Chapter 5

Understanding Requirements

Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Requirements Engineering-I

- Inception—ask a set of questions that establish...
  - basic understanding of the problem
  - the people who want a solution
  - the nature of the solution that is desired, and
  - the effectiveness of preliminary communication and collaboration
    between the customer and the developer
- Elicitation—determine requirements from all stakeholders
- Elaboration—create an initial 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 use cases (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

Inception

- Identify stakeholders
  - "who else do you think I should talk to?"
- Recognize multiple points of view
- Work toward collaboration
- The first questions
  - Who is behind the request for this work?
  - Who will use the solution?
  - What will be the economic benefit of a successful solution?
  - Is there another source for the solution that you need?

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
- An "evaluation mechanism" can be a white board, flip chart, or wall
  - A free 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

Quality Function Deployment

- Function deployment determines the "value"
  - (as perceived by the customer) of each function required of the system
- Information deployment identifies data objects and events
- Task deployment examines the behavior of the system
- Value analysis determines the relative priority of requirements

Elicitation Work Products

- A specification of need and feasibility
- A bounded statement of scope for the system or product
- A list of stakeholders who participated in requirements elicitation
- A description of the system's technical environment
- A list of requirements (preferably organized by function)
- A description of the technical environment
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- Any prototypes developed to better define requirements

Building the Analysis Model

- Elements of the analysis model
  - Scenario-based elements
    - Functional—processing narratives for software functions
    - Use case—descriptions of the interaction between an "actor" and the system
  - Class-based elements
    - Implied by scenarios
  - Behavioral elements
    - State diagrams
  - Flow-oriented elements
    - Data flow diagrams
Use-Cases
- A collection of user scenarios that describe the thread of usage of a system.
- Each scenario is described from the point-of-view of an "actor"—a person or device that interacts with the software in some way.
- Each scenario answers the following questions:
  - Who is the primary actor, the secondary actor(s)?
  - What is the actor’s goal(s)?
  - What procedures would exist before the story begins?
  - What main tasks or functions are performed by the actor(s)?
  - What interactions might be considered as the story is described?
  - What variations in the actor’s behavior are possible?
  - What system information will the actor acquire, produce, or change?
  - Will the actor have to inform the system about changes in the external environment?
  - What information does the actor derive from the system?
  - Does the actor wish to be informed about unexpected changes?

Analysis Patterns
- Pattern name: A descriptor that encapsulates the essence of the pattern.
- Intent: Describe what the pattern accomplishes or represents.
- Motivation: A scenario that illustrates how the pattern can be used to address the problem.
- Forces and constraints: A description of external issues (forces) that can affect how the pattern is used and also the external issues that will be resolved when the pattern is applied.
- Solution: A description of how the pattern is applied to solve the problem with an emphasis on structural and behavioral issues.
- Consequence: A description of what happens when the pattern is applied and what benefits it provides during the application.
- Design: Discusses how the antipattern can be achieved through the use of known design patterns.
- Known uses: Examples of uses within actual systems.
- Related patterns: One or more analysis patterns that are related to the named pattern because (1) it is commonly used with the named pattern; (2) it is structurally similar to the named pattern; (3) it is a variation of the named pattern.

Validating Requirements - I
- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement rarely necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?

Validating Requirements - II
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function and behavior of the system to be built?
- Have the requirements models been "partitioned" in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

Chapter 6
- Requirements Modeling: Scenarios, Information, and Analysis Classes
  - Software Engineering: A Practitioner's Approach, 7e
  - by Roger S. Pressman
**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 basins
  - 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.

**Domain Analysis**

Software domain analysis is the identification, analysis, and specification of common requirements. It's typically used in 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*

**A Bridge**

- System description
- Analytical model
- Design model

**Rules of Thumb**
- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.
- Each element of the analysis model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.
- Delay consideration of infrastructure and other non-functional models until design.
- Minimize coupling throughout the system.
- Be certain that the analysis model provides value to all stakeholders.
- Keep the model as simple as it can be.

**Domain Analysis**

- Define the domain to be investigated.
- Collect a representative sample of applications in the domain.
- Analyze each application in the sample.
- Develop an analysis model for the objects.

**Requirements Models**

- Scenario-based (system from the user's point of view)
- Data (shows the information domain for the problem)
- 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 as a consequence of external "events")

**Elements of Requirements Analysis**

- Use cases
- Collaboration diagrams
- Sequence diagrams
- Software requirements
- Use case models
- Class models
- Data models

**Scenario-Based Modeling**

"Use-cases are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use-cases)." Ivar Jacobson

1. What should we write about?
2. How much should we write about it?
3. How detailed should we make our description?
4. How should we organize the description?

