SE463 Exam Notes

December 8, 2024

TLDR - Key Definitions and Concepts

Requirements Types:

- **D Requirements (Scope Determined):** Define the specifics of a given scope. Essential to "build the system right." Very expensive to fix if discovered late (10-200x). *Examples:* Exception handling, edge cases, implied functionalities (e.g., denominator cannot be 0 in division).
- **G Requirements (Scope Determining):** Determine the overall project scope. Needed to build "the right system." Can change and expand the project's scope. *Examples:* Adding new features (e.g., adding a "log" function to a calculator).

Domain Modeling:

- **Domain Ignoramus:** A computing expert with limited knowledge of the project's application domain. They avoid tacit assumptions, ask clarifying questions, think outside the box, find "holes" in documentation, and avoid channeling. A mixed team of domain ignoramuses and domain experts is hypothesized to generate better requirements.
- **SYS (System Model):** Represents the system being built. Should have only one item not in the interface (INTF), representing the "black box" that hides implementation details.
- **ENV (Environment Model):** Includes anything that might affect or be affected by the system through the interface. Should include items from assumptions, exceptions, variations, and domain assumptions (D).
- **INTF (Interface Model):** Includes items that a user must be aware of, control, or sense to use the system's features and understand its responses. The level of detail depends on the system's purpose and user interaction.
- **Shared Phenomena:** Key to defining the interface and interactions between the system and its environment.

Requirements Engineering Concepts:

- **ZJVF (Zave-Jackson Validation Formula):** $D, S \vdash R$. *Means:* Specification (S) with Domain Knowledge (D) satisfies Requirements (R). A guideline, not a formal proof due to real-world complexities.
- **Use Case (UC):** A specific way a user interacts with a system to achieve a goal. Expressed as a simple imperative sentence (e.g., "Insert a coin into the coinSlot").
- **Scenario:** A particular sequence of interaction steps between a user and a system for a specific use case. A single use case can have many scenarios.
- **NFR (Non-Functional Requirement):** Attributes and characteristics of a system, not specific functions. Describe *how* the system performs, not *what* it does. Often called "quality attributes" or "the -ilities" (e.g., performance, reliability, usability, security). Can conflict and need prioritization.
- **SRS (Software Requirements Specification):** Defines software requirements in detail for developers, testers, and customers. Follows IEEE standard: Section 1 (Introduction), Section 2 (Overall Description), Section 3 (Specific Requirements - the "meat"). Section 3 has detailed input/output descriptions and UML diagrams.

Temporal Logic:

- **LTL (Linear Temporal Logic):** A system for describing how system states change over time. Focuses on the order of events.
- **Temporal Connectives:**
  - **Henceforth ($\Box$):** $\Box f$ means f is true now and in all future states.
  - **Eventually ($\Diamond$):** $\Diamond f$ means f is true now or in some future state.
  - **Next State ($\bigcirc$):** $\bigcirc f$ means f is true in the next state.
  - **Until ($\mathcal{U}$):** $f \mathcal{U} g$ means g will eventually be true, and f is true until then.
  - **Unless ($\mathcal{W}$):** $f \mathcal{W} g$ means f is true indefinitely or until g becomes true (but g is not guaranteed to become true).

  Note: When scanning a temporal logic formula from left to right, each temporal operator introduces a new implied bound time variable which is at the same time or later than the previous.

UI and Documentation:

- **UI (User Interface):** Should be specified during requirements, not left to implementers. Bad UIs cause user confusion. Design with functional requirements. Validate with usability testing.
- **Platt's Law:** "Know Thy User, for He Is Not Thee."
- **Why Software Sucks (Platt):** The core reason is bad UIs, often designed by programmers for themselves, not for the typical user.
- **UM (User's Manual):** A well-written UM, focusing on user perspective and functionality, can be an effective RS for many systems, especially those with a UI focus. Not suitable for all, especially those with complex, hidden NFRs or no direct user interaction.

