AGILE DEVELOPMENT
Module 3- Chapter 1

Dr. Vinod Kumar P
Associate Professor
Dept. of CSE-Data Science
ATMECE

Agile Software Development

- Program specification, design and implementation are inter-leaved
- The system is developed as a series of versions or increments with stakeholders involved in version specification and evaluation
- Frequent delivery of new versions for evaluation
- Extensive tool support (e.g. automated testing tools) used to support development.
- Minimal documentation – focus on working code

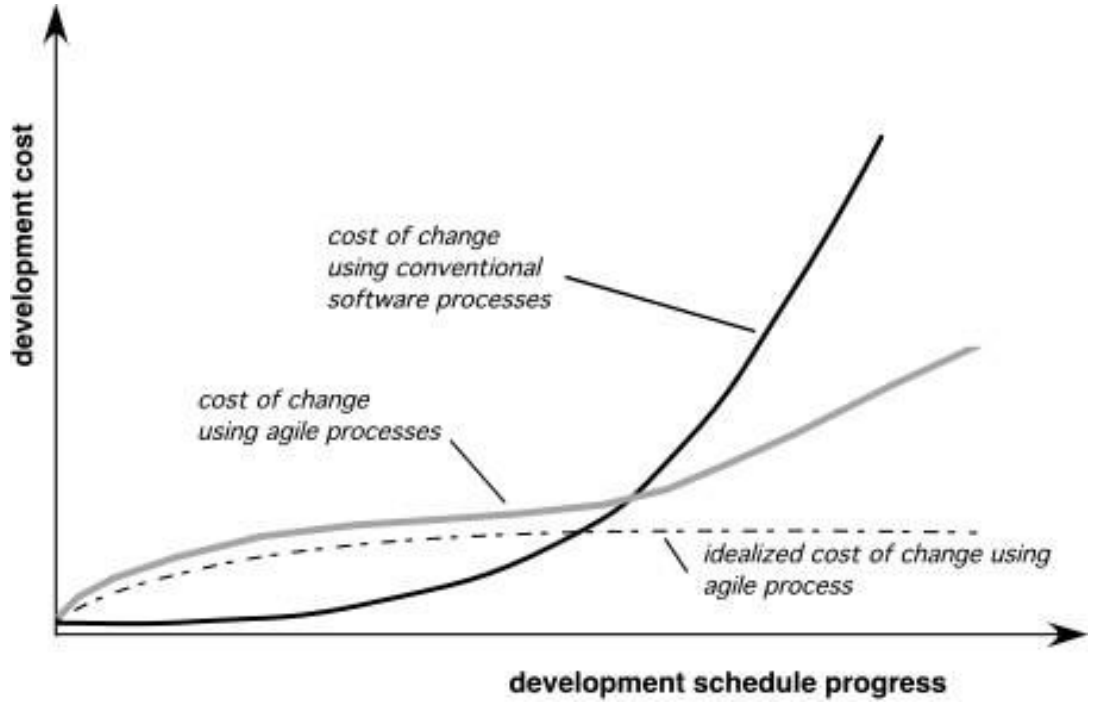
What is “Agility”?

Agile is a software development methodology to build software incrementally using short iterations of 1 to 4 weeks so that the development process is aligned with the changing business needs.

- Effective (rapid and adaptive) response to change
- Effective communication among all stakeholders
- Drawing the customer onto the team
- Organizing a team so that it is in control of the work performed
- Rapid, incremental delivery of software

Agility and the Cost of Change

- Conventional Process,the cost of change increases nonlinearly as a project progresse.It is relatively easy to accommodate a change when a team is gathering requirements early in a project.
- If there are any changes, the costs of doing this work are minimal.
- if in the middle of validation testing, a stakeholder is requesting a major functional change. Then the change requires a modification to the architectural design, construction of new components, changes to other existing components, new testing and so on. Costs escalate quickly.
- Agile process may **"flatten"** the cost of change curve by coupling incremental delivery with agile practices such as continuous unit testing and pair programming.


development cost
cost of change
using conventional
software processes
cost of change
using agile processes
idealized cost of change using
agile process
development schedule progress

What is An Agile Process?

Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects:An Agile software process is designed to handle the unpredictability inherent in most software projects.

- 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.
- It is difficult to predict how much design is necessary before construction is used to prove the design.
- Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Given these three assumptions, an important question arises:

How do we create a process that can manage unpredictability:

- It lies in process **adaptability**. An agile process, therefore, must be adaptable. But 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.

Extreme Programming (XP)

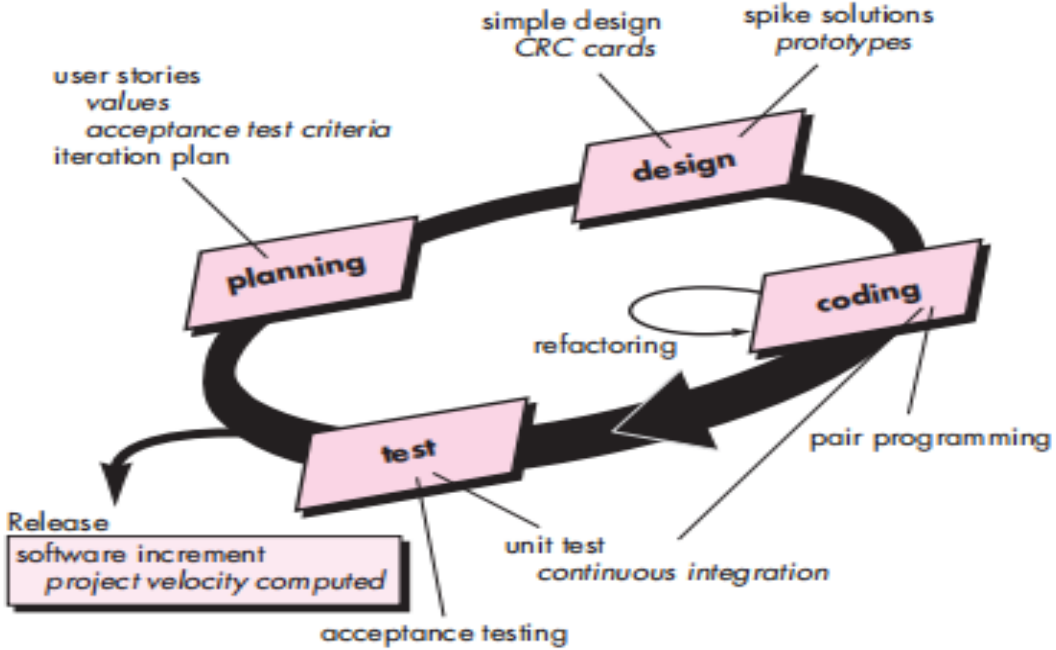
- The most widely used agile process, originally proposed by Kent Beck
- Extreme Programming (XP) takes an 'extreme' approach to iterative development.
  - New versions may be built several times per day;
  - Increments are delivered to customers every 2 weeks;
  - All tests must be run for every build and the build is only accepted if tests run successfully.

XP Planning

- Begins with the creation of "user stories"
- Agile team assesses each story and assigns a cost
- Stories are grouped to for a deliverable increment
- A commitment is made on delivery date
- After the first increment "project velocity" is used to help define subsequent delivery dates for other increments

- XP Design
  - Follows the Keep It Simple principle
  - Encourage the use of CRC cards
  - For difficult design problems, suggests the creation of "spike solutions"—a design prototype.
  - XP encourages refactoring—a construction technique that is also a method for design
  - optimization
  - Refactoring— means, it is the process of changing a software system in a way that it does not change the external behavior of the code and improves the internal structure.

