Requirements specification:

- A structured document that sets out the services the system is expected to provide.
- Should be precise so that it can act as a contract between the system procurer and software developer and thus needs to be understandable by procurers and developers.
- Describes what the system will do but not how it will do it (objectives but not how objectives will be achieved).

Design specification:

- An abstract description of the software that serves as a basis for (or describes) its detailed design and implementation.
- Describes how the requirements will be achieved.
- Primary readers will be software designers and implementers rather than users or management.
- The goals and constraints specified in requirements document should be traceable to the design specification (and from there to the code).
Contents of requirements document:

- **Introduction:** Describes the need for the system and places it in context, briefly describing its functions and presenting a rationale for the software. Describes how the system fits into the overall business or strategic objectives of the organization commissioning the software.

- **System Model:** Sets out the system model showing the relationships between the system components and the system and its environment. An abstract data model should also be described if appropriate to the type of system.

- **System Evolution:** The fundamental assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs, etc.

- **Goals** (sometimes called **Functional Requirements**): The services provided for the user. This includes timing and accuracy requirements.

- **Constraints:** Constraints on how the goals can be achieved (restrictions on the behavior of the software and the freedom of the designer).

  These are restrictions on allowable designs or the ways the goals can be achieved, e.g., safety constraints, hardware, programming languages, and standards that must be followed. They also include quality requirements, such as maintainability, availability, etc.

- **Priorities:** This section basically specifies how tradeoffs will be made if all of the goals and constraints cannot be completely achieved in the design—some may be more important than others. Guides tradeoffs among design decisions.

- **Interfaces to the environment:** Input or output interfaces and relevant assumptions about environmental components with which the software will be interacting.

- **Glossary:** Definitions of technical terms used in the document.

- **Indexes:** Various types of indexes may be provided.
Attributes of a good requirements document:

- Readable and understandable by customers, users, and designers.
- Specifies only external system behavior (black box).
- Structured to be easy to change.
- Specifies both goals and constraints.
- Able to serve as a reference tool for system maintainers.
- Contains only testable requirements.

An untestable requirement: The system should be easy to use by experienced controllers and should be organized in such a way that user errors are minimized.

A testable requirement: Experienced controllers should be able to use all of the system functions after a total of two hours training. After this training, the average number of errors made by experienced users should not exceed two per day.

- Consistent, complete, unambiguous, and realistic.
- Specifies acceptable responses to undesired events.
- Specifies incremental subsets if desired or minimum and maximum functionality.
- Specifies the changes anticipated in the future (in the environment or in the software).
Ensuring a Successful Product

Right Product

- Producibility Constraints
- Production Requirements
- Production
- In-service Experience
- Accidents and Incidents
- Public Perceptions
- Customer Requirements
- Regulatory Requirements
- Market Driven Requirements
- Boeing Requirements
- Lessons Learned
- Issue Resolution
- Technology Changes
- Airline Industry Trends
- Infrastructure Requirements

- Appropriate and Validated Requirements

Product Right

- Successful Product
- Allocation
- Certification
- Verification Testing
- Requirements Compliance

- Preliminary Physical and Functional Def.
- FHA
- Fault Trees
- Analyze and Validate
- Preliminary FMEA
- Detailed Design
- Analyses
- Safety
- Availability
- Reliability
- Supportability
- Maintainability
Types of Specifications

• Informal
  − Free form, natural language
  − Ambiguity and lack of organization can lead to incompleteness, inconsistency, and misunderstandings

• Formatted
  − Standardized syntax (e.g., UML)
  − Basic consistency and completeness checks
  − Imprecise semantics implies other sources of error may still be present.
## Intent Specifications

<table>
<thead>
<tr>
<th>Intent</th>
<th>Environment</th>
<th>Operator</th>
<th>System</th>
<th>Verification Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Purpose</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Design Principles</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blackbox Behavior</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design Representation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Physical Representation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Each level supports a different type of reasoning about system.
- Mappings between levels provide relational info necessary to reason across hierarchical levels.
Types of Specifications (2)

- Formal
  - Syntax and semantics rigorously defined.
  - Precise form, perhaps mathematical.
  - Eliminate imprecision and ambiguity.
  - Provide basis for mathematically verifying equivalence between specification and implementation.
  - May be hard to read without training.
  - Semantic distance too great?
A program is a mathematical object
A programming language is a mathematical language.
Therefore, we can prove properties about the program.
   e.g. does it do what it is supposed to do
does it not do anything harmful
Building a model like engineers do, but need discrete rather than continuous mathematics.
Formal Specifications

Goal: Describe external behavior without describing or constraining internal design (implementation).