Other Important Concepts:

- **Prescriptive Specifications:** Model-based, describe system behavior state-by-state based on inputs (e.g., state diagrams).
- **Descriptive Specifications:** Describe system-wide properties over time, using logic (e.g., temporal logic).
- **Phenomenon A:** Upfront requirements analysis leads to faster completion, but often impractical.
- **Phenomenon B:** New requirements emerge over time, requiring iterative development.

1 Course Introduction and Grading

TLDR The course focuses on software requirements specification and analysis. The grading breakdown is 40% for the project, 10% for assignments, and 50% for the final exam. For help, contact `se463 ATT uwaterloo DOTT ca`.

- **Grading:** Project (40%), Assignments (10%), Final Exam (50%).
- **Website:** `http://www.student.cs.uwaterloo.ca/~se463`
- **Contact:** `se463 ATT uwaterloo DOTT ca`
- **Instructor:** Daniel Berry (available by appointment, prefers Zoom)

2 Scope Determining (G) vs. Scope Determined (D) Requirements

TLDR D requirements define the specifics of a given scope, are essential to "build the system right," and are very expensive to fix if found late. G requirements determine the overall project scope, are needed to build "the right system," and can change, expanding the project's scope. Agile is good for G, but each sprint needs thorough upfront RE to find all D requirements for that sprint's scope.

- **Phenomenon A (Waterfall):** Upfront requirements analysis leads to faster completion, but is often impractical in real-world scenarios.
- **Phenomenon B (Agile):** New requirements emerge over time, necessitating iterative development.
- **D Requirements:**
  - **Definition:** Define the specifics of a given scope. Necessary to build the system correctly.
  - **Cost:** Expensive to fix if discovered late (10-200x).
  - **Examples:** Exception handling, edge cases, implied functionalities (e.g., the denominator cannot be 0 in a division operation).
- **G Requirements:**
  - **Definition:** Determine the overall project scope. Needed to build the correct system.
  - **Change:** Can change and expand the project's scope.**
  - **Examples:** Adding new features (e.g., adding a "log" function to a calculator).
- **Impact on Development:**
  - **Waterfall:** Better for projects where D requirements are dominant.
  - **Agile:** More suitable when G requirements are prevalent. Modified Agile approach: Iterate to define the scope of each sprint (deal with G requirements), then perform thorough upfront RE within each sprint to discover all D requirements for that sprint's scope.

3 Identifying Classes and Relationships (UML Class Diagrams)

TLDR In requirements engineering, classes are used to model problem domain concepts. The world is divided into the System (what you are building) and the Environment (what it interacts with). The shared interface is where they meet. Nouns often become classes, verbs become operations, and adjectives/adverbs become attributes or non-functional requirements (NFRs). The requirements engineer builds an initial model to understand the problem, often "encoding ignorance" by including unfamiliar terms as placeholders to be clarified later.

- **Classes in Requirements Engineering:** Classes represent key concepts in the problem domain and their relationships, helping to model the system's environment and interactions. They are not just for implementation.
- **Dividing the World:**
  - **System:** What you are building (the software).
  - **Environment:** What the system interacts with (the real world).
  - **Shared Interface:** Where the system and environment meet; a crucial area for analysis.
- **Identifying Classes:** Start by identifying nouns, verbs, adjectives, and adverbs in the problem description.
  - Nouns often become classes.
  - Verbs often become operations.
  - Adjectives and adverbs often become attributes or non-functional requirements.

- **Shared Phenomena:** Focus on the phenomena shared between the system and its environment. These are key to defining the interface and interactions.
- **Use Cases:** Use cases are identified as operations within the interface classes, accessible to and performed by actors (elements in the environment).
- **Role of the Requirements Engineer:** The requirements engineer (REng) builds an initial model, often with incomplete information. This model clarifies the problem domain, identifies areas needing more information, and facilitates communication with the client. Don't be afraid to "encode ignorance" by placing unfamiliar terms from the problem description into the model as placeholders for future clarification.
- **Multiplicities:**
  - **Class Multiplicities:** Symbols like , +, n, 0, 1, 2 indicate how many instances of one class can be related to instances of another class.
  - **Arc Multiplicities:** Numbers on association lines, where each end of the arc has its own multiplicity.
  - **Example:** For A —m—n—B:
    - Each A *verbs* n Bs.
    - Each B *is verbed by* m As.