**What to Write About?**

- Inception and elicitation—provide you with the information you'll need to begin writing use cases.
- Requirements gathering meetings, QFD, and other requirements engineering mechanisms are used to:
  - Identify stakeholders
  - Define the scope of the problem
  - Specify overall system goals
  - Establish priorities
  - Outline all known functional requirements, and
  - Describe the things (objects) that will be manipulated by the system.
- To begin developing a set of use cases, list the functions or activities performed by a specific actor.
How Much to Write About?
- As further conversations with the stakeholders progress, the requirements gathering team develops use cases for each of the functions noted.
- In general, use cases are written first in an informal narrative fashion.
- If more formality is required, the same use case is rewritten using a structured format similar to the one proposed.

Use-Cases
- a scenario that describes a “thread of usage” for a system
- actors represent roles people or devices play as the system functions
- users can play a number of different roles for a given scenario

Developing a Use-Case
- What are the main tasks or functions that are performed by the actor?
- What system information will the actor acquire, produce or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

Swimlane Diagrams

Data Modeling
- examines data objects independently of processing
- focuses attention on the data domain
- creates a model at the customer’s level of abstraction
- indicates how data objects relate to one another

What is a Data Object?
- a representation of almost any composite information that must be understood by software
- composite information—something that has a number of different properties or attributes
- can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).
- The description of the data object incorporates the data object and all of its attributes.
- A data object encapsulates data only—there is no reference within a data object to operations that act on the data.

Data Objects and Attributes
- A data object contains a set of attributes that act as an aspect, quality, characteristic, or descriptor of the object
  - object: automobile
  - attributes: make, model
  - body type
  - price
  - options code
What is a Relationship?
- Data objects are connected to one another in different ways.
  - A connection is established between person and car because the two objects are related.
  - A person owns a car
  - A person is insured to drive a car
- The relationships own and insured to drive define the relevant connections between person and car.
- Several instances of a relationship can exist
- Objects can be related in many different ways

ERD Notation
- One common form:
  - object (n, m) relationship (1, 1) object
- Another common form:
  - object relationship attribute

Building an ERD
- Level 1—model all data objects (entities) and their “connections” to one another
- Level 2—model all entities and relationships
- Level 3—model all entities, relationships, and the attributes that provide further depth

The ERD: An Example

Class-Based Modeling
- Class-based modeling represents:
  - objects that the system will manipulate
  - operations (also called methods or services) that will be applied to the objects to effect the manipulation
  - relationships (some hierarchical) between the objects
  - collaborations that occur between the classes that are defined.
- The elements of a class-based model include classes and objects, attributes, operations, CRC models, collaboration diagrams and packages.

Identifying Analysis Classes
- Examining the usage scenarios developed as part of the requirements model and perform a “grammatical parse” [Abb83]
- Classes are determined by underlying each noun or noun phrase and entering it into a simple table.
- Synonyms should be noted.
- If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a situation, it is part of the problem space.
- But what should we look for once all of the nouns have been isolated?

Manifestation of Analysis Classes
- Analysis classes manifest themselves in one of the following ways:
  - External entities (s, e.g., other systems, devices, people) that produce or consume information
  - Things (e.g., remote displays, buttons, signs) which 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 the system operation
  - Roles (e.g., manager, engineer, subsystem) 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 maintain the context of the problem and the overall function
  - Structures (e.g., sensors, homogeneous 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.
- Retained 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 “updates” information as close 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.

Defining Attributes
- Attributes describe a class that has been selected for inclusion in the analysis model.
- Build two different classes for professional baseball players:
  - For Playing Statistics software: name, position, batting average, fielding percentage, years played, and games played might be relevant
  - For Pension Fund software: average salary, credit toward full vesting, pension plans options chosen, marking address, and the like.
Defining Operations

- Do a grammatical parse of a processing narrative and look at the verbs
- Operations can be divided into four broad categories:
  1. Operations that manipulate data in some way (e.g., adding, deleting, formatting, selecting)
  2. Operations that perform a computation
  3. Operations that inquire about the state of an object, and
  4. Operations that monitor an object for the occurrence of a controlling event.

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:
  1. A CRC model really is 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 report) that the user sees and interacts with as the software is used.
- Controller classes manage a "unit of work" [UML93] from start to finish. That is, controller classes can be designed to manage:
  1. The creation or update of entity objects;
  2. The instantiation of boundary objects as they obtain information from the entity objects;
  3. Complex communication between sets of objects;
  4. Validation of data communicated between objects or between the user and the application.

Responsibilities

- System intelligence should be distributed across classes to best address the needs of the problem.
- Each responsibility should be stated as generally as possible.
- Information and the behavior related to it should reside within the same class.
- Information about one thing should be localized with a single class, not distributed across multiple classes.
- Responsibilities should be shared among related classes, when appropriate.

