail because important information falls into the cracks and/or stakeholders fail to coordinate their efforts to create a successful end product. These are management issues and they must be addressed.
Principle 6. Manage change. The approach may be either formal or informal, but mechanisms must be established to manage the way changes are requested, assessed, approved, and implemented. **Principle 7. Assess risk**. Lots of things can go wrong as software is being developed. It's essential that you establish contingency plans.
Principle 8. Create work products that provide value for others. Create only those work products that provide value for other process activities, actions, or tasks. Every work product that is produced as part of software engineering practice will be passed on to someone else. A list of required functions and features will be passed along to the person (people) who will develop a design; the design will be passed along to those who generate code, and so on. Be sure that the work product imparts the necessary information without ambiguity or omission. Part 4 of this book focuses on project and process management issues and considers various aspects of each of these principles in some detail.

SVIT

Principles That Guide Practice

- **Divide and Conquer**: Emphasizes **Separation of Concerns (SoC)**, where complex problems are broken down into smaller, manageable parts that can be solved and tested independently.

- **Use of Abstraction**: Involves creating simplified representations of complex elements to communicate key ideas without overwhelming details. However, abstractions can sometimes "leak," revealing underlying complexity, so understanding the details behind abstractions remains essential.

- **Strive for Consistency**: Whether in interface design, code structure, or documentation, consistency provides a familiar environment that enhances usability and reduces errors.

- **Focus on Information Transfer**: Effective software development prioritizes error-free information flow across interfaces, as software is fundamentally about exchanging information between components, systems, and users.

- **Effective Modularity**: Building on SoC, this principle emphasizes cohesive modules with **low coupling**. Each module should handle a single, specific function and interact simply with other parts of the system.

- **Look for Patterns**: Leveraging established patterns allows developers to solve recurring problems and build upon collective knowledge, creating a shared language for common challenges and solutions in software development.

- **Multiple Perspectives**: Examining problems and solutions from different viewpoints (data, function, behavior) helps uncover errors, increase understanding, and develop a more comprehensive solution.

- **Maintainability**: Software inevitably needs maintenance; good practices from the outset can facilitate smoother adaptation, enhancement, and defect correction over time.

Communication principles

These communication principles emphasize effective practices for gathering and clarifying requirements from customers and stakeholders, laying a solid foundation for software development. Here's a brief summary of each principle:

1. **Listen**: Focus fully on the speaker's words without immediately forming responses. Ask clarifying questions as needed but avoid interrupting or showing negative reactions.
2. **Prepare Before Communicating**: Understand the problem, research relevant terminology, and prepare an agenda if you're leading the meeting.
3. **Facilitate the Communication**: A facilitator should guide discussions, resolve conflicts, and ensure that communication principles are respected.
4. **Face-to-Face Communication**: This is the most effective form of communication, especially when supplemented with visual aids or reference materials.
5. **Document Notes and Decisions**: Designate someone as a recorder to capture key points and decisions to avoid any loss of information.
6. **Strive for Collaboration**: Encourage teamwork to leverage collective knowledge, build trust, and align team members toward a common goal.
7. **Stay Focused and Modularize Discussions**: Keep discussions organized by sticking to one topic at a time, which the facilitator should ensure.
8. **Use Visuals for Clarity**: When verbal explanations are unclear, create sketches or diagrams to make complex concepts easier to understand.
9. **Know When to Move On**: Don't get stuck on unresolved issues; keep the conversation agile by acknowledging when it's best to revisit certain topics later.
10. **Approach Negotiation Collaboratively**: Aim for a "win-win" outcome in negotiations, balancing compromises with a focus on the shared goal.

These principles help build a constructive communication environment that supports clear understanding, collaborative decision-making, and effective problem-solving within software engineering projects.

SVIT

Planning principles

These planning principles provide a comprehensive approach for guiding software project development, balancing flexibility with structured guidance. Here’s a brief overview of each principle:

1. **Understand the Scope**: Clearly define the project’s objectives and goals to provide direction and clarity.
2. **Involve Stakeholders**: Include stakeholders in planning to set priorities and constraints, ensuring alignment with their needs.
3. **Recognize Iterative Planning**: Accept that planning is not static; plans will need adjustment as new information and feedback becomes available.
4. **Estimate Based on Known Information**: Use the best available information to make realistic estimates of time, effort, and cost, recognizing that estimates may change as more information becomes available.
5. **Consider Risk**: Identify high-impact, high-probability risks and prepare contingency plans to manage potential disruptions.
6. **Be Realistic**: Account for human factors, including work capacity, communication errors, and inevitable changes, as part of the planning process.
7. **Adjust Granularity**: Plan with greater detail for near-term tasks, while keeping a broader view for longer-term activities that may be less certain.
8. **Define Quality Assurance Methods**: Clearly outline how quality will be maintained, such as scheduling reviews, testing, and quality control measures.
9. **Plan for Change**: Establish a process for accommodating changes, including assessing the cost and impact of requested modifications.
10. **Track Progress and Adjust**: Monitor progress frequently, making necessary adjustments to the plan to address slippage or unexpected issues.

Incorporating these principles into project planning promotes proactive management, adaptability, and a structured yet flexible approach to handling changes and unforeseen challenges throughout the software development process.