    Remember to describe the relationship from both directions, using a relevant verb.
- **Examples:** Turnstile system, WhenIsGood.net, Car, Classroom Podium.

4 Importance of Ignorance

TLDR A "domain ignoramus" is a computing expert who is unfamiliar with the specific application domain of a project. They are beneficial because they challenge assumptions, ask clarifying questions, find holes in documentation and systems, and improve the requirements elicitation process.

- **Domain Ignoramus:**
  - **Definition:** A computing expert with limited knowledge of the project's application domain. They are language-smart, able to detect inconsistencies even without full domain understanding.
  - **Advantages:**
    - Avoids tacit assumptions.
    - Asks clarifying questions.
    - Thinks outside the domain box.
    - Finds "holes" in documentation and systems.
    - Avoids channeling (getting stuck in familiar patterns).
- **Hypothesis:** A mixed team composed of both domain ignoramuses and domain experts generates better requirements than teams composed solely of either group.
- **Immigration to projects:** Domain ignorance can be helpful for new project members, allowing them to be immediately useful and learn the domain gradually.

5 Domain Modeling: SYS, ENV, and INTF

TLDR Model the system from the user's perspective. The system model (SYS) should have only one item not in the interface (INTF), representing the "black box" that hides implementation details. The environment (ENV) includes anything that might affect or be affected by the system through the interface. The interface (INTF) should include items that a user must be aware of, control, or sense to use the system's features and understand its responses.

- **SYS:**

  - The system model should have only one item not in the interface, representing the black box that hides implementation details. This is referred to as the "X system".
  - If there are more than one item in the system (SYS) that is not also in the environment (ENV), you probably have too many.
  - Decomposition of the system is acceptable later, but the pieces should be apparent to the user and part of the ENV, not just implementation details.

- **ENV:**
  - Include anything that might affect or be affected by the system through the interface.
  - Items mentioned in assumptions, exceptions, variations, and domain assumptions (D) should be included.
  - It's better to include too much than too little; you can always remove items later.

- **INTF:**
  - The interface should include items that a user must be aware of, control, or sense to use the system's features and understand its responses.
  - What constitutes an INTF item depends on the system's features.
  - For an operating system, items like the screen, keyboard, and mouse are appropriate.
  - For a specific application, specific buttons, menu items, and windows are appropriate.
  - The appropriate level of detail for INTF items depends on what the system is supposed to do and how it appears to the user.

Most Important Overall Advice

- It depends on what the system is supposed to do and how it appears to the user!!!
- A good place to start is your team's abstract, because you were careful to write it at the user level so as to attract non-techie investors. :-)
- Remember that the abstract is written at the user level, so it is a good place to start for determining what should be in the INTF.

6 Reference Model for Requirements Engineering

TLDR The reference model divides the world into the System (SUD), the Environment, and the Interface (shared phenomena between the two). Requirements (R) describe desired changes to the environment, expressed in terms of environment phenomena. The Specification (S) describes the proposed behavior of the system, expressed in terms of interface (shared) phenomena. Domain Knowledge (D) represents guaranteed properties of the environment that are assumed to be true. The Zave-Jackson Validation Formula (ZJVF) states: D, S $\vdash$ R, meaning that the specification (S), given the domain knowledge (D), is sufficient to satisfy the requirements (R).

- **Reference Model:** Divides the world into the *System* (SUD), the *Environment*, and the *Interface* (shared phenomena between the two). The boundary of the environment is fuzzy.
- **Requirements (R):** Statements of *desired changes* to the environment, expressed solely in terms of environment phenomena. They describe *what* the system must do.
- **Specification (S):** Description of the *proposed behavior* of the system. Expressed in terms of interface (shared) phenomena. Explains *how* the system will satisfy the requirements. Avoids design and implementation bias.
- **Domain Knowledge (D):** Guaranteed properties of the environment *assumed to be true.* Necessary for the system to fully meet the requirements.
- **Zave-Jackson Validation Formula (ZJVF):** D, S $\vdash$ R

  - **Meaning:** The specification (S), given the domain knowledge (D), is sufficient to satisfy the requirements (R).
  - This formula is a guideline, not a precise, formal proof due to the complexities of the real world.