Collaborations

- Classes fulfill their responsibilities in one of two ways:
  1. A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
  2. A class can collaborate with other classes.
- Collaborations identify relationships between classes.
- Collaborations are identified by determining whether a class can fulfill such a responsibility itself.
- Three different generic relationships between classes [Wir90]:
  1. The is-part-of relationship
  2. The has-aggregation relationship
  3. The depends-upon relationship

Composite Aggregate Class

Associations and Dependencies

- Two analysis classes are often related to one another in some fashion.
  1. In UML these relationships are called associations.
  2. Associations can be refined by indicating multiplicity (the term cardinality is used in data modeling).
- In many instances, a client-server relationship exists between two analysis classes.
  1. In such cases, a client-class depends on the server-class in some way and a dependency relationship is established.

Multiplicity
Dependencies

Analysis Packages

- Various elements of the analysis model (e.g., use-cases, analysis classes) are categorized in a manner that packages them as a grouping.
- The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages.
- Other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package.

Reviewing the CRC Model

- All participants in the review of the CRC model are given a subset of the CRC model and instance cards.
- Cards that correspond to a collaboration (i.e., no reviewer should receive two cards that collaborate).
- All use case scenarios and corresponding use case diagrams should be organized into categories.
- The reviewer leader must make the use case diagrams available before the cards are passed to the participants.
- There is one page for each use case scenario and a page for each use case diagram.
- The group analyzes the contents of the CRC model card and decides to discuss the responsibilities noted on the cards.
- The group discusses whether one or more of the responsibilities satisfies the use case requirements.
- If the responsibilities and collaborations noted on the CRC model card do not accommodate the use case, modifications are made to the card.

Chapter 7

Requirements Modeling: Flow, Behavior, Patterns, and WebApps

Software Engineering: A Practitioner's Approach, 7/e
by Roger B. Pressman

Flow-Oriented Modeling

- Represents how data objects are transformed as they move through the system.
- Data flow diagram (DFD) is the diagrammatic form that is used.
- Considered by many to be an "old school" approach, but continues to provide a view of the system that is unique—it should be used to supplement other analysis model elements.

The Flow Model

- Every computer-based system is an information transform.

Flow Modeling Notation

- **external entity**
- **process**
- **data flow**
- **data store**
External Entity

A producer or consumer of data

Examples: a person, a device, a sensor

Another example: computer-based system

Data must always originate somewhere and must always be sent to something

Process

A data transformer (changes input to output)

Examples: compute taxes, determine area, format report, display graph

Data must always be processed in some way to achieve system function

Data Flow

Data flows through a system, beginning as input and transformed into output.

Data Stores

Data is often stored for later use.

- sensor #
- type, location, age
- sensor number
- sensor data
- look-up sensor data
- report required
- type, location, age
- sensor data

Data Flow Diagramming: Guidelines

- all icons must be labeled with meaningful names
- the DFD evolves through a number of levels of detail
- always begin with a context level diagram (also called level 0)
- always show external entities at level 0
- always label data flow arrows
- do not represent procedural logic

Constructing a DFD—I

- review user scenarios and/or the data model to isolate data objects and use a grammatical parse to determine "operations"
- determine external entities (producers and consumers of data)
- create a level 0 DFD

Level 0 DFD Example

Constructing a DFD—II

- write a narrative describing the transform
- parse to determine next level transforms
- "balance" the flow to maintain data flow continuity
- develop a level 1 DFD
- use a 1:5 (approx.) expansion ratio

The Data Flow Hierarchy

- level 0
- level 1
Flow Modeling Notes

- each bubble is refined until it does just one thing
- the expansion ratio decreases as the number of levels increase
- most systems require between 3 and 7 levels for an adequate flow model
- a single data flow item (arrow) may be expanded as levels increase (data dictionary provides information)

Control Flow Modeling

- Represents "events" and the processes that manage events
- An "event" is a Boolean condition that can be ascertained by:
  - listing all sensors that are "read" by the software,
  - listing all interrupt conditions,
  - listing all "switches" that are actuated by an operator,
  - listing all data conditions,
  - recalling the counter/vernier pair that was used to the processing narrative, review all "control items" as possible CSPEC inputs/outputs.

Control Specification (CSPEC)

- The CSPEC can be:
  - state diagram (sequential spec)
  - state transition table
  - decision tables
  - activation tables

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 [DIAM].
  - 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.

State Diagram for the ControlPanel Class

The States of a System

- state—a set of observable circumstances 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
Behavioral Modeling

- make a list of the different states of a system (How does the system behave?)
- indicate how the system makes a transition from one state to another (How does the system change state?)
- indicate event
- indicate action
- draw a state diagram or a sequence diagram

Patterns for Requirements Modeling

- Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered
- domain knowledge can be applied to a new problem within the same application domain
- the domain knowledge captured by a pattern can be applied by analogy to a completely different application domain
- The original author of an analysis pattern does not "create" the pattern, but rather, discovers it as requirements engineering work is being conducted.
- Once the pattern has been discovered, it is documented

Discovering Analysis Patterns

- The most basic element in the description of a requirements model is the use case.

