Midterm Exam Study Guide

This study guide **does not imply** that any topics/slides that are not mentioned are excluded. The objective is to guide you while studying by focusing on certain topics that might have a high weight with regard to the midterm exam.

Please note:

MCQ & T/F will come from all chapters
Short Answer Q. will be from Chapters **1, 2, 6, 7, 8, 10 and 11**
Long Answer Q. will be from Chapter **4&5**

Chapter 1: **The Nature of Software**

⦿ Legacy Software
- ✧ software must be adapted to meet the needs of new computing environments or technology.
- ✧ software must be enhanced to implement new business requirements.
- ✧ software must be extended to make it interoperable with other more modern systems or databases.
- ✧ software must be re-architected to make it viable within a network environment**.**

⦿ Cloud Computing
- ✧ *Cloud computing* provides distributed data storage and processing resources to networked computing devices
- ✧ Computing resources reside outside the cloud and have access to a variety of resources inside the cloud
- ✧ Cloud computing requires developing an architecture containing both frontend and backend services
- ✧ Frontend services include the client devices and application software to allow access
- ✧ Backend services include servers, data storage, and server-resident applications
- ✧ Cloud architectures can be segmented to restrict access to private data

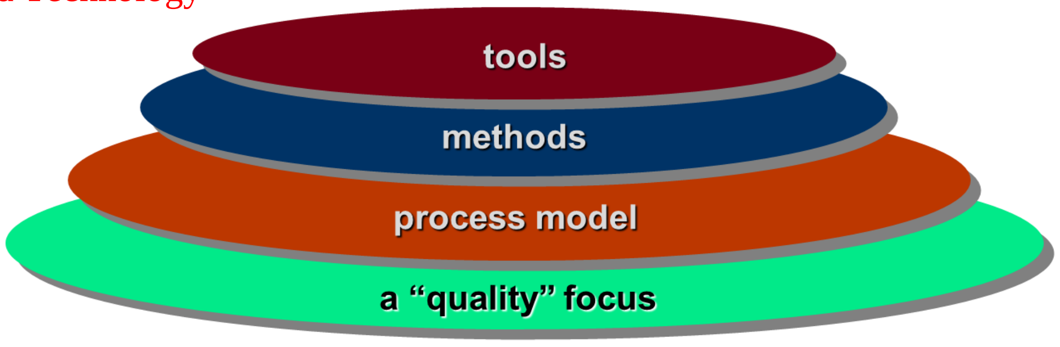
⦿ Product Line Software
- ✧ *Product line software* is a set of software-intensive systems that share a common set of features and satisfy the needs of a particular market
- ✧ These software products are developed using the same application and data architectures using a common core of reusable software components
- ✧ A software product line shares a set of assets that include *requirements, architecture, design patterns, reusable components, test cases,* and other work products
- ✧ A software product line allow in the development of many products that are engineered by capitalizing on the commonality among all products with in the product line

⦿ Characteristics of WebApps - II
- ✧ **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end-user.
- ✧ **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApps.
- ✧ **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically-spaced releases, Web applications evolve continuously.
- ✧ **Immediacy.** Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time to market that can be a matter of a few days or weeks.

- ✧ **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end-users who may access the application.
- ✧ **Aesthetics.** An undeniable part of the appeal of a WebApp is its look and feel.