- **Key Distinctions:**
  - **Requirements vs. Specification:** Requirements focus on *what* needs to be changed in the environment, while the specification describes *how* the system will achieve those changes.
  - **Environment vs. Interface:** The environment encompasses all relevant real-world aspects, while the interface only includes the phenomena shared by the system and environment.

- **Context Diagram:** A graphical model representing the environment, system, and their components, highlighting shared phenomena. Also called a Domain Model, Class Model, or Class Diagram. Connections in the diagram represent shared phenomena.

- **Deriving Specifications:** The process of identifying actions, functions, and constraints on shared phenomena (the specification) that satisfy the requirements.

- **Correctness and ZJVF:** If the ZJVF (D, S $\vdash$ R) cannot be "proven" (argued convincingly), then:
  1. Strengthen the specification.
  2. Strengthen the domain knowledge.
  3. Weaken the requirements.

- **Uncertainty:** The real world doesn't perfectly behave like any model. D and R are approximations of the real world. Simpler models are easier to work with but less accurate. Formal proof of correctness is difficult; arguments and justifications are used instead.

- **Important Considerations:**
  - **Scoping:** The environment and interface should be scoped to include only what's necessary to express the requirements and specification—no more, no less.
  - **Humans in the System:** If humans are part of the system, their behavior must be specified, with the understanding that they may not always behave as expected.
  - **User Interface:** The user interface is part of the specification, *not* an implementation detail. It describes how the user interacts with the system through shared phenomena.

7 Use Cases and Scenarios

TLDR A Use Case (UC) is a general way a user interacts with a system to achieve a goal. A Scenario is a specific instance of a use case, represented as a sequence of interaction steps. A Typical Scenario is the "normal" or most common sequence. Alternatives (As) are variations of a UC that achieve the main goal through different steps. Exceptions (Es) handle conditions deviating from the typical scenario. Misuse Cases are use cases that a system should be protected against (e.g., security breaches).

- **Use Case (UC):**
  - **Definition:** A specific way a user interacts with a system (S) to achieve a goal.
  - Expressed as a simple imperative sentence (e.g., "Insert a coin into the coinSlot").

- **Scenario:**
  - **Definition:** A particular sequence of interaction steps between a user and a system for a specific use case.
  - A single use case can have many scenarios.

- **Typical Scenario:** The "normal" or most common sequence of steps within a use case, as defined by stakeholders.

- **Alternatives (As):** Variations of a UC that achieve the main goal through different steps or fail to achieve the goal although following most of the steps. They are considered sub-use-cases.
- **Exceptions (Es):** Sub-use-cases that handle conditions deviating from the typical scenario or other covered sub-use-cases.
- **Misuse Cases:** Use cases that a system should be protected against (e.g., security breaches).
  - **Example:** For UC "Insert a coin into the coinSlot", a misuse case could be "Insert a counterfeit coin into the coinSlot".
- **Sub-use-cases:** Shared subsequences of steps within different scenarios of a use case. Sub-use-cases may be worth treating as use cases of their own if they can be used by other use cases.
- **Distinctions:**
  - **Use Case vs. Scenario:** A use case is a general way of using the system, while a scenario is a specific instance of that use.
  - **Alternatives vs. Exceptions:** Both are variations, but the distinction isn't critical as long as you find all of them. Think of them as prompts.
- **Relationships:**
  - A single use case contains many scenarios.
  - Scenarios of a use case share subsequences of steps.
  - Alternatives and exceptions are variations within a use case.
- **Purpose and Applications of Scenarios and Use Cases:**
  - Requirements elicitation.
  - User manuals/help documentation.
  - Test case generation.
  - User validation.
  - Active specification reviews.
  - Rapid prototyping.
- **Advantages (according to Jo Atlee):**
  - Simple and easy to create.
  - Understandable by clients.
  - Reflect user's essential requirements.
  - Separate normal from exceptional behavior.
- **Disadvantages (according to Jo Atlee):**
  - Don't scale well in size or complexity.
- **Example:**
  - **UC:** Insert a coin into the coinSlot
    - **Typical Scenario:**
      1. Visitor drops a coin into the coinSlot.
      2. Coin is accepted.
      3. CoinSlot informs the system that someone paid.
    - **As & Es:** Fail to insert coin, insert a coin that is returned unused, hit the coinSlot in anger, curse the coinSlot in anger.
  - **UC:** Push and walk through the barrier
    - **Typical Scenario:**
      1. Visitor places both hands on the barrier.