- A coherent set of use cases may serve as the basis for discovering one or more analysis patterns.

- A semantic analysis pattern (SAP) is a pattern that describes a small set of coherent use cases that together describe a basic generic application." [Far00]

An Example

- This use case implies a variety of functionality that would be refined and elaborated into a coherent set of use cases during requirements gathering and modeling.

- Regardless of how much elaboration is accomplished, the use case(s) suggest(s) a simple, yet widely applicable SAP—the software-based monitoring and control of sensors and actuators in a physical system.

- In this case, the "sensors" provide information about the proximity and video of the environment. The "actuator" is the breaking system of the vehicle. If an object is close to the vehicle,

- But in a more general case, a widely applicable pattern is discovered: Actuator-Sensor Pattern

Actuator-Sensor Pattern—I

- Pattern Name: Actuator-Sensor

- Context: Sensors are devices that are used to detect physical phenomena, typically expressed in the form of a single value or a set of values. There are various types of sensors, such as temperature sensors, pressure sensors, and proximity sensors. These sensors provide input to the system, which is then processed by the system to perform various tasks. The Actuator-Sensor pattern is a pattern that describes the interaction between a sensor and an actuator.

- Problem: The problem is to design a system that can use sensors to detect physical phenomena and use actuators to perform actions based on the sensor data.

- Solution: The solution is to use the Actuator-Sensor pattern, which describes the interaction between a sensor and an actuator.

- Consequences: The Consequences of the Actuator-Sensor pattern are that it provides a reusable solution to the problem of using sensors and actuators in a system. It allows for the easy integration of new sensors and actuators into the system, as well as the modification of existing sensors and actuators. Additionally, it provides a way to encapsulate the behavior of a sensor and actuator pair, making it easier to reuse and maintain the system.

- Applicability: The Actuator-Sensor pattern is applicable in any system where sensors and actuators are present. Examples of such systems include home automation, industrial control systems, and transportation systems.

- Alternatives: There are other patterns that can be used to describe the interaction between sensors and actuators, such as the Sensor-Controller pattern. However, the Actuator-Sensor pattern is more general and can be applied to a wider range of systems.

- Restrictions: There are no significant restrictions on the use of the Actuator-Sensor pattern. It can be used in any system where sensors and actuators are present.
Actuator-Sensor Pattern—III

- See SEPA, 7e for additional information on:
  - Participants
  - Collaborations
  - Consequences

When Do We Perform Analysis?

- In some WebE situations, analysis and design merge. However, an explicit analysis activity occurs when ...
  - the WebApp to be built is large and/or complex
  - the number of stakeholders is large
  - the number of Web engineers and other contributors is large
  - the goals and objectives (determined during formulation) for the WebApp will affect the business bottom line
  - the success of the WebApp will have a strong bearing on the success of the business

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

Data Tree

- [Tree diagram showing data tree structure]

State Diagram

- [State diagram showing state transitions]

Sequence Diagram

- [Sequence diagram showing method sequence]

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

Requirements Modeling for WebApps

- Context Analysis. The full spectrum of context to be provided by the WebApp is identified, including text, graphics and images, video, and audio clips. Data modeling can be used to identify and describe each of the data objects.

  Interaction Analysis. The manner in which the user interacts with the WebApp is described in detail. Use-cases can be developed to provide detailed descriptions of this interaction.

  Functional Analysis. The usage scenarios (use-cases) created as part of interaction analysis define the operations that will be applied to WebApp content and imply other processing functions. All operations and functions are described in detail.

  Configuration Analysis. The environment and infrastructure in which the WebApp resides are described in detail.
The Functional Model

- The functional model addresses two processing elements of the WebApp:
  - user observable functionality that is delivered by the WebApp to end-users
  - the operations contained within analysis classes that implement behaviors associated with the class.
- An activity diagram can be used to represent processing flow.

Navigation Modeling-I

- Should certain elements be easier to reach (require fewer navigation steps) than others? What is the priority for presentation?
- Should certain elements be emphasized to force users to navigate in their direction?
- How should navigation errors be handled?
- Should navigation to related groups of elements be given priority over navigation to a specific element?
- Should navigation be accomplished via links, via search-based access, or by some other means?
- Should certain elements be presented to users based on the context of previous navigation actions?
- Should a navigation log be maintained for users?