Chapter 2: **Software Engineering**

⦿ A Layered Technology



Software Engineering

⦿ Umbrella Activities

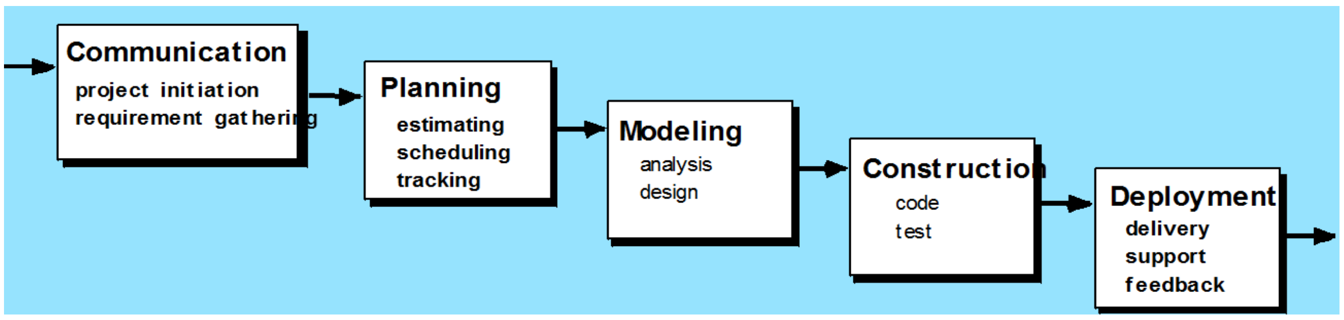
- ✧ Software project tracking and control
- ✧ Risk management
- ✧ Software quality assurance
- ✧ Technical reviews
- ✧ Measurement
- ✧ Software configuration management
- ✧ Reusability management
- ✧ Work product preparation and production

⦿ Hooker's General Principles

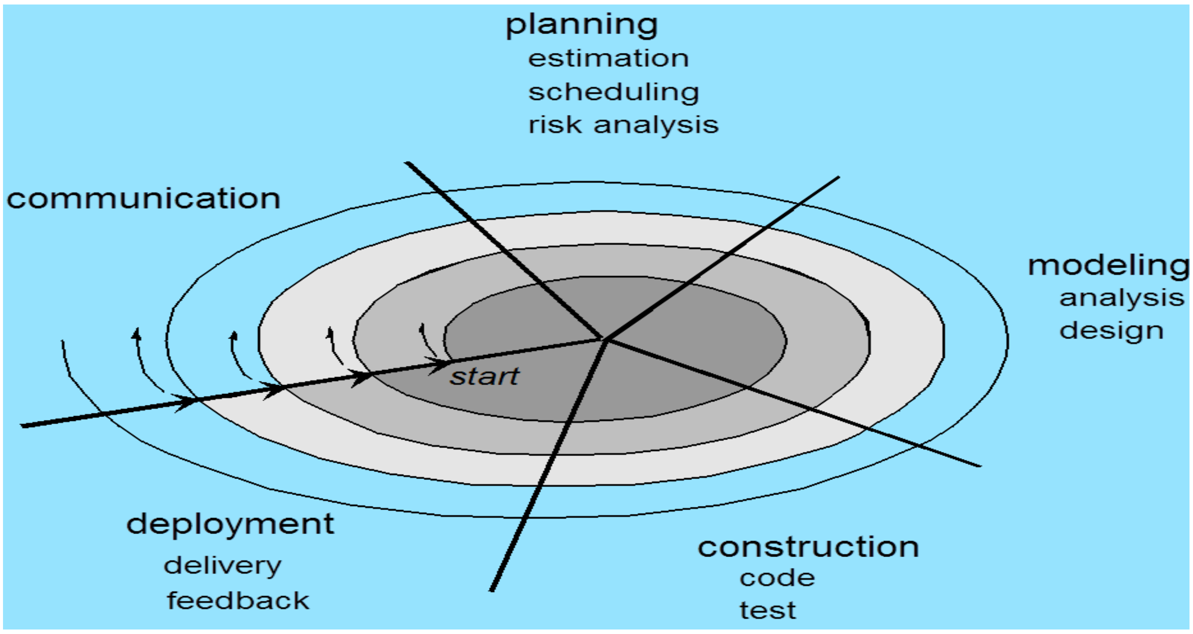
1) The Reason It All Exists
2) KISS (Keep It Simple, Stupid!)
3) Maintain the Vision
4) What You Produce, Others Will Consume
5) Be Open to the Future
6) Plan Ahead for Reuse
7) Think!