    2. Visitor applies horizontal force.
    3. Barrier starts to move.
    4. Visitor walks through.
  * **As & Es:** Push but don't walk through, push and walk through only part way, push and find it locked, hit the barrier in anger, curse the barrier in anger.

- **Methodology:**
  - **Brainstorming:** Find as many alternatives and exceptions as possible.
  - **Feedback Loop:** Expect to iterate between defining use cases, scenarios, and the domain model.

8 Non-Functional Requirements (NFRs)

TLDR NFRs are attributes and characteristics of a system, not specific functions. They describe *how* the system should perform its functions, acting as constraints. Often called "quality attributes" or "the -ilities," they differentiate products with similar functionality (e.g., performance, reliability, usability, security). NFRs can conflict and often require prioritization to make trade-off decisions.

- **What are NFRs?**
  - **Attributes and characteristics** of a system, not specific functions.
  - Describe **how** the system should perform its functions, not *what* it should do, acting as constraints on the way the software operates.
  - Often called "quality attributes" or "the -ilities".
  - Differentiate products with similar functionality. (e.g. performance, reliability, usability, security).
- **Key Concepts**
  - **Functional vs. Non-Functional:**
    * Functional requirements are like verbs (actions the system performs).
    * NFRs are like adjectives or adverbs (qualities or characteristics).
  - **Customer Perspective:** Customers often focus on functional requirements, but NFRs heavily impact user experience.
  - **Motherhood Requirements:** Terms like "reliable" and "user-friendly" are expected, but the **degree** and **relative importance** vary.
  - **Fit for Use:** Quality means how well software meets its intended purpose and environment, rather than an absolute measure of "goodness."
- **Types of NFRs**
  - **Product-Oriented**
    * **Performance:** Speed, response time, throughput.
    * **Reliability:** Fault tolerance, mean-time to failure.
    * **Usability:** Ease of learning, user productivity.
    * **Security:** Access control, data protection.
    * **Robustness:** Handling invalid input, graceful degradation.
    * **Scalability:** Handling increasing workloads, users, data.
    * **Efficiency:** Resource utilization, user productivity.
    * **Accuracy/Precision:** Tolerance of errors, precision of results.
    * **Adaptability:** Ease of adding new functionality, reusability.
  - **Process-Oriented**
    * **Process requirements:** Restrictions on development process, resources, personnel.
    * **Design constraints:** Pre-determined design decisions (e.g., platform, components).

        * **Product family requirements:** Integration with other products.

- **Measuring NFRs**
    - **Fitness Criteria:** Quantify the extent to which an NFR must be met. (e.g., "90% of users can complete a task within 4 minutes").
    - **Measurable Metrics:** Define specific ways to measure NFRs (e.g., response time, mean-time to failure, user error rates).
    - **Challenges:** Some NFRs are hard to test before delivery (e.g., long-term reliability).
- **Techniques for Handling Difficult NFRs**
    - **Monte Carlo Techniques:** Estimate unknown quantities using known quantities (e.g., estimating bugs remaining in a program).
    - **Quality-Function Deployment (QFD):** Relate unmeasurable NFRs to measurable functional requirements.
- **Prioritizing NFRs**
    - **Conflicts:** NFRs often conflict (e.g., maintainability vs. robustness, performance vs. security).
    - **Trade-offs:** Prioritization helps make decisions when conflicts arise.
    - **Stakeholder Input:** Different stakeholders may have different priorities.
    - **Quality Grid:** A tool for classifying NFRs by importance (critical, important, as usual, unimportant, ignore).