Navigation Modeling-II

- Should a full navigation map or menu (as opposed to a single "back" link or directed pointer) be available at every point in the user's interaction?
- Should navigation design be driven by the most commonly expected user behaviors or by the perceived importance of the defined WebApp elements?
- Can a user "shrink" the previous navigation through the WebApp to expedite future usage?
- For which user category should optimal navigation be designed?
- How should links external to the WebApp be handled? overlaying the existing browser window? as a new browser window? as a separate frame?

Chapter 8

- Design Concepts

  Software Engineering: A Practitioner's Approach, 7/e
  by Roger S. Pressman

  For non-profit educational use only

Design

- Mitch Kapor, the creator of Lotus 1-2-3, presented a "software design manifesto" in Dr. Dobbs Journal. He said:
  - Good software design should exhibit:
  - Firmness: A program should not have any bugs that inhibit its function.
  - Commodity: A program should be suitable for the purposes for which it was intended.
  - Delight: The experience of using the program should be pleasurable one.

Analysis Model -> Design Model

Design and Quality

- the design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- the design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- the design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.
Quality Guidelines

- A design should exhibit an architecture that (a) is correct, (b) can be changed in an evolutionary fashion.
- A design should be modular, that is, the software should be logically partitioned into smaller or interchangeable parts.
- A design should be represented using a notation that effectively conveys its meaning.

FURPS Quality Factors

- Functionality – feature set and program capabilities
- Usability – human factors (aesthetics, consistency, documentation)
- Reliability – frequency and severity of failure
- Performance – processing speed, response time, throughfrout, efficiency
- Supportability – maintainability (extensibility, adaptability, serviceability), testability, compatibility, configurability.

Design Principles

- The design process should not suffer from a "tunnel vision;"
- The design should be translated to the analysis model.
- The design should not detract from the user interface.
- The design should minimize the "intelligence division" (DAVIII) between the software and the problem as it exists in the real world.
- The design should be flexible and adaptable.
- The design should be structured to accommodate change.
- The design should be structured so that it can be decomposed into smaller, more manageable parts.
- The design should be structured so that it can be decomposed into smaller, more manageable parts.
- The design should be reviewed to minimize conceptual (semantic) errors.

From Davis (DAVIII)

Design - Task Set

- Examine information domain model and design appropriate data model and structures.
- Select an architectural pattern appropriate to the software.
- Partition the analysis model into design subsystems.
- Design any interface required with external systems or devices.
- Design user interface.
- Conduct component level design.
- Develop deployment model.

Design - Fundamental Concepts

- Abstraction—data, procedure, control.
- Architecture—overall structure of the software.
- Patterns—conveys the essence of a proven design solution.
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces.
- Modularity—compartmentalization of data and functions, that make low program intellectually manageable.
- Hiding—controlled interfaces.
- Functional independence—single-minded function and low coupling.
- Refinement—abstraction of detail for all abstractions.
- Aspects—a mechanism for understanding how global requirements affect design.
- Refactoring—an organization technique that simplifies the design.
- DOR design concept—Appendix II.
- Design Classes—provide design detail that will enable analysis, classes to be implemented.

Data Abstraction

Implemented as a data structure.

Procedural Abstraction

"The overall structure of the software and the ways in which that structure provides conceptual integrity for a system." [Shah, 1981]

Architecture

Parts Of Architectural Designs

- Structural properties—The aspect of the architectural design representation that deals with the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with each other. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interface via the invocation of methods.
- Extra-functional properties—The aspect of the architectural design representation that deals with the components of a system (e.g., objects, filters) and the manner in which those components are packaged and interact with each other. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interface via the invocation of methods.

Patterns

"Proven Knowledge of a problem defined earlier" Design Pattern Template

Pattern name—describes the essence of the pattern in a short but expressive name.
Intent—describes the pattern and what it does.
Also known as—Lists any synonyms for the pattern.
Motivation—provides an example of the problem.
Applicability—notes specific design situations in which the pattern is applicable.
Structure—describes the classes that are required to implement the pattern.
Participants—describes the responsibilities of the classes that are required to implement the pattern.
Collaborations—describes how the participants collaborate to carry out their responsibilities.
Consequences—describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented.
Related patterns—cross-references related design patterns.
Separation of Concerns
- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- A concern is a feature or behavior that is specified as part of the requirements model for the software.
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

Modularity
- "Modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
- The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

Information Hiding
- Module: algorithm, data structure, details of external interface, resource allocation policy.
- Controlled interface: interface to clients.
- "Secret": a specific design decision.

Why Information Hiding?
- Reduces the likelihood of "side effects".
- Limits the global impact of local design decisions.
- Emphasizes communication through controlled interfaces.
- Discourages the use of global data.
- Leads to encapsulation—an attribute of high quality design.
- Results in higher quality software.

Modularity: Trade-offs
- What is the "right" number of modules for a specific software design?

Stepwise Refinement
- If the door doesn't turn, then take key out, find correct key, insert in lock, end.
- Pull/pull door move out of way, end repeat.

