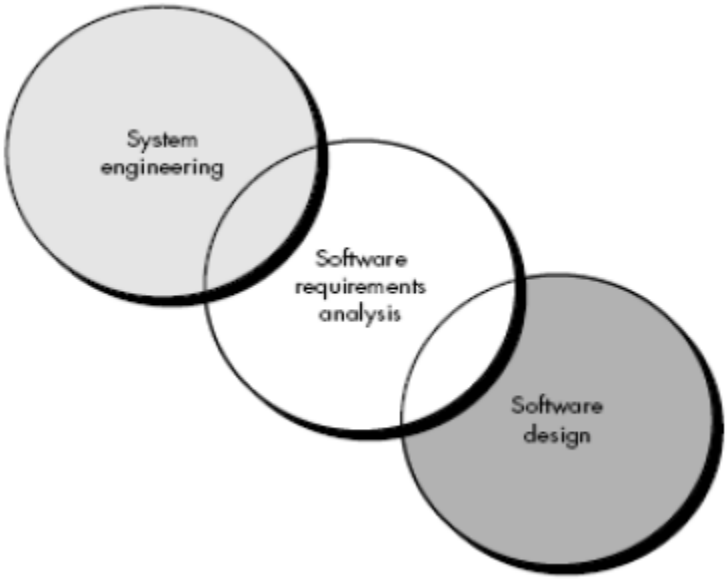
Analysis Concepts & Principles (Chapter 6)

Software requirements engineering is a process of discovery, refinement, modeling, and specification. The system requirements and role allocated to software—initially established by the system engineer—are refined in detail. Models of the required data, information and control flow, and operational behavior are created. Alternative solutions are analyzed and a complete analysis model is created.

REQUIREMENTS ANALYSIS

Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design.



- Requirements engineering activities result in the *specification of software's operational characteristics* (function, data, and behavior), indicate software's interface with other system elements, and establish constraints that software must meet.
- Requirements analysis allows the software engineer/ analyst to *refine the software allocation and build models of the data, functional, and behavioral domains* that will be treated by software.
- Requirements analysis **provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs.**

- Requirements specification *provides the developer and the customer* with the means to *assess quality* once software is built.

Software requirements analysis may be divided into five areas of effort:
(1) Problem recognition,
(2) Evaluation and synthesis,
(3) Modeling,
(4) Specification, and
(5) Review.

First goal is **recognition of the basic problem** elements as perceived by the customer/users.
The analyst studies the ***System Specification*** (if one exists) and the ***Software Project Plan.***
It is important to understand software in a system context and to review the software scope that was used to generate planning estimates.
Next, communication for analysis must be established so that problem recognition is ensured.

Evaluation and solution synthesis is the next major area of effort for analysis.
The analyst must define all externally observable data objects, evaluate the flow and content of information, define and elaborate all software functions.
Upon evaluating current problems and desired information (input and output), the analyst begins to synthesize one or more solutions.
Throughout evaluation and solution synthesis, the analyst's primary focus is on "what," not "how." What data does the system produce and consume, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined and what constraints apply?
During the evaluation and solution synthesis activity, *the analyst* ***creates models*** *of the system* in an effort to better understand data and control flow, functional processing, operational behavior, and information content.
The model serves as a foundation for software design and as the basis for the creation of specifications for the software.

Modeling *helps the analyst to understand the functionality of the system, system flow information and the process flow in the system.*

Specification is a *complete description* of the behavior of a system to be developed. It includes a set of use cases that describe all the

interactions the users will have with the software. *SRS document* is a result of specification.

