Software Engineering

Third class

Lecture 4

Analysis Concepts and Principles

Software Requirements Engineering

Software requirements engineering is a process of discovery, refinement, modeling, and specification.

Both the software engineer and customer take an active role in software requirements engineering—a set of activities that is often referred to as *analysis.* The customer attempts to reformulate a sometimes nebulous system-level description of data, function, and behavior into concrete detail. The developer acts as interrogator, consultant, problem solver, and negotiator.

1.1 Requirements Analysis

Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design (Figure 1).

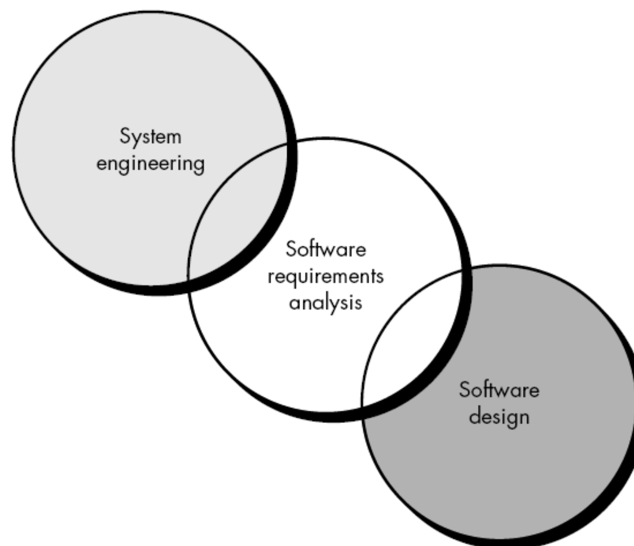


Figure 1 Analysis as a bridge between system engineering and software design

Requirements engineering activities result in:

1. The specification of software's operational characteristics (function, data, and behavior).
2. Indicate software's interface with other system elements

3. 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. Finally, the 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
5. Review.

1.2 Requirements Elicitation for Software

Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process. A customer has a problem that may be amenable to a computer-based solution. A developer responds to the customer's request for help. Communication has begun.

1.2.1 Initiating the Process

The most commonly used requirements elicitation technique is to conduct a meeting or interview. During the meeting Gause and Weinberg suggest that the analyst start 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. Gause and Weinberg call these *meta*-questions.

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.

1.2.2 Facilitated Application Specification Techniques

A number of independent investigators have developed a team-oriented approach to requirements gathering that is applied during early stages of analysis and specification. Called *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

1.2.3 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:

- **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 and specific system functions.
- **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, overall operational correctness and reliability, and ease of software installation.
- **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.

In meetings with the customer, *function deployment* is used to determine the value of each function that is required for the system. *Information deployment* identifies both the data objects and events that the system must consume and produce. These are tied to the functions. Finally, *task deployment* examines the behavior of the system or product within the context of its environment. *Value analysis* is conducted to determine the relative priority of requirements determined during each of the three deployments.

QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity. These data are then translated into a table of requirements—called the *customer voice table*—that is reviewed with the customer. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements

1.2.4 Use-Cases

As requirements are gathered as part of informal meetings, FAST, or QFD, the software engineer (analyst) can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called *use-case*, provide a description of how the system will be used.

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* actually represent roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself.

As an example, consider a microprocessor-based home security system called *SafeHome* will be built to protect against and/or recognize a variety of undesirable "situations" such as illegal entry, fire, flooding, and others. This product (SafeHome) will use appropriate sensors to detect each situation, can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected. For this product we can define three actors: the homeowner (the user), sensors (devices attached to the system), and the monitoring and response subsystem (the central station that monitors *SafeHome*).

Once actors have been identified, use-cases can be developed. The use-case describes the manner in which an actor interacts with the system.

1.3 Analysis Principles

Over the past two decades, a large number of analysis modeling methods have been developed. Investigators have identified analysis problems and their causes and have developed a variety of modeling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view. However, all analysis methods are related by a set of operational principles:

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

By applying these principles, the analyst approaches a problem systematically.

1.3.1-The Information Domain

All software applications can be collectively called *data processing*. This term contains a key to our understanding of software requirements. Software is built to process data, to transform data from one form to another; that is, to accept input, manipulate it in some way, and produce output.

Software also processes events. An event represents some aspect of system control and is really nothing more than Boolean data—it is either on or off, true or false, there or not there. For example, a pressure sensor detects that pressure exceeds a safe value and sends an alarm signal to monitoring software. The alarm signal is an event that controls the behavior of the system. Therefore, 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.
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.

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.

Similarly, the content of a control object called **system status** might be defined by a string of bits. Each bit represents a separate item of information that indicates whether or not a particular device is on- or off-line.

Data and control objects can be related to other data and control objects.

Information flow represents the manner in which data and control change as each moves through a system. Referring to Figure 2, input objects are transformed to intermediate information (data and/or control), which is further transformed to output.