9 Software Requirements Specification (SRS)

TLDR The SRS defines software requirements in detail for developers, testers, and customers. It typically follows the IEEE standard, which includes: Section 1 (Introduction), Section 2 (Overall Description), and Section 3 (Specific Requirements - the "meat" of the document). Section 3 contains detailed input/output descriptions, use cases, and UML diagrams.

- **SRS Basics:**
    - **Purpose:** Defines software requirements in detail for developers, testers, and customers.
    - **Main Issues Addressed:**
        * **Functionality:** What the software does.
        * **External Interfaces:** Interactions with users, hardware, and other software.
        * **Performance:** Speed, availability, response time.
        * **Quality Attributes (NFRs):** Portability, correctness, maintainability, security.
        * **Design Constraints:** Limitations on acceptable solutions (standards, languages, policies).
    - **Typically NOT Included:** Process requirements, design decisions.
- **IEEE SRS Organization:**
    - **Section 1: Introduction**
        * **1.1 Purpose:** SRS purpose, audience, usage.
        * **1.2 Scope:** Product name, overview (what it will/won't do), application summary, boundaries.
        * **1.3 Acronyms, Abbreviations, Definitions, Notational Conventions:** Domain-level definitions, naming/notational conventions.
        * **1.4 References:** Sources of information (project documentation, interviews, external sources).
        * **1.5 Overview:** Structure of the rest of the SRS.

  - **Section 2: Overall Description** (Background, not specific requirements)
    * **2.1 Product Perspective:** System environment, context diagram, overview of interfaces.
    * **2.2 Product Features:** Overview of main features (list of UC names or brief summaries).
    * **2.3 User Characteristics:** Assumptions about user background/training.
    * **2.4 General Constraints:** Sources of constraints (regulations, hardware, audit/control functions, security, standards, laws).
    * **2.5 Assumptions and Dependencies:** Assumptions about input/environment, potential failure conditions, environmental changes affecting requirements.
  - **Section 3: Specific Requirements** (The "meat" of the SRS)
    * **3.1 External Interfaces:** Detailed input/output descriptions (name, purpose, source/destination, range, units, timing, formats).
    * **3.2 Functional Requirements:** Use case descriptions, sequence diagrams, domain model, functional specifications, state machine model, constraints. It can be organized by user, feature, or stimulus (use case view).
    * **3.3 Performance Requirements:** Concrete terms (number of users/terminals, information handled, transactions processed, workload conditions).
    * **3.4 Design Constraints:**
    * **3.5 Quality Attributes:** Testable non-functional properties (besides performance).

- **Example:** ATM SRS
  - Illustrates the different sections with a real-world example.
  - Highlights definitions, abbreviations, user characteristics, and product perspective.

- **Key Takeaways for the Exam:**
  - Understand the purpose and main components of an SRS.
  - Know the IEEE SRS structure (Sections 1, 2, 3 and their sub-sections).
  - Be able to differentiate between what belongs in each section.
  - Recognize the different ways to organize functional requirements in Section 3.2.
  - Understand the concept of quality attributes (NFRs) and how they are expressed.
  - Be able to identify examples of definitions, abbreviations, assumptions, and constraints from the ATM example.
  - Remember that Section 3 is the most detailed, containing all UML diagrams and input/output behavior specifications.
  - The IEEE standard describes what information should be included in a SRS document, and how that information should be arranged.
  - The main issues that a SRS document should address are functionality, external interfaces, performance, quality attributes, and design constraints.
  - An SRS document should generally not include process requirements or design decisions.
  - The introduction of an SRS document should include its purpose, the scope of the software product, and any definitions, acronyms, abbreviations, or notational conventions used in the document.
  - The overall description section provides a background on the system, which is further detailed in Section 3, and should include product perspective, product functions, user characteristics, general constraints, and assumptions and dependencies.
  - Section 3 should include descriptions of all interfaces to the system, all functions performed by the system, and should be at a level of detail sufficient to enable designers and testers to do their jobs.