Review is necessary to find out that whether the *objectives of the project* have been achieved or not. It is used for *detecting and correcting defects* in software artifacts, and preventing their leakage into field operations. Most common *method used is FTR.*

REQUIREMENTS ELICITATION FOR SOFTWARE

- Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation (intelligence collection) process.
- Requirements elicitation is the practice of *obtaining the requirements* of a system from users, customers and other stakeholders.The practice is also sometimes referred to as *requirements gathering.*

- **Initiating the Process**
- **Facilitated Application Specification Techniques**
- **Quality Function Deployment**
- **Use-Cases**

Initiating the Process

The most commonly used requirements elicitation technique is to conduct a meeting or interview. The analyst starts by asking ***context-free questions***. That is, a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first encounter itself.
The **first set of context-free questions focuses on the customer**, the overall goals, and the benefits.
The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution.
The final set of questions focuses on the effectiveness of the meeting.
The Q&A session should be used for the first encounter only and then replaced by a meeting format that combines elements of problem solving, negotiation, and specification.

Facilitated Application Specification Techniques

(FAST), this approach **encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements**

Many different approaches to FAST have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

A **meeting** is conducted **at a neutral site** and attended by both software engineers and customers. **Rules** for preparation and participation are established. **An agenda** is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas. **A "facilitator**" (can be a customer, a developer, or an outsider) controls the meeting.

A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

Initial meetings between the developer and customer occur and basic questions and answers help to establish the scope of the problem and the overall perception of a solution.

➢ Quality Function Deployment

Quality function deployment **(QFD) is a quality management technique that**
translates the needs of the customer into technical requirements for software.

QFD "concentrates on maximizing customer satisfaction from the software engineering process". To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.

QFD identifies three types of requirements:

1. Normal requirements.

The objectives and goals that are stated for a product or system during meetings with the customer.

If these requirements are present, the customer is satisfied.

Examples of normal requirements might be requested types of graphical displays.

2. Expected requirements.

These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them.

Their absence will be a cause for significant dissatisfaction.

Examples of expected requirements are: ease of human/machine interaction, and ease of software installation.

3. Exciting requirements.

These features go beyond the customer's expectations and prove to be very satisfying when present.

For example, word processing software is requested with standard features. The delivered product contains a number of page layout capabilities that are quite pleasing and unexpected.

- **Use-Cases**

  Provide a description of how the system will be used.

  The use-case describes the manner in which an actor interacts with the system.

  To create a use-case, the analyst must first identify the different types of people (or devices) that use the system or product.

  These ***actors*** **a**ctually represent roles that people (or devices) play as the system operates. It is important to note that an actor and a user are not the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities that play just one role.

  Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration.

  Primary actors interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software.

  Secondary actors support the system so that primary actors can do their work.

  Once actors have been identified, use-cases can be developed.

****

1.2 ANALYSIS PRINCIPLES

All analysis methods are related by a set of operational principles:

- The **information domain of a problem must be represented and understood.**
- The **functions that the software is to perform must be defined**.
- The **behavior of the software** (as a consequence of external events) must be represented.
- The **models that depict information function and behavior must be partitioned in a manner that uncovers detail in a layered** (or hierarchical) **fashion.**
- The **analysis process should move from essential information toward implementation detail.**

By applying these principles, the analyst *approaches a problem systematically.*
The information domain is examined so that *function may be understood more completely.*
Models are used so that the characteristics of function and behavior can be communicated in a compact fashion.
Partitioning is applied to reduce complexity. Essential and implementation views of the software are necessary to accommodate the logical constraints imposed by processing requirements and the physical constraints imposed by other system elements.

In addition, Davis suggests a set of guiding principles for requirements engineering:

1. Understand the problem before you begin to create the analysis model.
*T*here is a tendency to rush to a solution, even before the problem is understood. This often leads to elegant software that solves the wrong problem!
2. ***Develop prototypes that enable a user to understand how human/machine interaction will occur***.
Since the perception of the quality of software is often based on the perception of the "friendliness" of the interface, prototyping (and the iteration that results) are highly recommended.

3. ***Record the origin of and the reason for every requirement***.
This is the first step in establishing traceability back to the customer.