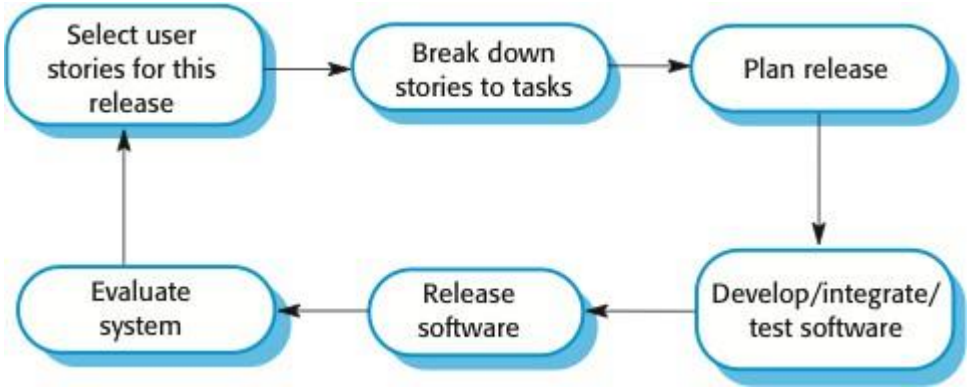
- XP Coding
  - Recommends the construction of a unit test for a store *before* coding commences
  - Encourages "pair programming"
- XP Testing
  - All unit tests are executed daily
  - "Acceptance tests" are defined by the customer and excuted to assess customer visible functionality


simple design
CRC cards
spike solutions
prototypes
user stories
values
acceptance test criteria
iteration plan
design
planning
coding
refactoring
test
pair programming
Release
software increment
project velocity computed
unit test
continuous integration
acceptance testing

XP and agile principles

- Incremental development is supported through small, frequent system releases.
- Customer involvement means full-time customer engagement with the team.
- People not process through pair programming, collective ownership and a process that avoids long working hours.
- Change supported through regular system releases.
- Maintaining simplicity through constant refactoring of code.

The extreme programming release cycle



Other Agile Models

- The most widely used of all agile process models is Extreme Programming (XP).
- But many other agile process models have been proposed and are in use across the industry.
- Among the most common are:
  - Adaptive Software Development (ASD)
  - Scrum
  - Dynamic Systems Development Method (DSDM)
  - Crystal
  - Feature Drive Development (FDD)
  - Lean Software Development (LSD)
  - Agile Modeling (AM)

Adaptive Software Development

- It is a method to build complex software and system. ASD focuses on human collaboration and self-organization.
- Originally proposed by Jim Highsmith. ” He defines an ASD “life cycle” that incorporates three phases, **speculation, collaboration, and learning**
- ASD — distinguishing features
  - Mission-driven planning
  - Component-based focus
  - Uses “time-boxing” (See Chapter 24)
  - Explicit consideration of risks
  - Emphasizes collaboration for requirements gathering