Chapter 4 & Chapter 5 Important Concepts : **Process Models**

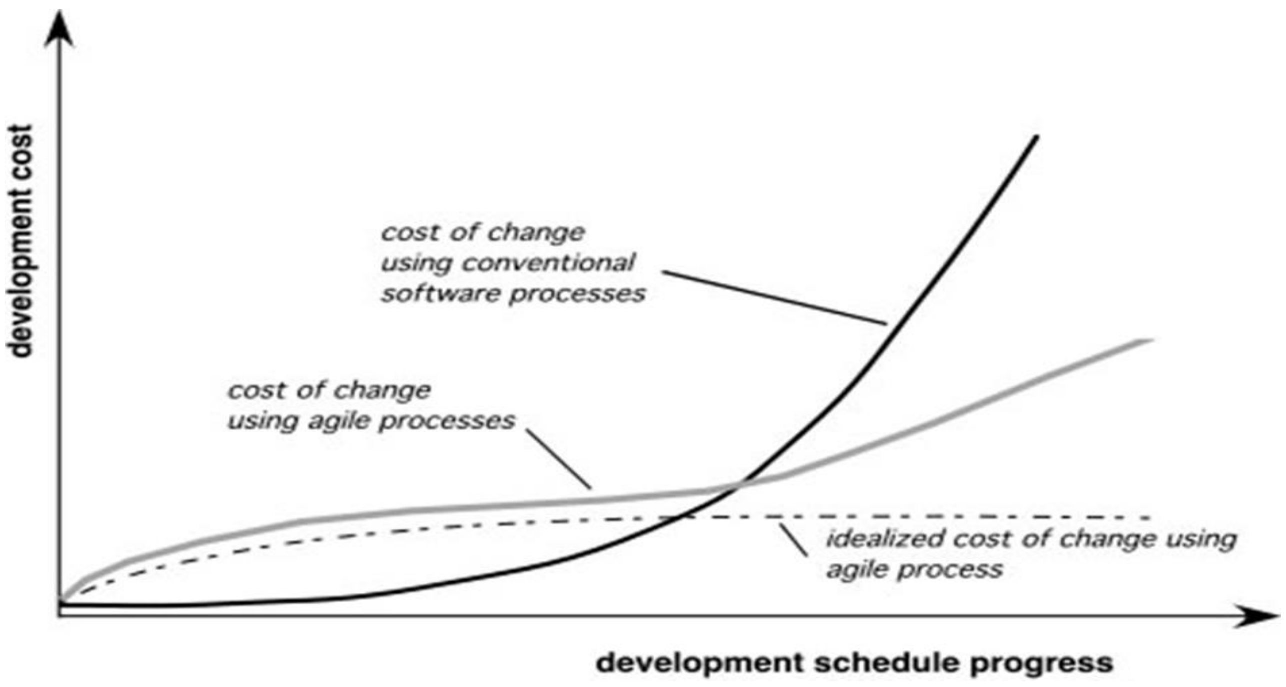
⦿ The Waterfall Model



⊙ Evolutionary Models: The Spiral



⊙ Agility and the Cost of Change



⊙ An Agile Process

✧ Is driven by customer descriptions of what is required (scenarios)

✧ Recognizes that plans are short-lived

✧ Develops software iteratively with a heavy emphasis on construction activities

✧ Delivers multiple 'software increments'

✧ Adapts as changes occur

⊙ Extreme Programming (XP)