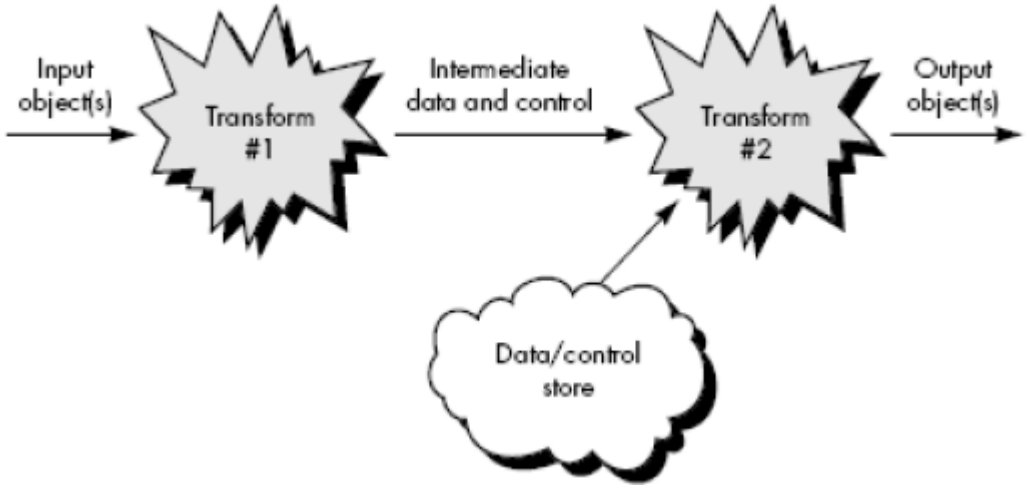
4. ***Use multiple views of requirements***.
Building data, functional, and behavioral models provide the software engineer with three different views. This reduces the likelihood that something will be missed and increases the likelihood that inconsistency will be recognized.
5. ***Rank requirements***.
Tight deadlines may preclude the implementation of every software requirement. If an incremental process model is applied, those requirements to be delivered in the first increment must be identified.
6. ***Work to eliminate ambiguity***
Because most requirements are described in a natural language, the opportunity for ambiguity abounds. The use of formal technical reviews is one way to uncover and eliminate ambiguity.
A software engineer who takes these principles to heart is more likely to develop a software specification that will provide an excellent foundation for design.

1.2.1 INFORMATION DOMAIN

All **software applications can be collectively called** ***data processing.***
Data (numbers, text, images, sounds, video, etc.) and control (events) both reside within the information domain of a problem.
The information domain contains three different views of the data and control as each is processed by a computer program:
(1) **Information content and relationships** (the **data model**),
(2) **Information flow**, and
(3) **Information structure.**
To fully understand the information domain, each of these views should be considered.

Information content **represents the individual data and control objects that constitute some larger collection of information transformed by the software**. For example, the data object, **paycheck** is a composite of a number of important pieces of data: the payee's name, the net amount to be paid, the gross pay, deductions, and so forth. Therefore, the content of **paycheck** is defined by the attributes that are needed to create it.
First analysis principle requires an ***examination of the information domain and the creation of data model.***
During the analysis of the information domain, relationships should be defined.

Information flow *represents the manner in which data and control change as each move through a system.*
The transformations applied to the data are functions or sub functions that a program must perform. Data and control that move between two transformations (functions) define the interface for each function.



Information structure represents the internal organization of various data and control items. Data Structure refers to the design and implementation of information structure within the software.

1.2.2 MODELING

We create functional models to gain a better understanding of the actual entity to be built. When the entity to be built is software, our model must take a different form.

It must be capable of representing the information that software transforms, the functions (and sub functions) that enable the transformation to occur, and the behavior of the system as the transformation is taking place.

The second and third operational analysis principles require that we build models of function and behavior.

i.) **Functional models** :

Software transforms information, and in order to accomplish this, it must perform at least three generic functions: input, processing, and output.

When functional models of an application are created, the software engineer focuses on problem specific functions. The functional model begins with a single context level model (i.e., the name of the software to be built).