10 Temporal Logic

TLDR Temporal logic is a system for describing how system states change over time. It uses temporal connectives like Henceforth ($\Box$), Eventually ($\Diamond$), Next State ($\bigcirc$), Until ($\mathcal{U}$), and Unless ($\mathcal{W}$). When scanning a temporal logic formula from left to right, each temporal operator introduces a new implied bound time variable which is at the same time or later than the previous.

- **Prescriptive vs. Descriptive Specifications:**
  - **Prescriptive:** Model-based, describe system behavior state-by-state based on inputs (e.g., state diagrams).
  - **Descriptive:** Describe system-wide properties over time, using logic (e.g., temporal logic).
- **Predicate Logic Review:**
  - Used for expressing properties about fixed-valued variables.
  - Key components:
    - Typed variables (Booleans, Integers, Sets, Objects)
    - Functions on variables (+, -, *, /, ∪, ∩, ∧, ∨, ¬)
    - Predicates (¡, ¿, ⊂, ∈)
    - Equivalence (=)
    - Propositional connectives (¬, ∧, ∨, →, ↔)
    - Quantifiers (∀, ∃)
- **Time-Dependent Logic:**
  - Variables are treated as time-functions (e.g., `coin(t)` means the value of `coin` at time `t`).
  - Formulas express relationships between variable values at different times.
    - **Example:** $\forall t \in Time : (coin(t) \rightarrow \neg locked(t+1))$ (If a coin is inserted at time t, the system will be unlocked at time t+1).
- **Linear Temporal Logic (LTL):**
  - Focuses on the order of events, abstracting away exact time.
  - **System State:** A snapshot of all variable values at a specific point in time.
  - **Execution:** A sequence of system states ($\sigma = s_0, s_1, s_2, ...$).
- **Temporal Connectives:**
  - **Henceforth ($\Box$):** $\Box f$ means f is true now and in all future states.
  - **Eventually ($\Diamond$):** $\Diamond f$ means f is true now or in some future state.
  - **Next State ($\bigcirc$):** $\bigcirc f$ means f is true in the next state.
  - **Until ($\mathcal{U}$):** $f\ \mathcal{U}\ g$ means g will eventually be true, and f is true until then.
  - **Unless ($\mathcal{W}$):** $f\ \mathcal{W}\ g$ means f is true indefinitely or until g becomes true (but g is not guaranteed to become true).
- **Important Concepts:**
  - $\Box(\Diamond f)$: f happens infinitely often.
  - $\Diamond(\Box f)$: Eventually, f is true forever.
  - When scanning a temporal logic formula from left to right, each temporal operator introduces a new implied bound time variable which is at the same time or later than the previous.
    - **Example:** ($\Box$(coin → $\bigcirc$ ¬Locked)) means for all times i, if coin is true at time i then at a later time i+1 locked will be false.
- **Describing State Machines with LTL:** LTL can describe the behavior of finite state machines by expressing properties about transitions and states over time.

11 User Interface (UI) Design in Requirements

TLDR UIs should be considered and specified during the requirements phase, not left solely to the implementers. Poor UIs can lead to user confusion and errors. UIs must be designed in conjunction with functional requirements to ensure consistency. Usability testing with real users is crucial to validate proposed UIs. Remember Platt's Law: "Know Thy User, for He Is Not Thee."