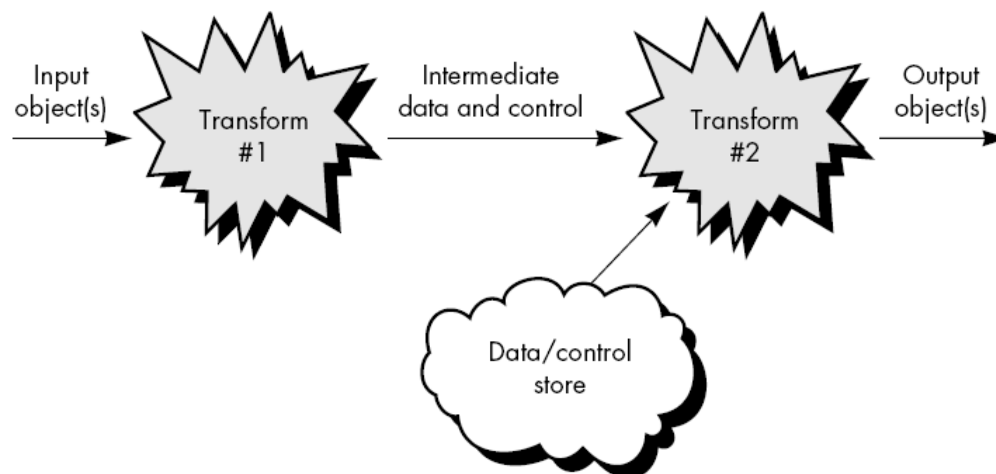


Figure 2: Information flow and transformation

Along this transformation path (or paths), additional information may be introduced from an existing data store (e.g., a disk file or memory buffer). The transformations applied to the data are functions or subfunctions that a program must perform.

Information structure represents the internal organization of various data and control items. Are data or control items to be organized as an *n*-dimensional table or as a hierarchical tree structure? Within the context of the structure, what information is related to other information? Is all information contained within a single structure or

are distinct structures to be used? How does information in one information structure relate to information in another structure? These questions and others are answered by an assessment of information structure.

1.3.2 Modeling

We create functional models to gain a better understanding of the actual entity to be built. When the entity is a physical thing (a building, a plane, a machine), we can build a model that is identical in form and shape but smaller in scale. However, 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 subfunctions) 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.

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.

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) 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 a software to change state.

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

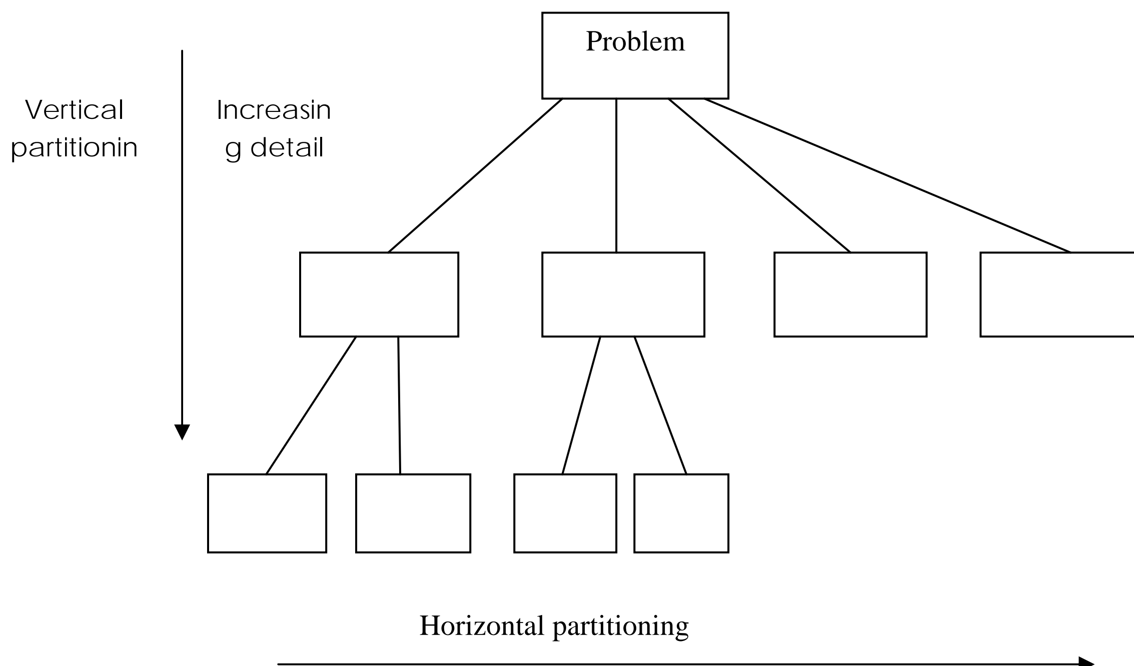
- The model aids the analyst in understanding the information, function, and behavior of a system, thereby making the requirements analysis task easier and more systematic.
- The model becomes the focal point for review and, therefore, the key to a determination of completeness, consistency, and accuracy 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.

1.3.3 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 and establish interfaces between the parts 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
2. Functionally decomposing the problem by moving horizontally in the hierarchy.



1.3.4 Essential and Implementation Views (Logical and Physical Views)

An *essential view* of software requirements presents the functions to be accomplished and information to be processed without regard to implementation details.

For example, the essential view of the *SafeHome* function **read sensor status** does not concern itself with the physical form of the data or the type of sensor that is used. In fact, it could be argued that read status would be a more appropriate name for this function, since it disregards details about the input mechanism altogether. Similarly, an essential data model of the data item **phone number** (implied by the function *dial phone number*) can be represented at this stage without regard to the underlying data structure (if any) used to implement the data item. By focusing attention on the essence of the problem at early stages of requirements engineering, we leave our options open to specify implementation details during later stages of requirements specification and software design.

The *implementation view* of software requirements presents the real world manifestation of processing functions and information structures.

A *SafeHome* input device is a perimeter sensor (not a watch dog, a human guard, or a booby trap). The sensor detects illegal entry by sensing a break in an electronic circuit. The general characteristics of the sensor should be noted as part of a software requirements specification. The analyst must recognize the constraints imposed by predefined system elements (the sensor) and consider the implementation view of function and information when such a view is appropriate.

Note:

Software requirements engineering should focus on what the software is to accomplish, rather than on how processing will be implemented.