Over a series of iterations, more and more functional detail is provided, until a thorough delineation of all system functionality is represented.

ii) **Behavioral models:**

Most software responds to events from the outside world. This stimulus/response characteristic forms the basis of the behavioral model. A computer program always exists in some state—an externally observable mode of behavior (e.g., waiting, computing, printing, polling) that is changed only when some event occurs. For example, software will remain in the wait state until

(1) **An internal clock indicates that some time interval has passed,**

(2) **An external event (e.g., a mouse movement) causes an interrupt**, or

(3) **An external system signals the software to act in some manner**.

A behavioral model creates a representation of the states of the software and the events that cause software to change state.

Models created during requirements analysis serve a number of important roles:

- The model aids the analyst in understanding the information, function, and behavior of a system, thereby making the requirements analysis task easier and more systematic.
- The model becomes the focal point for review and, therefore, the key to a determination of completeness, consistency, and accuracy of the specifications.
- The model becomes the foundation for design, providing the designer with an essential representation of software that can be "mapped" into an implementation context.

Although the modeling method that is used is often a matter of personal (or organizational) preference, the modeling activity is fundamental to good analysis work.

1.2.3 PARTITIONING

Problems are often too large and complex to be understood as a whole.
For this reason, we tend to partition (divide) such problems into parts that can be easily understood, so that overall function can be accomplished.
The fourth operational analysis principle suggests that the information, functional, and behavioral domains of software can be partitioned.
In essence, ***partitioning*** **decomposes a problem into its constituent parts**. Conceptually, we establish a hierarchical representation of function or information and then partition the uppermost element by
(1) Exposing increasing detail by moving vertically in the hierarchy or
(2) Functionally decomposing the problem by moving horizontally in the hierarchy.

Considering the Example of "safe home system" it can be further partitioned as horizontal and vertical.