1. Speculation:

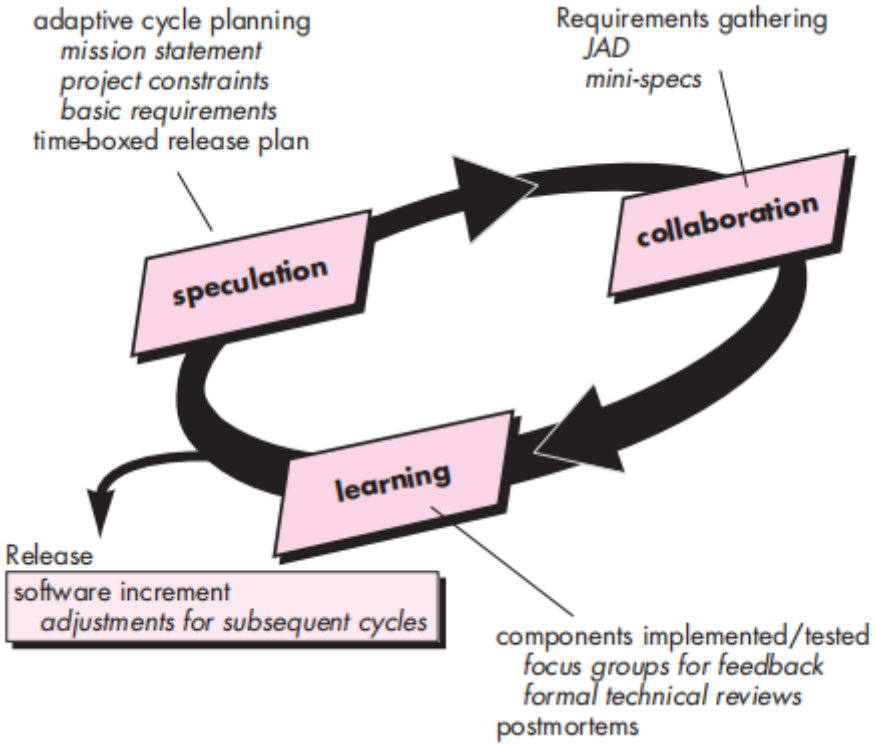
- During this phase project is initiated and planning is conducted. The project plan uses project initiation information like project requirements, user needs, customer mission statement, etc, to define set of release cycles that the project wants.

2. Collaboration:

- It is the difficult part of ASD as it needs the workers to be motivated. It collaborates communication and teamwork but emphasizes individualism as individual creativity plays a major role in creative thinking. People working together must trust each other to
- (1) criticize without animosity, (2) assist without resentment, (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action

3. Learning:

- Learning helps the workers to increase their level of understanding over the project.
- Learning process is of 3 ways:
  - Focus groups
  - Technical reviews
  - Project postmortem
- overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning.


adaptive cycle planning
mission statement
project constraints
basic requirements
time-boxed release plan
Requirements gathering
JAD
mini-specs
speculation
collaboration
learning
Release
software increment
adjustments for subsequent cycles
components implemented/tested
focus groups for feedback
formal technical reviews
postmortems

Dynamic Systems Development Method

- The Dynamic Systems Development Method (DSDM) is an agile software development approach that "provides a framework for building and maintaining systems which meet the tight time constraints through the use of incremental prototyping in a controlled project environment
- Similar in most respects to XP and/or ASD
- The DSDM life cycle that defines three different iterative cycles, preceded by two additional life cycle activities:
  - Feasibility study—establishes the basic business requirements and constraints associated with the application.
  - Business study—establishes the functional and information requirements that will allow the application to provide business value.
  - Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer.

- Design and build iteration—revisits prototypes built during functional model iteration to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users.
- Implementation—places the latest software increment into the operational environment. Itshould be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place.