✧ The most widely used agile process, originally proposed by Kent Beck

✧ 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

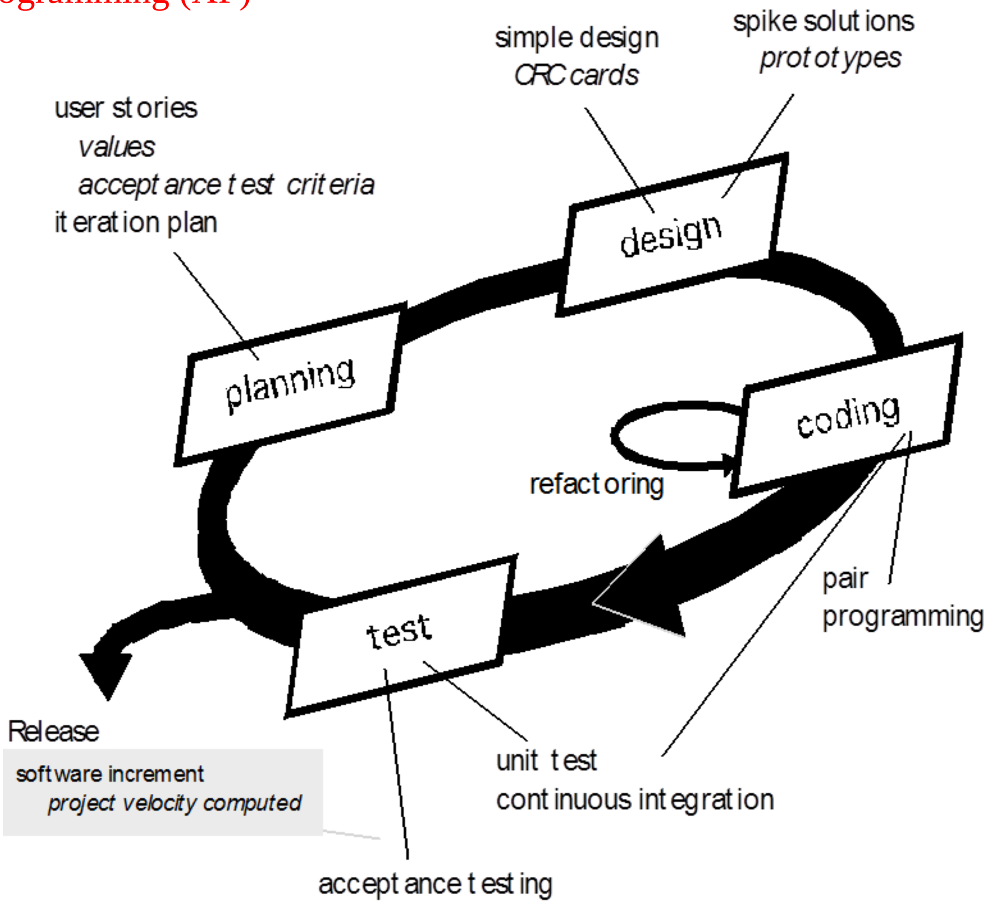
✧ A commitment is made on delivery date

- After the first increment "project velocity" is used to help define subsequent delivery dates for other increments

- Extreme Programming (XP)

- XP Design
  - Follows the KIS principle
  - Encourage the use of CRC cards (see Chapter 8)
  - For difficult design problems, suggests the creation of "spike solutions"—a design prototype
  - Encourages "refactoring"—an iterative refinement of the internal program design
- 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

- Extreme Programming (XP)



Chapter 6 : **Human Aspects of Software Engineering**

- Boundary Spanning : Team Roles
  - Ambassador – represents team to outside constituencies
  - Scout – crosses team boundaries to collect information
  - Guard – protects access to team work products
  - Sentry – controls information sent by stakeholders
  - Coordinator – communicates across the team and organization

- Avoid Team “Toxicity”
  - A frenzied work atmosphere in which team members waste energy and lose focus on the objectives of the work to be performed.
  - High frustration caused by personal, business, or technological factors that cause friction among team members.
  - “Fragmented or poorly coordinated procedures” or a poorly defined or improperly chosen process model that becomes a roadblock to accomplishment.
  - Unclear definition of roles resulting in a lack of accountability and resultant finger-pointing.
  - “Continuous and repeated exposure to failure” that leads to a loss of confidence and a lowering of morale.
- Organizational Paradigms
  - closed paradigm—structures a team along a traditional hierarchy of authority
  - random paradigm—structures a team loosely and depends on individual initiative of the team members
  - open paradigm—attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm
  - synchronous paradigm—relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves
- Team Decisions Making Complications
  - Problem complexity
  - Uncertainty and risk associated with the decision
  - Work associated with decision has unintended effect on another project object (law of unintended consequences)
  - Different views of the problem lead to different conclusions about the way forward
  - Global software teams face additional challenges associated with collaboration, coordination, and coordination difficulties

Chapter 7 : **Principles that Guide Practice**

- 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.
- 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.