Sizing Modules: Two Views
- Functional Independence
  - Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
  - Cohesion is an indication of the relative functional strength of a module.
    - A cohesive module performs a single task, requiring little interaction with other components in other parts of a program.
    - Stated simply, a cohesive module should (ideally) do just one thing.
  - Coupling is an indication of the relative interdependence among modules.
    - Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Aspects
- Consider two requirements, A and B. Requirement A crosstalk requirement B "if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account. [Rus04]
- An aspect is a representation of a cross-cutting concern.
Aspects—An Example

Consider the requirements for the SafeHomeAssured.com WebApp. Requirement A is described in the use-case Access camera surveillance via the Internet. A design refinement would focus on those issues that would enable a registered user to access video from cameras placed throughout a space. Requirement B is a generic security requirement that states that a registered user must be validated prior to using SafeHomeAssured.com. This requirement is applicable for all functions that are available to registered SafeHome users. As design refinement occurs, A' is a design representation for requirement A and B' is a design representation for requirement B. Therefore, A' and B' are representations of concerns, and B' cross-cuts A'.

An aspect is a representation of a cross-cutting concern. Therefore, the design representation, B', of the requirement, a registered user must be validated, is using SafeHomeAssured.com, is an aspect of the SafeHome WebApp.

Refactoring

Refactoring ([FOW98] defines refactoring in the following manner:

- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code (aspects) yet improves its internal structure.

- When software is refactored, the existing design is examined for:
  - Redundancy
  - Unused design elements
  - Inefficient or unnecessary algorithms
  - Poorly constructed or inappropriate data structures
  - Or any other design flaw that can be corrected to yield a better design.

OO Design Concepts

- Design classes
- Entity classes
- Boundary classes
- Controller classes

Inheritance—All responsibilities of a super class are immediately inherited by all subclasses

Messages—stimulate some behavior to occur in the receiving object.

Polygamy—a characteristic that greatly reduces the effort required to extend the design.

Design Classes

- Analysis classes are refined during design to become entity classes.
- Boundary classes are developed during design 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 are designed with the responsibility of managing the way entity objects are represented to users.

Design Model Elements

- Data elements
  - Data model or database architecture
- Architectural elements
  - Application domain
  - Analysis classes, their relationships, collaborations, and behavior
  - Design model or database architecture
- Component elements
  - Process models/other system requirements, networks, or other considerations or concerns of information
  - Internal interfaces between various design components

Component Elements
Deployment Elements

Chapter 9

- Architectural Design
  
  Software Engineering: A Practitioner’s Approach, 7/e
  by Roger S. Pressman

- What is Architecture?
- Software Architecture
- Why is Architecture Important?
- Architectural Descriptions
- Architectural Styles
  - Data-centered architectures
  - Data flow architectures
  - Call and return architectures
  - Object-oriented architectures
  - Layered architectures
- Architectural Patterns
- Architectural Design
- Analyzing Architectural Design

What is Architecture?

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

1. analyze the effectiveness of the design in meeting its stated requirements,
2. consider architectural alternatives at a stage when making design changes is still relatively easy, and
3. reduce the risks associated with the construction of the software.

Architecture

- The architecture of a system is a comprehensive framework that describes its form and structure – its components and how they fit together. Jerrold Grochow
- Architectural design represents the structure of the data and program components required to build a computer-based system.

Software Architecture

- The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and relationships among them. Bass, Clements & Kazman

Why is Architecture Important?

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture “constitutes a relatively small, intellectually graspable mode of how the system is structured and how its components work together” [BA093].

Architectural Descriptions

- The IEEE Computer Society has proposed IEEE-Std-1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems, [IEEE90]:
  - to establish a conceptual framework and vocabulary for use during the design of software architecture,
  - to provide detailed guidelines for representing an architectural description, and
  - to encourage sound architectural design practices.
- The IEEE Standard defines an architectural description (AD) as “a collection of products to document an architecture.”
- The description itself is represented using multiple views, where each view is “a representation of a whole system from the perspective of a related set of [stakeholder] concerns.”

Architectural Decision Description Template

- Used to document each major architectural decision for later review by stakeholders who want to understand the proposed architectural decision
  - Design issues – describe architectural design issues to be addressed
  - Resolution – State the approach you've chosen to address issues
  - Category – Specify the design category (data, content and component design)
  - Assumptions – any assumptions that helped
  - Constraints – specify any environmental constraints that helped
  - Alternatives – describe architectural design alternatives
  - Argument – state why you chose the resolution
  - Implications – indicate the design consequences
  - Related decisions – what other decisions are related to this decision
  - Work products – where this decision is going to be reflected in the architecture
  - Notes – reference
Architectural Genres
- Genre implies a specific category within the overall software domain.
- Within each category, you encounter a number of subcategories.
- For example, within the genre of buildings, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial buildings, warehouses, and so on.
- Within each general style, more specific styles might apply. Each style would have a structure that can be described using a set of predictable patterns.