Scrum

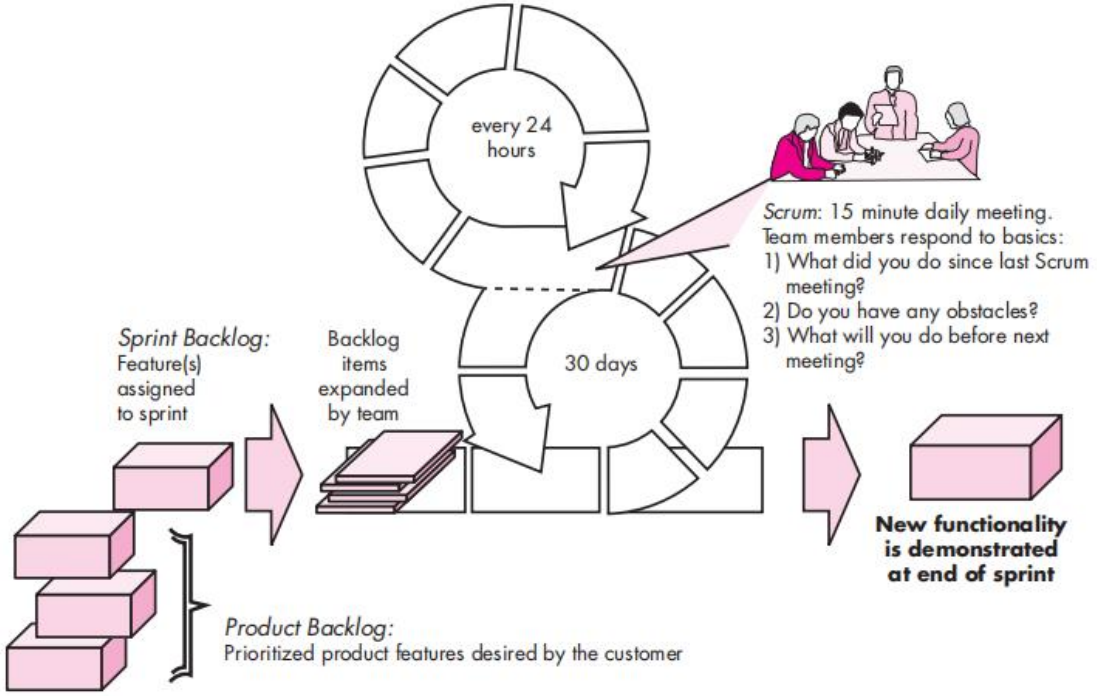
- Originally proposed by Schwaber and Beedle
- SCRUM is an agile development process focused primarily on ways to manage tasks in team-based development conditions.
- Scrum—distinguishing features
  - Development work is partitioned into "packets"
  - Testing and documentation are on-going as the product is constructed
  - Work occurs in "sprints" and is derived from a "backlog" of existing requirements
  - Meetings are very short and sometimes conducted without chairs
  - "demos" are delivered to the customer with the time- box allocated

Scrum

- There are three roles in it, and their responsibilities are:

- **Scrum Master**: The scrum can set up the master team, arrange the meeting and remove obstacles for the process
- **Product owner**: The product owner makes the product backlog, prioritizes the delay and is responsible for the distribution of functionality on each repetition.
- **Scrum Team**: The team manages its work and organizes the work to complete the sprint or cycle.