Formal method has 2 parts:

1. Logical theory: means by which reason about specs, properties, and programs.
   - First-order predicate calculus (quantification over variables)
   - Second-order predicate calculus (quantification over relations)
   - Temporal logic

2. Structuring theory: defines elements being reasoned about
Structuring Theory

1. Descriptive Specifications: State desired properties in a purely declarative way.
   - Input-output assertions
   - Algebraic specifications (set of axioms)

2. Operational Specification: Describe desired behavior by providing a model of system.
   - Abstract Model (in terms of previously defined mathematical objects, e.g., sets and sequences operations, e.g., functions and mappings)
     - State machine (states and transitions between states)
Input-Output Assertions

\[ S \{P\} Q \]

If \( S \) holds before execution of \( S \), then \( Q \) holds afterward.

Examples:

1. \( \text{sum} = 0 \ \{ \text{for } i=1 \ \text{to } n \ \text{do sum:=sum+a(i)} \} \ \text{sum} = \sum_{j=1}^{n} a_j \)

2. \textbf{proc} search(A,n,x) \textbf{int};

\textbf{pre} \quad n \geq 0

\textbf{post} \quad \begin{align*}
  &\left( \text{result} = 0 \land \forall i \in \{1,...,n\} : A[i] \neq x \right) \lor \\
  &\left( \text{result} = i \land 1 \leq i \leq n \land A[i] = x \land \\
  &\forall i \in \{1,...,i-1\} : A[i] \neq x \right)
\end{align*}
Algebraic Specifications

Uses: (1) set of objects
(2) set of operations
(3) axioms specifying behavior of operations

Two parts to a specification:
syntax
axioms

Example: STACK

Intuitive definitions of stack operations:
NEW creates an empty stack
PUSH adds a new item to top of stack
TOP returns a copy of top item
POP removes top item
Empty tests for an empty stack
Algebraic Specifications (2)

SYNTAX:

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>DOMAIN</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>()</td>
<td>Stack</td>
</tr>
<tr>
<td>Push</td>
<td>(Stack, Item)</td>
<td>Stack</td>
</tr>
<tr>
<td>Pop</td>
<td>(Stack)</td>
<td>Stack</td>
</tr>
<tr>
<td>Top</td>
<td>(Stack)</td>
<td>Item</td>
</tr>
<tr>
<td>Empty</td>
<td>(Stack)</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

AXIOMS: (stk is of type Stack, itm is of type Item)

1. Empty (new) = true
2. Empty (push (stk, itm)) = false
3. Pop (new) = error
4. Top (new) = error
5. Pop (Push (stk, itm)) = stk
6. Top (Push (stk, itm)) = itm
Algebraic Specifications (3)

Axioms stated in English:

1. A new stack is empty.
2. A stack is not empty immediately after pushing an item onto it.
3. Attempting to pop a new stack results in an error.
4. There is no top item on a new stack.
5. Pushing an item onto a stack and immediately popping it off leaves the stack unchanged.
6. Pushing an item onto a stack and immediately requesting the top item returns the item just pushed onto the stack.
Algebraic Specifications (4)

If want to add an operator Replace (stk,itm):

\[
\text{Replace (stk,itm)} = \begin{cases} 
\text{error} & \text{if Empty (stk)} \\
\text{else Push (Pop (stk),itm)} & \text{else}
\end{cases}
\]

What does this do (in English)?

Canonical Form: A particular stack written in terms of the operations used to form it.

\[
A = \begin{array}{c}
  \text{d} \\
  \text{c} \\
  \text{b} \\
  \text{a}
\end{array} = (\text{Push (Push (Push (New,a)), b), c), d})
\]

Top (Pop (Pop A)) = b