- **UI as a Requirement:** UIs should be considered and specified during the requirements phase, not left solely to the implementers. Poor UIs can lead to user confusion and errors.
- **UI and Functional Requirements:** UIs must be designed in conjunction with functional requirements to ensure consistency.
- **UI Validation:** Usability testing with real users is crucial to validate proposed UIs.
- **Specifying UIs:** A recommended method is to attach screen diagrams to scenario steps, showing screen appearance, user interactions, and system responses.
- **Why Software Sucks (According to Platt):** The core reason is bad UIs, often designed by programmers for themselves, not for the typical user.
- **Programmers vs. Users:** Programmers value control, while users value ease of use.
- **Platt's Law of UI Design:** "Know Thy User, for He Is Not Thee."
- **Good UI Principles:**
  - Design should make manuals and help systems unnecessary.
  - Guide users through solutions.
  - Default behavior should align with common user needs.
  - Offer guided preference settings.
  - Make operations undoable/redoable when possible.
- **Lostness Formula (Tullis and Albert):** A way to quantify user lostness on a website:
  - $L = \sqrt{((N/S) - 1)^2 + ((R/N) - 1)^2}$
  - R = minimum pages needed for a task.
  - N = number of different pages visited.
  - S = total pages visited (including revisits).
  - L ranges from 0 (not lost) to 1 (totally lost). 0.4 is considered bad.
- **Examples of poor UI choices**
  - Asking the user if they want to save changes when closing a file, instead of asking if they want to lose their changes.
  - Asking the user to confirm sending a file to the recycle bin.
- **General Advice:** KIS (Keep it Simple!)

12 User's Manual (UM) as Requirements Specification (RS)

TLDR A well-written User's Manual (UM), focusing on the user's perspective and functionality, can serve as an effective Requirements Specification (RS) for many types of Computer-Based Systems (CBSs), particularly those with a user interface focus. However, it's not suitable for all systems, especially those with complex, hidden NFRs or those lacking a direct user interaction component.

- **Key Arguments for UM as RS:**
  - **UM as a useful tool:** When done correctly, a UM can serve as a valuable tool for elicitation, analysis, and validation during requirements engineering, and it can even function as the RS itself.

  - **Focus on user's perspective:** A good UM describes the CBS's function from the user's point of view, not the implementer's, aligning with the goals of an RS.
  - **Avoids implementation details:** Like a good RS, a UM should describe *what* the system does, not *how* it does it, leaving design freedom to the implementers.
  - **Historical support:** Several industry figures (Brooks, DeMarco, McConnell) and projects (Lisa, Macintosh) have advocated for or successfully used UMs as RSs.

- **UMs and the 5 Roles of an RS:** The creation and use of a UM can fulfill the five roles of an RS:
  1. **Learning requirements:** Writing a UM helps in understanding the CBS's requirements.
  2. **Reconciling stakeholder differences:** The process helps align different stakeholder perspectives.
  3. **Customer validation:** A UM allows customers to validate that the system meets their needs before implementation.
  4. **Implementation clarity:** A UM clarifies what needs to be built.
  5. **Verification basis:** A UM provides a basis for deriving test cases and verifying the implementation.

- **Elements of a Good UM:**
  1. **Lexicon:** Descriptions of underlying concepts.
  2. **Use Cases:** Graduated examples showing user problems, interactions, and system responses.
  3. **Command Summary:** A systematic overview of all commands.
  4. **Domain Model:** Organization around abstractions in the problem domain, explaining objects, their actions, and what is done to them.

- **Fairley's Preliminary UM Outline:**
  1. **Introduction:** Overview, terminology, formats, and manual outline.
  2. **Getting Started:** Sign-on, help mode, sample run.
  3. **Modes of Operation:** Commands/Dialogues/Reports (essentially scenarios).
  4. **Advanced Features:**
  5. **Command Syntax and System Options:**

- **When UMs as RSs Work (and When They Don't):**
  - **Suitable CBSs:**
    * Have at least one type of user.
    * Provide functionality through a user interface that's clear to the user.
    * Have well-understood, easily described Non-Functional Requirements (NFRs).
  - **Unsuitable CBSs:**
    * Autonomous systems with no human users.
    * CBSs where complex algorithms are the main focus, not the UI.
    * CBSs with complex NFRs (security, reliability) not directly visible to the user.

- **Help Systems as RSs:**
  - Modern alternative to UMs.
  - Provide a good set of scenarios.
  - Can be used as the RS.
  - Scenarios often focus on specific user tasks, making them potentially better than traditional UMs for requirements specification.