Scrum


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

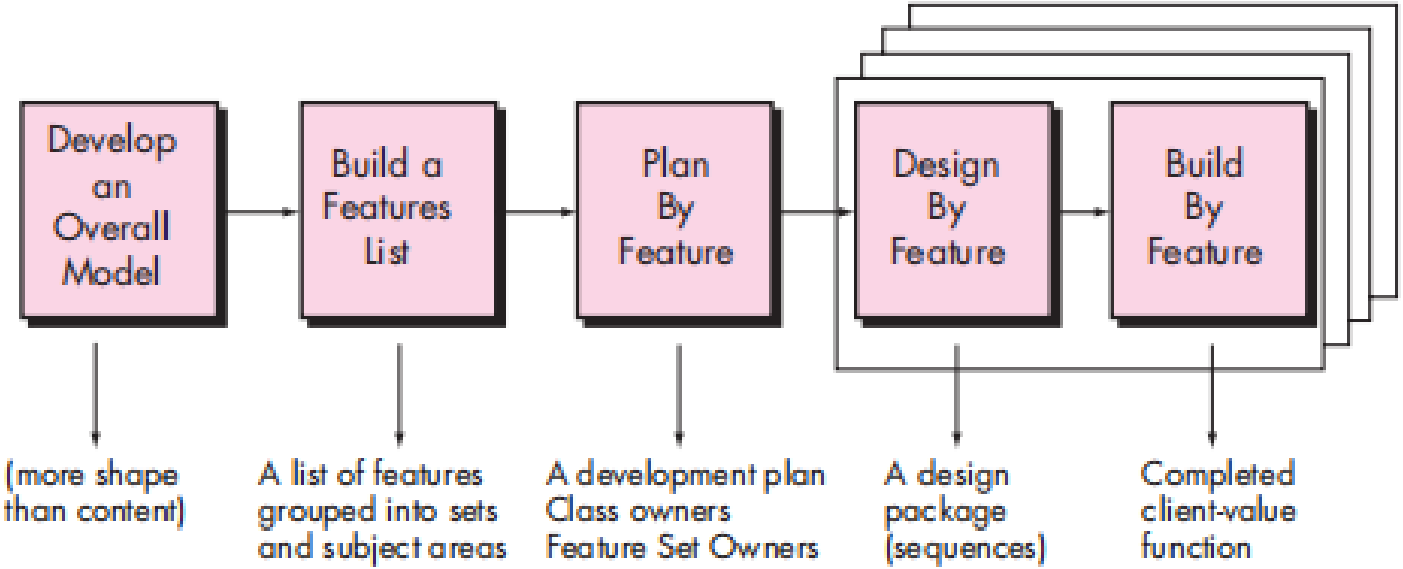
Crystal

- Proposed by Cockburn and Highsmith
- Crystal methods are flexible approaches used in Agile software development to manage projects effectively.
- They adapt to the needs of the team and the project, promoting collaboration, communication, and adaptability for successful outcomes.
- Crystal—distinguishing features
  - Frequent Delivery
  - Reflective Improvement
  - Personal Safety
  - Focus
  - Teamwork
  - Continuous learning
  - Customer involvement
  - Timeboxing

Feature Driven Development

- Originally proposed by Peter Coad et al
- This method focuses on "Designing and Building" features. In contrast to other smart methods, FDD describes the small steps of the work that should be obtained separately per function.
- FDD—distinguishing features
  - Emphasis is on defining "features"
    - a *feature* "is a client-valued function that can be implemented in two weeks or less."
  - Uses a feature template
    - <action> the <result> <by | for | of | to> a(n) <object>
  - A features list is created and "plan by feature" is conducted
  - Design and construction merge in FDD

Feature Driven Development



Lean Software Development

- Lean software development methodology follows the principle "just in time production."
- The lean method indicates the increasing speed of software development and reducing costs.
- Lean development can be summarized in seven phases.
  - Eliminating Waste
  - Amplifying learning
  - Defer commitment (deciding as late as possible)
  - Empowering the team
  - Building Integrity
  - Optimize the whole

LSD PRINCIPLE

Agile Modeling

- It is a methodology for modeling and documenting software systems based on best practices. It is a collection of values and principles that can be applied on an (agile) software development project.
- Suggests a set of agile modeling principles.
  - Model with a purpose
  - Adopt Simplicity.
  - Embrace Change
  - Incremental Change
  - Maximize Stakeholder Investment.
  - Know the models and the tools you use to create them
  - Remember the Existence of Multiple Models.

PRINCIPLES THAT GUIDE PRACTICE
Module 3- Chapter 2

Software Engineering Knowledge

- *You often hear people say that software development knowledge has a 3-year half-life: half of what you need to know today will be obsolete within 3 years.*
- *In the domain of technology-related knowledge, that's probably about right. But there is another kind of software development knowledge —a kind that is of as "software engineering principles"— that does not have a three-year half-life. These software engineering principles are likely to serve a professional programmer throughout his or her career.*

Steve McConnell

Core Principles

- Software engineering is guided by a **collection of core principles** that help in the application of a meaningful software process and the execution of effective software engineering methods.
- At the **process level,** core principles establish a philosophical foundation that guides a software team as it performs framework and umbrella activities, navigates the process flow, and produces a set of software engineering work products.
- At the **level of practice**, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

Principles that Guide Process - I

- **Principle #1.** ***Be agile.*** Whether the process model you choose is prescriptive or agile, the basic tenets of agile development should govern your approach.
- **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.

Principles that Guide Process - II

- **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.
- **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.