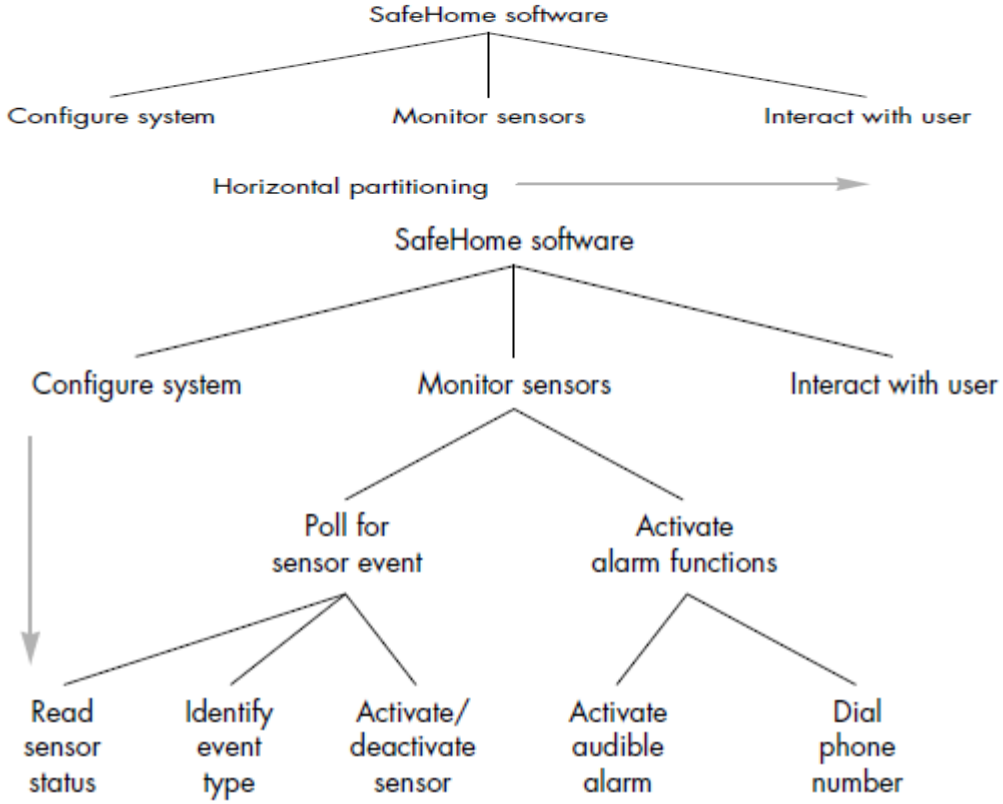


Fig: vertical partitioning

1.2.4 ESSENTIAL AND IMPLEMENTAION VIEW

An ***essential view*** of software requirements presents the functions to be accomplished and information to be processed without regard to implementation details.
The ***implementation view*** of software requirements presents the real world manifestation of processing functions and information structures. In some cases, a physical representation is developed as the first step in software design. However, most computer-based systems are specified in a manner that dictates accommodation of certain implementation details.

Software requirements engineering should focus on what the software is to accomplish, rather than on how processing will be implemented.
However, the implementation view should not necessarily be interpreted as a representation of how. Rather, an implementation model represents the current mode of operation; that is, the existing or proposed allocation for all system elements.
The **essential model (of function or data) is generic in the sense that realization of function is not explicitly indicated.**

SOFTWARE PROTOTYPING

Requirements elicitation is conducted, analysis principles are applied, and a model of the software to be built, called **a *prototype,*** is constructed for customer and developer assessment.
Finally, some circumstances require the construction of a prototype at the beginning of analysis, since the model is the only means through which requirements can be effectively derived. The model then evolves into production software.

Selecting the Prototyping Approach

The prototyping paradigm can be either **close-ended** or **open-ended**.

- **The close-ended approach** is often called ***throwaway prototyping***. Using this approach, a prototype serves solely as a rough demonstration of requirements. It is then discarded, and the software is engineered using a different paradigm.
- **An open-ended approach, called *evolutionary prototyping*,** uses the prototype as the first part of an analysis activity that will be continued into design and

construction. The prototype of the software is the first evolution of the finished system.

Before a close-ended or open-ended approach can be chosen, it is necessary to determine whether the system to be built is suitable to prototyping.

A number of **prototyping candidacy factors** can be defined: **application area, application complexity, customer characteristics, and project characteristics.**

Prototyping Methods and Tools

For software prototyping to be effective, a prototype must be developed rapidly so that the customer may assess results and recommend changes.

Three generic classes of methods and tools are available:

1. Fourth generation techniques.

Fourth generation techniques (4GT) encompass a broad array of database query and reporting languages, program and application generators, and other very high-level nonprocedural languages. Because 4GT enable the software engineer to generate executable code quickly, they are ideal for rapid prototyping.

2. Reusable software components.

Another approach to rapid prototyping is to assemble, rather than build, the prototype by using a set of existing software components.

Melding prototyping and program component reuse will work only if a library system is developed so that components that do exist can be cataloged and then retrieved.

It should be noted that an existing software product can be used as a prototype for a new, improved" competitive product.

3. Formal specification and prototyping environments.

Over the past two decades, a number of formal specification languages and tools have been developed as a replacement for natural language specification techniques.

Today, developers of these formal languages are in the process of developing interactive environments that

(1) Enable an analyst to interactively create language-based specifications of a system or software,

(2) Invoke automated tools that translate the language-based specifications into executable code, and

(3) Enable the customer to use the prototype executable code to refine formal requirements.

****

SPECIFICATION

- *Developed as a consequence of analysis.*

Mode of specification has much to do **with the quality** of solution.

Software engineers who have been forced to work with incomplete, inconsistent, or misleading specifications have experienced the frustration and confusion that invariably results.

The quality, timeliness, and completeness of the software suffer as a consequence.

Specification Principles

Specification, regardless of the mode through which we accomplish it, may be **viewed as a representation process**.

Requirements are represented in a manner that ultimately leads to successful software implementation.

A **number of specification principles**, can be **proposed:**

1. Separate functionality from implementation.

2. Develop a model of the desired behavior of a system that encompasses data and the functional responses of a system to various stimuli from the environment.