⦿ 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 makies them amenable to change.
- ◊ Principle #5. Be able to state an explicit purpose for each model that is created.

⦿ 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.

⦿ 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.

⦿ Deployment Principles

- ◊ Principle #1. Customer expectations for the software must be managed.
- ◊ Principle #2. A complete delivery package should be assembled and tested.
- ◊ Principle #3. A support regime must be established before the software is delivered.
- ◊ Principle #4. Appropriate instructional materials must be provided to end-users.
- ◊ Principle #5. Buggy software should be fixed first, delivered later.

Chapter 8 : **Understanding Requirements**

⦿ Requirements Engineering-I

- ♣ Inception—ask a set of questions that establish …
    - o basic understanding of the problem
    - o the people who want a solution
    - o the nature of the solution that is desired, and
    - o the effectiveness of preliminary communication and collaboration between the customer and the developer
- ♣ Elicitation—elicit requirements from all stakeholders

- Elaboration—create an analysis model that identifies data, function and behavioral requirements
- Negotiation—agree on a deliverable system that is realistic for developers and customers

Requirements Engineering-II

- Specification—can be any one (or more) of the following:
  - A written document
  - A set of models
  - A formal mathematical
  - A collection of user scenarios (use-cases)
  - A prototype
- Validation—a review mechanism that looks for
  - errors in content or interpretation
  - areas where clarification may be required
  - missing information
  - inconsistencies (a major problem when large products or systems are engineered)
  - conflicting or unrealistic (unachievable) requirements.
- Requirements management

Eliciting Requirements

- meetings are conducted and attended by both software engineers and customers
- rules for preparation and participation are established
- an agenda is suggested
- a "facilitator" (can be a customer, a developer, or an outsider) controls the meeting
- a "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used
- the goal is
  - to identify the problem
  - propose elements of the solution
  - negotiate different approaches, and
  - specify a preliminary set of solution requirements

Negotiating Requirements

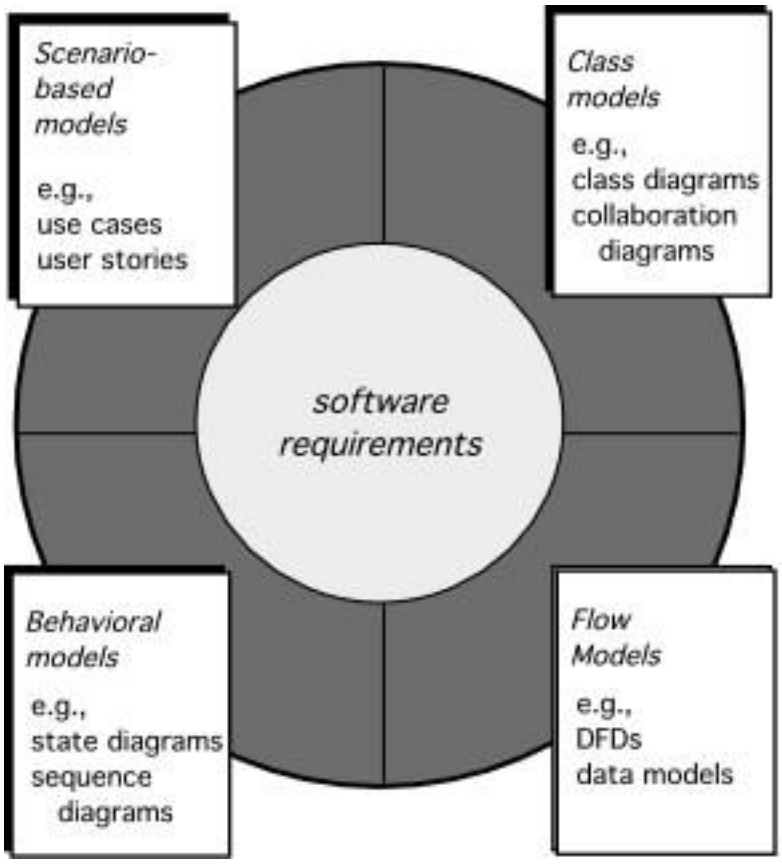
- Identify the key stakeholders
  - These are the people who will be involved in the negotiation
- Determine each of the stakeholders "win conditions"
  - Win conditions are not always obvious
- Negotiate
  - Work toward a set of requirements that lead to "win-win"