Principles that Guide Practice

- **Principle #1.** ***Divide and conquer.*** Stated in a more technical manner, analysis and design should always emphasize *separation of concerns* (SoC).
- **Principle #2.** ***Understand the use of abstraction.*** At it core, an abstraction is a simplification of some complex element of a system used to communication meaning in a single phrase.
- **Principle #3. Strive for consistency.** A familiar context makes software easier to use.
- **Principle #4.** ***Focus on the transfer of information.*** Pay special attention to the analysis, design, construction, and testing of interfaces.

Principles that Guide Practice

- **Principle #5. Build software that exhibits effective modularity.** Separation of concerns (Principle #1) establishes a philosophy for software. Modularity provides a mechanism for realizing the philosophy.

- **Principle #6.** ***Look for patterns.*** Brad Appleton [App00] suggests that: "The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development.

- **Principle #7.** When possible, represent the problem and its solution from a number of different perspectives.
- **Principle #8.** Remember that someone will maintain the software.

Principles that Guide each Framework Activities

- **Principle #5. Build software that exhibits effective modularity.** Separation of concerns (Principle #1) establishes a philosophy for software. Modularity provides a mechanism for realizing the philosophy.

- **Principle #6.** ***Look for patterns.*** Brad Appleton [App00] suggests that: "The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development.

- **Principle #7.** When possible, represent the problem and its solution from a number of different perspectives.
- **Principle #8.** Remember that someone will maintain the software.

Principles that Guide each Framework Activities

Communication Principles

- **Principle #1.** ***Listen.*** Try to focus on the speaker's words, rather than formulating your response to those words.
- **Principle # 2.** ***Prepare before you communicate.*** Spend the time to understand the problem before you meet with others.
- **Principle # 3.** ***Someone should facilitate the activity.*** Every communication meeting should have a leader (a facilitator) to keep the conversation moving in a productive direction; (2) to mediate any conflict that does occur, and (3) to ensure than other principles are followed.
- **Principle #4.** ***Face-to-face communication is best.*** But it usually works better when some other representation of the relevant information is present.

Communication Principles

- **Principle # 5. Take notes and document decisions**. Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.
- **Principle # 6. Strive for collaboration.** Collaboration and consensus occur when the collective knowledge of members of the team is combined …
- **Principle # 7. Stay focused, modularize your discussion.** The more people involved in any communication, the more likely that discussion will bounce from one topic to the next.
- **Principle # 8.** If something is unclear, draw a picture.
- **Principle # 9. (a) Once you agree to something, move on;** (b) If you can't agree to something, move on; (c) If a feature or function is unclear and cannot be clarified at the moment, move on.
- **Principle # 10. Negotiation is not a contest or a game**. It works best when both parties win.

Planning Principles

- **Principle #1.** ***Understand the scope of the project.*** It's impossible to use a roadmap if you don't know where you're going. Scope provides the software team with a destination.
- **Principle #2.** ***Involve the customer in the planning activity.*** The customer defines priorities and establishes project constraints.
- **Principle #3.** ***Recognize that planning is iterative.*** A project plan is never engraved in stone. As work begins, it very likely that things will change.
- **Principle #4.** ***Estimate based on what you know.*** The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.

Planning Principles

- **Principle #5.** ***Consider risk as you define the plan.*** If you have identified risks that have high impact and high probability, contingency planning is necessary.
- **Principle #6.** ***Be realistic.*** People don't work 100 percent of every day.
- **Principle #7.** ***Adjust granularity as you define the plan.*** *Granularity* refers to the level of detail that is introduced as a project plan is developed.
- **Principle #8.** ***Define how you intend to ensure quality.*** The plan should identify how the software team intends to ensure quality.
- **Principle #9.** ***Describe how you intend to accommodate change.*** Even the best planning can be obviated by uncontrolled change.
- **Principle #10.** ***Track the plan frequently and make adjustments as required.*** Software projects fall behind schedule one day at a time.