Architectural Styles
Each style describes a system category that encompasses:
- A set of components (e.g., a database, computational modules) that perform a function required by a system,
- A set of connectors that enable "communication, coordination, and cooperation" among components,
- Constraints that define how components can be integrated to form the system,
- And semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.
- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

Data-Centered Architecture

Data Flow Architecture

Call and Return Architecture

Layered Architecture

Architectural Patterns
- Consuming—Applications must handle multiple tasks in a manner that simulates panekism
- Operating system process management pattern
- Task scheduler pattern
- Transaction—Data persists through the execution of the process that created it. Two patterns are common:
- A database management system pattern that applies the storage and retrieval capability of SMS to the application architecture
- An application level persistence pattern that uses persistence features into the application architecture
- Distribution—The manner in which systems or components within systems communicate with one another in a distributed environment
- A client acts as a "middle-man" between the client component and a server component.

Architectural Design
- The software must be placed into context
- The design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction
- A set of architectural archetypes should be identified
- An archetype is an abstraction (similar to a class) that represents one element of system behavior
- The designer specifies the structure of the system by defining and refining software components that implement each archetype

Architectural Context

...
Analyzing Architectural Design

1. Collect scenarios.
2. Elaborate requirements, constraints, and environment description.
3. Describe the architectural style/patterns that have been chosen to address the scenarios and requirements:
   - modules view
   - process view
   - data flow view
4. Evaluate quality attributes considered to each attribute in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.

Architectural Complexity

- the overall complexity of a proposed architecture is assessed by considering the dependencies between components within the architecture [Zha98]
  - Sharing dependencies represent dependence relationships among components who use the same resource or producer who produce for the same consumer.
  - Flow dependencies represent dependence relationships between producers and consumers of resources.
  - Constrained dependencies represent constraints on the relative flow of control among a set of activities.

ADL

- Architectural description language (ADL) provides a semantics and syntax for describing a software architecture.
- Provide the designer with the ability to:
  - decompose architectural components
  - compose individual components into larger architectural blocks
  - represent interfaces (connection mechanisms) between components.

An Architectural Design Method

customer requirements
"four bedrooms, three baths, lots of glass ..."

architectural design

Deriving Program Architecture

Partitioning the Architecture

- "horizontal" and "vertical" partitioning are required
Horizontal Partitioning
- Define separate branches of the module hierarchy for each major function
- Use control modules to coordinate communication between functions

Structured Design
- Objective: to derive a program architecture that is partitioned
- Approach:
  - A DFD is mapped into a program architecture
  - The PSPEC and STD are used to indicate the content of each module
- Notation: Structure chart

Vertical Partitioning: Factoring
- Design so that decision making and work are stratified
- Decision making modules should reside at the top of the architecture

Why Partitioned Architecture?
- Results in software that is easier to test
- Leads to software that is easier to maintain
- Results in propagation of fewer side effects
- Results in software that is easier to extend

Flow Characteristics
- Transform flow

General Mapping Approach
- Isolate incoming and outgoing flow boundaries, for transaction flows, isolate the transaction center
- Working from the boundary outward, map DFD transforms into corresponding modules
- Add control modules as required
- Refine the resultant program structure using effective modularity concepts

General Mapping Approach
- Isolate the transform center by specifying incoming and outgoing flow boundaries
- Perform "first-level factoring."
- The program architecture derived using this mapping results in a top-down distribution of control
- Factoring leads to a program structure in which top-level components perform decision-making, and lower-level components perform most input, computation, and output work
- Middle-level components perform some control and do moderate amounts of work
- Perform "second-level factoring."

Transform Mapping
- "Transform" mapping

Factoring
First Level Factoring

Second Level Mapping

Chapter 11

User Interface Design

Software Engineering: A Practitioner's Approach, 3/e
by Roger S. Pressman

Interface Design

Easy to learn?
Easy to use?
Easy to understand?

Interface Design

Typical Design Errors
- lack of consistency
- too much memorization
- no guidance / help
- no context sensitivity
- poor response
- Arcane/unfriendly

Golden Rules

- Place the user in control
- Reduce the user's memory load
- Make the interface consistent

1. Place the User in Control
- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction.
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen.

2. Reduce the User's Memory Load
- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real world metaphor.
- Disclose information in a progressive fashion.