Modeling principles

These modeling principles are designed to guide software teams in creating effective, efficient models that facilitate software development without becoming an end in themselves. Here's a summary of each principle:

1. **Focus on Building Software, Not Models**: The main goal is delivering functional software quickly. Models should help this process, not slow it down.
2. **Create Only Necessary Models**: Each model requires updates and maintenance. Avoid creating more than necessary to keep the process lean and focused.
3. **Keep Models Simple**: Simple models are easier to understand, integrate, test, and maintain. Complexity in models often leads to complexity in the software, which can reduce overall quality and increase development time.
4. **Build Models for Change**: Anticipate that models will need adjustments as requirements evolve. However, this doesn't mean creating incomplete models; a reasonable level of detail is still essential.
5. **Define the Purpose of Each Model**: Every model should serve a specific purpose. If a model doesn’t add significant value, it’s not worth the time to create it.
6. **Adapt Models to the System**: Modify models to suit the nature of the system being developed. Different types of applications may benefit from specialized modeling techniques.
7. **Prioritize Utility Over Perfection**: Avoid the trap of striving for perfect models. A model that adequately supports the next steps is more valuable than an idealized, complete model.

SVIT

8. **Don't Be Too Rigid with Syntax**: Consistency is helpful, but the ultimate purpose of a model is to communicate effectively. Syntax is secondary to clarity and utility.
9. **Trust Your Instincts**: Experienced engineers often have an intuitive sense when a model is problematic. If something feels off, investigate further or consider alternatives.
10. **Seek Feedback Early**: Regular reviews help identify and correct issues early, ensuring models remain aligned with project goals and requirements.

By following these principles, software teams can create models that are both useful and efficient, supporting the agile and iterative nature of modern software development.

Construction principles

Coding Principles

Coding is more than just writing lines of code—it's about creating software that is maintainable, reliable, and functional. Here are some coding principles organized into three categories:

1. **Preparation Principles**: Before starting to code, ensure you:
   - Fully understand the problem and design principles.
   - Choose an appropriate programming language and environment.
   - Prepare unit tests to validate the code once it's completed.
2. **Programming Principles**: While coding, you should:
   - Use structured programming practices.
   - Consider pair programming for collaborative benefits.
   - Choose suitable data structures and align interfaces with the architecture.
   - Keep logic straightforward, name variables meaningfully, and follow coding standards.
   - Write self-documenting code and maintain a clear, visually accessible layout.
3. **Validation Principles**: After coding, make sure to:
   - Conduct code walkthroughs.
   - Perform unit testing and resolve identified errors.
   - Refactor the code for improvements.

Testing Principles

Effective testing is vital for ensuring the software works as expected. Here are foundational principles for testing:

1. **Error-Finding Goal**: The aim of testing is to uncover errors, not to show the software works without flaws. A successful test finds undiscovered errors.
2. **Traceability to Requirements**: Every test should be tied back to customer requirements to ensure the software meets its objectives.
3. **Early Test Planning**: Begin planning tests once the requirements model is complete. This approach allows for thorough testing design as the system is developed.
4. **Apply the Pareto Principle**: Focus testing efforts on the areas that are most error-prone, as a small portion of components often contains the majority of errors.
5. **Testing Progression**: Start testing with individual components (unit testing), and gradually move toward larger, integrated sections of the system.
6. **Avoid Exhaustive Testing**: It's impractical to test all possible paths in most programs. Instead, focus on adequately covering program logic and identifying critical test cases.

By following these coding and testing principles, developers can create more reliable, maintainable, and user-aligned software.

SVIT

Deployment Principles

Deployment is a recurring activity that includes delivery, support, and feedback. Each deployment cycle provides an opportunity for the software team to gather valuable feedback and make necessary adjustments for future increments. Here are the core principles for a successful deployment:

1. **Manage Customer Expectations**:
   - Ensure that customer expectations are realistic and aligned with what has been promised.
   - Communicate clearly to avoid misunderstandings about deliverables and timelines, as overpromising or delivering inconsistent increments can lead to disappointment and negatively impact team morale.
2. **Assemble and Test a Complete Delivery Package**:
   - Create a comprehensive delivery package that includes all software executables, data files, documentation, and relevant support files.
   - Beta test the package with actual users across a range of environments and configurations (e.g., operating systems, hardware) to confirm that installation and functionality work seamlessly.
3. **Establish a Support System**:
   - Prepare a well-defined support structure with materials and resources to respond effectively to user questions and issues.
   - Implement record-keeping to track support requests, which can help in assessing common user issues and improve future software increments.
4. **Provide Instructional Materials**:
   - Supply users with clear instructional aids, such as user guides, troubleshooting resources, and updates on new or modified features in each increment.
   - Plan training materials or workshops if necessary, ensuring users are well-equipped to use the software effectively.
5. **Fix Bugs Before Delivery**:
   - Avoid releasing buggy software with a promise to fix it in the next release. Delivering a high-quality, reliable product builds trust and satisfaction, even if it means taking a bit more time.
   - Bug-free (or minimal issues) releases reduce user frustration and prevent long-term dissatisfaction.

By adhering to these deployment principles, software teams can create a smoother, more reliable experience for end users and foster constructive feedback, which in turn supports continuous improvement in future development cycles.

SVIT