3. Establish the context in which software operates by specifying the manner in which other system components interact with software.

4. Define the environment in which the system operates and indicate how **"a highly intertwined collection of agents react to stimuli in the environment (changes to objects) produced by those agents"**

5. Create a cognitive model rather than a design or implementation model. The cognitive model describes a system as perceived by its user community.

6. Recognize that "the specifications must be tolerant of incompleteness and augmentable." A specification is always a model—an abstraction—of some real (or envisioned) situation that is normally quite complex. Hence, it will be incomplete and will exist at many levels of detail.

7. Establish the content and structure of a specification in a way that will enable it to be amenable to change.

Representation

Requirements are committed to paper or an electronic presentation medium a simple set of guidelines is well worth following:

Representation format and content should be relevant to the problem.

A general outline for the contents of a *Software Requirements Specification* can be developed. However, the representation forms contained within the specification are likely to vary with the application area. For example, a specification for a manufacturing automation system might use different symbology, diagrams and language than the specification for a programming language compiler.

Information contained within the specification should be nested.

Representations should reveal layers of information so that a reader can move to the level of detail required. Paragraph and diagram numbering schemes should indicate the level

of detail that is being presented. It is sometimes worthwhile to present the same information at different levels of abstraction to aid in understanding.

Diagrams and other notational forms should be restricted in number and consistent in use.

Confusing or inconsistent notation, whether graphical or symbolic, degrades understanding and fosters errors.

Representations should be revisable.

The content of a specification will change. Ideally, CASE tools should be available to update all representations that are affected by each change.

****

➢ Software Requirements Specification (SRS)

The *Software Requirements Specification* is **produced at the culmination of the analysis task.**

The function and performance allocated to software as part of system engineering *are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.*

The ***Introduction*** of the software requirements specification **states the goals and objectives of the software**, describing it in the context of the computer-based system.

The *Information Description* provides **a detailed description of the problem** that the software must solve. Information content, flow, and structure are documented. Hardware, software, and human interfaces are described for external system elements and internal software functions.

A **description of each function required to solve the problem is presented in the *Functional Description.*** A processing narrative is provided for each function, design constraints are stated and justified, performance characteristics are stated, and one or more diagrams are included to graphically represent the overall structure of the software and interplay among software functions and other system elements.

The *Behavioral Description* *section of the specification examines the operation of the software as a consequence of external events and internally generated control characteristics.*

Validation Criteria is probably the most important and, ironically, the most often neglected section of the *Software Requirements Specification.* How do we recognize a successful implementation? What classes of tests must be conducted to validate function, performance, and constraints?
Finally, the specification includes a ***Bibliography and Appendix***.
1. The **bibliography** contains references to all documents that relate to the software. These include other software engineering documentation, technical references, vendor literature, and standards.
2. The **appendix** contains information that supplements the specifications. Tabular data, detailed description of algorithms, charts, graphs, and other material are presented as appendixes.

In many cases the *Software Requirements Specification* may be accompanied by an executable prototype, a paper prototype or a *Preliminary User's Manual.*

➢ SPECIFICATION REVIEW

A review of the *Software Requirements Specification* (and/or prototype) is conducted by both the software developer and the customer.
Because the specification forms the **foundation of the development phase**, extreme care should be taken in conducting the review.
The review is **first conducted at a macroscopic level**; that is, reviewers attempt to ensure that the specification is complete, consistent, and accurate when the overall information, functional, and behavioral domains are considered. However, to fully explore each of these domains, the review becomes more detailed, examining not only broad descriptions but the way in which requirements are worded.
Once the review is complete, the *Software Requirements Specification* is **"signed off" by both the customer and the developer.**
The specification **becomes a "contract"** for software development.
Requests for changes in requirements after the specification is finalized will not be eliminated.
But the customer should note that each after the fact change is an extension of software scope and therefore can increase cost and/or protract the schedule.