Make the Interface Consistent

- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.
3. User Interface Design Models
- User model — a profile of all end users of the system
- Design model — a design realization of the user model
- Mental model (system perception) — the user’s mental image of what the interface is
- Implementation model — the interface “look and feel” coupled with supporting information that describe interface syntax and semantics

Interface Analysis
- Interface analysis means understanding
  1. The people (end-users) who will interact with the system through the interface,
  2. The tasks that end-users must perform to do their work,
  3. The content that is presented as part of the interface,
  4. The environment in which those tasks will be conducted.

User Interface Design Process
1. Analysis of Physical Work Environment
2. Interface Design
3. Interface Construction
4. Validation Considerations

User Analysis
- Are users trained professionals, technicians, clerks, or manufacturing workers?
- What level of formal education does the average user have?
- What level of formal education does the average user have?
- Are users capable of learning from written material or have they expressed a desire for on-the-job training?
- Are users experts in the use of the product?
- What is the age range of the user community?
- What is the sex ratio of the user community?
- Will users be represented predominantly by one gender?
- How are users compensated for the work they perform?
- Do users work on a fixed hour or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology that is behind the interface?

Task Analysis and Modeling
- Answers the following questions:
  - What work will the user perform in specific circumstances?
  - What tasks and subtasks will be performed as the user does the work?
  - What specific problem domain objects will the user manipulate as work is performed?
  - What is the sequence of work tasks in the workflow?
  - What is the hierarchy of tasks?
  - Use-cases define basic interaction.
  - Task elaboration refines interactive tasks.
  - Object elaboration identifies interface objects (classes).
  - Workflow analysis defines how a work process is completed when several people (and roles) are involved.

Display Content
- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large data sets?
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color be used to enhance understanding?
- How will error messages and warning messages be presented to the user?

Interface Design Steps
- Using information developed during interface analysis, define interface objects and actions (operations).
- Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
- Depict each interface state as it will actually look to the end-user.
- Indicate how the user interprets the state of the system from information provided through the interface.

Design Issues
- Response Time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility
- Internationalization
WebApp Interface Design
- Where am I? The interface should:
  - provide an indication of the WebApp that has been accessed
  - inform the user of their location in the content hierarchy
- What can I do now? The interface should always help the user understand the current options
  - what functions are available?
  - what links are live?
  - what content is relevant?
- Where have I been, where am I going? The interface must facilitate navigation,
  - provide a "map" (implemented in a way that is easy to understand) of where the user has been and what path may be taken to move elsewhere within the WebApp.

Effective WebApp Interfaces
- Bruce Tognozi [Tog01] suggests…
  - Effective interfaces are visually apparent and forgiving, instilling in their users a sense of control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.
  - Effective interfaces do not concern the user with the inner workings of the system. Work is carefully and continuously saved, with full option for the user to undo any activity at any time.
  - Effective applications and services perform a maximum of work, while requiring a minimum of information from users.

Interface Design Principles-I
- Anticipation: A WebApp should be designed so that it anticipates the user's next move.
- Communication: The interface should communicate the status of any activity initiated by the user.
- Consistency: The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- Controlled autonomy: The interface should facilitate user movement throughout the WebApp, but should do so in a manner that enables navigation conventions that have been established for the application.
- Efficiency: The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client/server environment that executes it.

Interface Design Principles-II
- Focus: The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- Fit: A task’s time to acquire a target is a function of the distance to and size of the target.
- Human interface objects: A vast library of reusable human interface objects has been developed for WebApps.
- Latency reduction: The WebApp should use multithreading in a way that lets the user proceed with work as if the operation has been completed.
- Learnsability: A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.

Interface Design Principles-III
- Interact with preserved integrity: A WebApp should be designed so that it will not be affected by any errors or interruptions.
- Reliability: All information presented through the interface should be verifiable by the user.
- Feedback: When appropriate, the state of the user interaction should be tracked and stored so that a user can log off and return later to pick up where she left off.
- Visible navigation: A well-designed WebApp interface provides the "look" that users are in the same place, with the work brought to them.

Interface Design Workflow-I
- Review information contained in the analysis model and refine as required.
- Develop a rough sketch of the WebApp interface layout.
- Map user objectives into specific interface actions.
- Define a set of user tasks that are associated with each action.
- Storyboard screen images for each interface action.
- Refine interface layout and storyboards using input from aesthetic design.

Mapping User Objectives

Interface Design Workflow-II
- Identify user interface objects that are required to implement the interface.
- Develop a procedural representation of the user's interaction with the interface.
- Develop a behavioral representation of the interface.
- Describe the interface layout for each state.
- Refine and review the interface design model.

Aesthetic Design
- Don't be afraid of white space.
- Emphasize content.
- Organize layout elements from top-left to bottom-right.
- Group navigation, content, and function geographically within the page.
- Don't extend your real estate with the scrolling bar.
- Consider resolution and browser window size when designing layout.
Design Evaluation Cycle

Diagram showing a cycle with steps labeled and arrows indicating the flow of evaluation.