Modeling Principles

- In software engineering work, two classes of models can be created:
  - ***Requirements models*** **(also called** ***analysis models*****)** represent the customer requirements by depicting the software in three different domains: the **information** domain, the **functional** domain, and the **behavioral** domain.
  - ***Design models*** represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

Requirements Modeling Principles

- **Principle #1.** ***The information domain of a problem must be represented and understood.***
- **Principle #2.** ***The functions that the software performs must be defined.***
- **Principle #3.** **The behavior of the software (as a consequence of external events) must be represented.**
- **Principle #4.** ***The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.***
- **Principle #5.** ***The analysis task should move from essential information toward implementation detail.***

Design Modeling Principles

- **Principle #1.** Design should be traceable to the requirements model.
- **Principle #2.** Always consider the architecture of the system to be built.
- **Principle #3.** Design of data is as important as design of processing functions.
- **Principle #4.** User interface design should be tuned to the needs of the end-user. However, in every case, it should stress ease of use.
- **Principle #5.** Component-level design should be functionally independent.
- **Principle #6.** Components should be loosely coupled to one another and to the external environment.
- **Principle #7.** Design representations (models) should be easily understandable.
- **Principle #8.** The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity.

Agile Modeling Principles

- **Principle #1.** The primary goal of the software team is to build software, not create models.
- **Principle #2.** Travel light—don't create more models than you need.
- **Principle #3.** Strive to produce the simplest model that will describe the problem or the software.
- **Principle #4.** Build models in a way that makes them amenable to change.
- **Principle #5.** Be able to state an explicit purpose for each model that is created.
- **Principle #6.** Adapt the models you develop to the system at hand.
- **Principle #7.** Try to build useful models, but forget about building perfect models.

Agile Modeling Principles

- **Principle #8**. Don't become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary.
- **Principle #9**. If your instincts tell you a model isn't right even though it seems okay on paper, you probably have reason to be concerned.
- **Principle #10**. Get feedback as soon as you can.

Construction Principles

- The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end-user.
- **Coding principles and concepts** are closely aligned programming style, programming languages, and programming methods.
- **Testing principles and concepts** lead to the design of tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

Preparation Principles

- ***Before you write one line of code, be sure you:***
  - Understand of the problem you're trying to solve.
  - Understand basic design principles and concepts.
  - Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
  - Select a programming environment that provides tools that will make your work easier.
  - Create a set of unit tests that will be applied once the component you code is completed.

Coding Principles

- ***As you begin writing code, be sure you:***
  - Constrain your algorithms by following structured programming [Boh00] practice.
  - Consider the use of pair programming
  - Select data structures that will meet the needs of the design.
  - Understand the software architecture and create interfaces that are consistent with it.
  - Keep conditional logic as simple as possible.
  - Create nested loops in a way that makes them easily testable.
  - Select meaningful variable names and follow other local coding standards.
  - Write code that is self-documenting.
  - Create a visual layout (e.g., indentation and blank lines) that aids understanding

Validation Principles

- After you've completed your first coding pass, be sure you:
  - Conduct a code walkthrough when appropriate.
  - Perform unit tests and correct errors you've uncovered.
  - Refactor the code.

Testing Principles

- Al Davis [Dav95] suggests the following:
  - **Principle #1.** All tests should be traceable to customer requirements.
  - **Principle #2.** Tests should be planned long before testing begins.
  - **Principle #3.** *The Pareto principle applies to software testing.*
  - **Principle #4.** Testing should begin "in the small" and progress toward testing "in the large."
  - **Principle #5.** Exhaustive testing is not possible.

Deployment Principles

- **Principle #1.** ***Customer expectations for the software must be managed.*** Too often, the customer expects more than the team has promised to deliver, and disappointment occurs immediately.
- **Principle #2.** ***A complete delivery package should be assembled and tested.***
- **Principle #3.** A support regime must be established before the software is delivered. An end-user expects responsiveness and accurate information when a question or problem arises.
- **Principle #4.** *Appropriate instructional materials must be provided to end-users.*
- **Principle #5.** Buggy software should be fixed first, delivered later.