Requirements Monitoring

- Especially needs in incremental development
  - ◊ Distributed debugging – uncovers errors and determines their cause.
  - ◊ Run-time verification – determines whether software matches its specification.
  - ◊ Run-time validation – assesses whether evolving software meets user goals.
  - ◊ Business activity monitoring – evaluates whether a system satisfies business goals.
  - ◊ Evolution and codesign – provides information to stakeholders as the system evolves.

Chapter 9 : **Requirements Modeling: Scenario-Based Methods**

- Requirements Analysis
  - Requirements analysis
    - specifies software's operational characteristics
    - indicates software's interface with other system elements
    - establishes constraints that software must meet
  - Requirements analysis allows the software engineer (called an *analyst* or *modeler* in this role) to:
    - elaborate on basic requirements established during earlier requirement engineering tasks
    - build models that depict user scenarios, functional activities, problem classes and their relationships, system and class behavior, and the flow of data as it is transformed.
- Elements of Requirements Analysis


Scenario-based models
e.g., use cases user stories
Class models
e.g., class diagrams collaboration diagrams
software requirements
Behavioral models
e.g., state diagrams sequence diagrams
Flow Models
e.g., DFDs data models

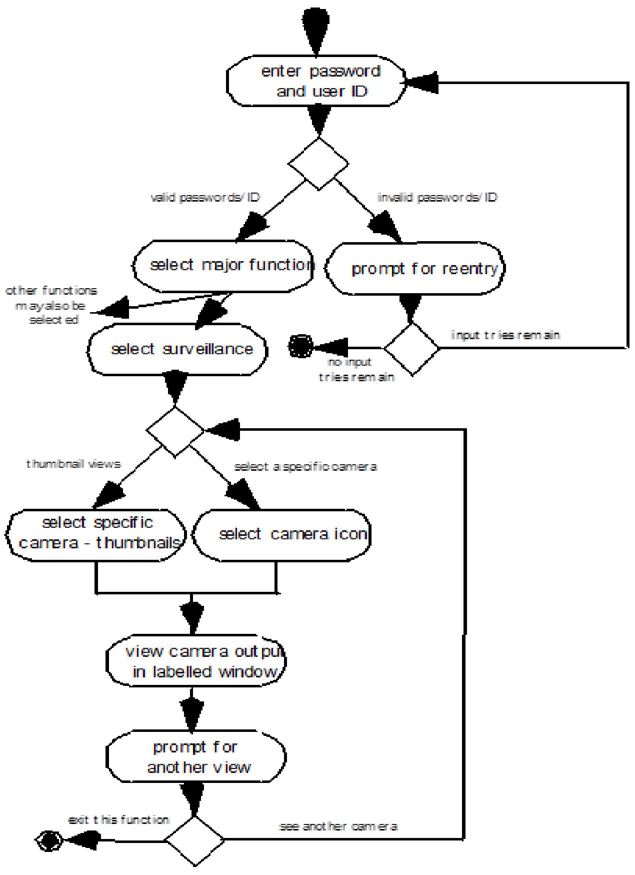
- Requirements Modeling
  - Scenario-based
    - system from the user's point of view
  - Data
    - shows how data are transformed inside the system
  - Class-oriented
    - defines objects, attributes, and relationships
  - Flow-oriented
    - shows how data are transformed inside the system
  - Behavioral
    - show the impact of events on the system states
- Domain Analysis

✧ Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks . . .

Donald Firesmith

⦿ Activity Diagram

Supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario



Chapter 10 : **Requirements Modeling: Class-Based Methods**

⦿ Requirements Modeling Strategies

✧ One view of requirements modeling, called structured analysis, considers data and the processes that transform the data as separate entities.

- Data objects are modeled in a way that defines their attributes and relationships.
- Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

✧ A second approach to analysis modeled, called object-oriented analysis, focuses on

- the definition of classes and
- the manner in which they collaborate with one another to effect customer requirements.

⦿ Manifestations of Analysis Classes

- Analysis classes manifest themselves in one of the following ways:
    - External entities (e.g., other systems, devices, people) that produce or consume information
    - Things (e.g, reports, displays, letters, signals) that are part of the information domain for the problem

   - Occurrences or events (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation
   - Roles (e.g., manager, engineer, salesperson) played by people who interact with the system
   - Organizational units (e.g., division, group, team) that are relevant to an application
   - Places (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function
   - Structures (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects

- Potential Classes
  - Retained information. The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
  - Needed services. The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
  - Multiple attributes. During requirement analysis, the focus should be on "major" information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
  - Common attributes. A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
  - Common operations. A set of operations can be defined for the potential class and these operations apply to all instances of the class.
  - Essential requirements. External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.
- CRC Models
- Class-responsibility-collaborator (CRC) *modeling* [Wir90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. Ambler [Amb95] describes CRC modeling in the following way:
    - A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.
- Class Types
- Entity classes, also called model or business classes, are extracted directly from the statement of the problem (e.g., FloorPlan and Sensor).
- Boundary classes are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
- Controller classes manage a “unit of work” [UML03] from start to finish. That is, controller classes can be designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.
- Collaborations
  - Classes fulfill their responsibilities in one of two ways:

  - A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
  - a class can collaborate with other classes.
- Collaborations identify relationships between classes
- Collaborations are identified by determining whether a class can fulfill each responsibility itself
- three different generic relationships between classes [WIR90]:
  - the is-part-of relationship
  - the has-knowledge-of relationship
  - the depends-upon relationship

Chapter 11 : **Requirements Modeling: Behavior, Patterns, and Web/Mobile Apps**

Behavioral Modeling

- The behavioral model indicates how software will respond to external events or stimuli. To create the model, the analyst must perform the following steps:
  - Evaluate all use-cases to fully understand the sequence of interaction within the system.
  - Identify events that drive the interaction sequence and understand how these events relate to specific objects.
  - Create a sequence for each use-case.
  - Build a state diagram for the system.
  - Review the behavioral model to verify accuracy and consistency.

State Representations

- In the context of behavioral modeling, two different characterizations of states must be considered:
  - the state of each class as the system performs its function and
  - the state of the system as observed from the outside as the system performs its function
- The state of a class takes on both passive and active characteristics [CHA93].
  - A *passive state* is simply the current status of all of an object's attributes.
  - The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing.

The States of a System

- state—a set of observable circum-stances that characterizes the behavior of a system at a given time
- state transition—the movement from one state to another
- event—an occurrence that causes the system to exhibit some predictable form of behavior
- action—process that occurs as a consequence of making a transition

The Content Model

- Content objects are extracted from use-cases
  - examine the scenario description for direct and indirect references to content
- Attributes of each content object are identified
- The relationships among content objects and/or the hierarchy of content maintained by a WebApp
  - Relationships—entity-relationship diagram or UML
  - Hierarchy—data tree or UML

The Interaction Model

- Composed of four elements:
  - use-cases
  - sequence diagrams
  - state diagrams
  - a user interface prototype
- Each of these is an important UML notation and is described in Appendix I