Subject Name: Software Engineering

Subject Code: SCS1305

Unit – II

Software Engineering Process

Faculty Name: A.C. SANTHA SHEELA

SOFTWARE ENGINEERING PROCESS

- Functional Requirements
- Non-Functional Requirements
- User Requirements
- System Requirements
- Requirement Engineering Process
- Requirements
- Elicitation
- Validation and management
- Feasibility Studies

- Fundamental of requirement analysis
- Analysis principles Software prototyping
- Prototyping in the Software Process
- Rapid Prototyping Techniques
- User Interface Prototyping

SOFTWARE ENGINEERING PROCESS

Software Document Analysis and Modelling

- Data
- Functional and
- Behavioural Models

Structured Analysis and Data Dictionary.

COURSE OBJECTIVES

-Software process models and compare their applicability.

-Identify the key activities in managing a software project.

Requirements Engineering

- **Requirements engineering (RE)** refers to the
- Process of defining,
- Documenting, and
- Maintaining Requirements
- in the engineering design process.

Requirements Engineering

- Requirement engineering provides the appropriate mechanism to understand
- what the customer desires,
- analyzing the need, and assessing feasibility,
- negotiating a reasonable solution,
- specifying the solution clearly,
- validating the specifications and
- managing the requirements as they are transformed into a working system.

Requirements Engineering

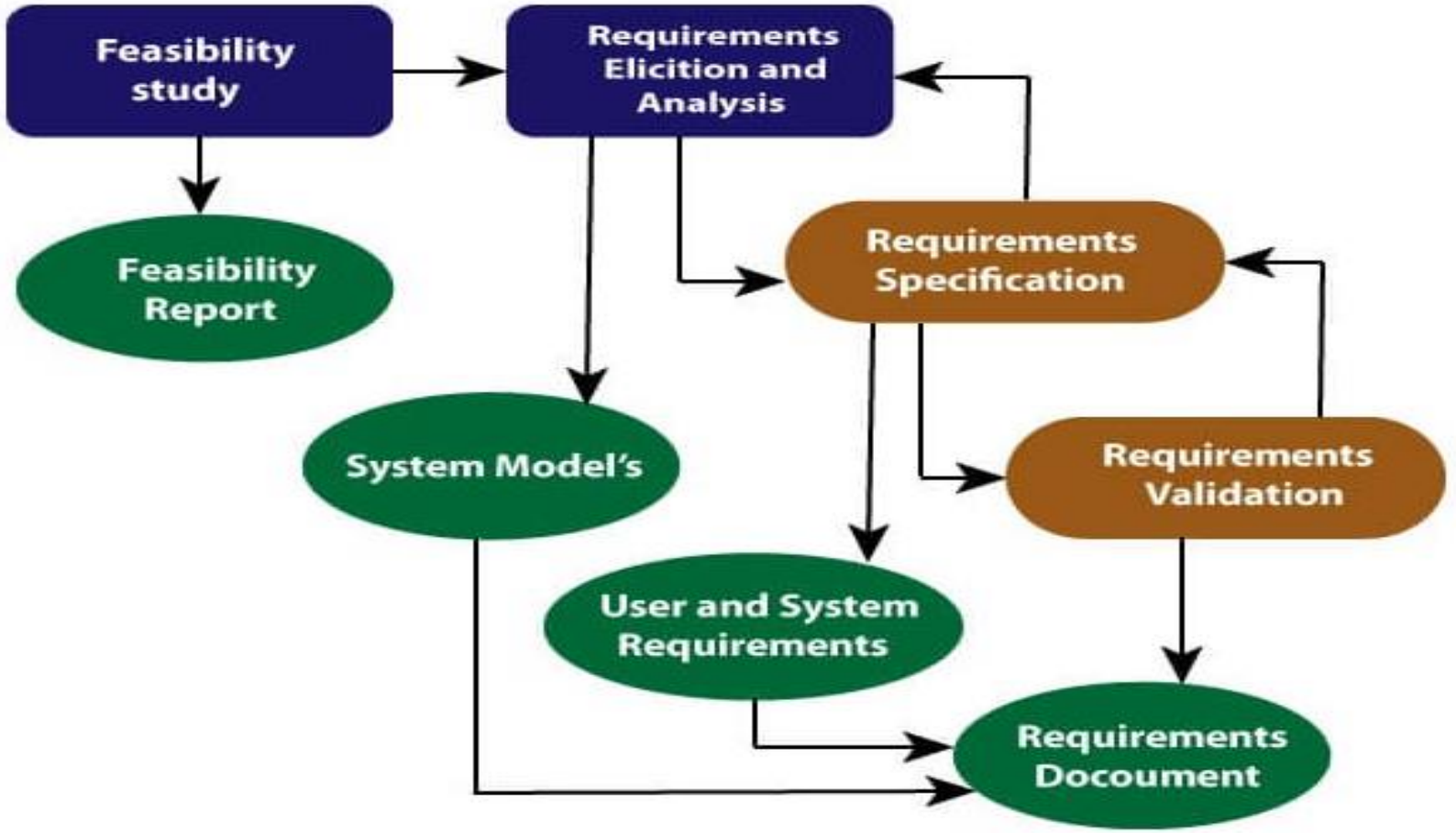
Thus,

- requirement engineering is the
- Disciplined application of **Proven**
- Principles, Methods, Tools, and Notation
- To describe a proposed system's intended behavior and
- its associated constraints.

Requirements Engineering Process

1. Feasibility Study
2. Requirement Elicitation and Analysis
3. Software Requirement Specification
4. Software Requirement Validation
5. Software Requirement Management

Requirements Engineering Process



Requirement Engineering Process

1. Feasibility Study

A feasibility study decides whether or not the proposed system is worthwhile or not.

A short focused study that checks

- ✓ If the system contributes to organizational objectives;
- ✓ If the system can be engineered using current technology and within budget;
- ✓ If the system can be integrated with other systems that are used.

1. Feasibility Study

The objective behind the feasibility study is to create the reasons for developing the software that is acceptable to users, flexible to change and conformable to established standards.

Types of Feasibility:

- Technical Feasibility
- Operational Feasibility
- Economic Feasibility
- Alternatives.

1. Feasibility Study

Technical Feasibility -

Technical feasibility evaluates the current technologies, which are needed to accomplish customer requirements within the time and budget.

Operational Feasibility -

Operational feasibility assesses the range in which the required software performs a series of levels to solve business problems and customer requirements.

1. Feasibility Study

Economic Feasibility -

Economic feasibility decides whether the necessary software can generate financial profits for an organization.

Alternatives.

An evaluation of alternative approaches to the development of the system or product.

Feasibility Report

1. **Executive Summary** - Management Summary and Recommendations A summary of important findings and recommendations for further system development. Should be maximum of one page and self contained, i. e., no references to tables or figures.
2. **Introduction** A brief statement of the problem, the computing environment in which the system is to be implemented and constraints that affect the project. The scope of the problem should be defined.

Feasibility Report

3. Background Discuss what others have done in relation to the problem. Define terms and other information which will be needed as background in order for the reader to understand the report.

4. Alternatives A presentation of alternative system configurations; state the criteria that were used in selecting the final approach.

5. System Description An abbreviated statement of scope of the system. Feasibility of allocated elements

Feasibility Report

6. Cost-Benefit Analysis An economic justification for the system.

7. Evaluation of Technical Risk A presentation of technical feasibility.

8. Legal Ramifications a complex or unwelcome consequence of an action or event.

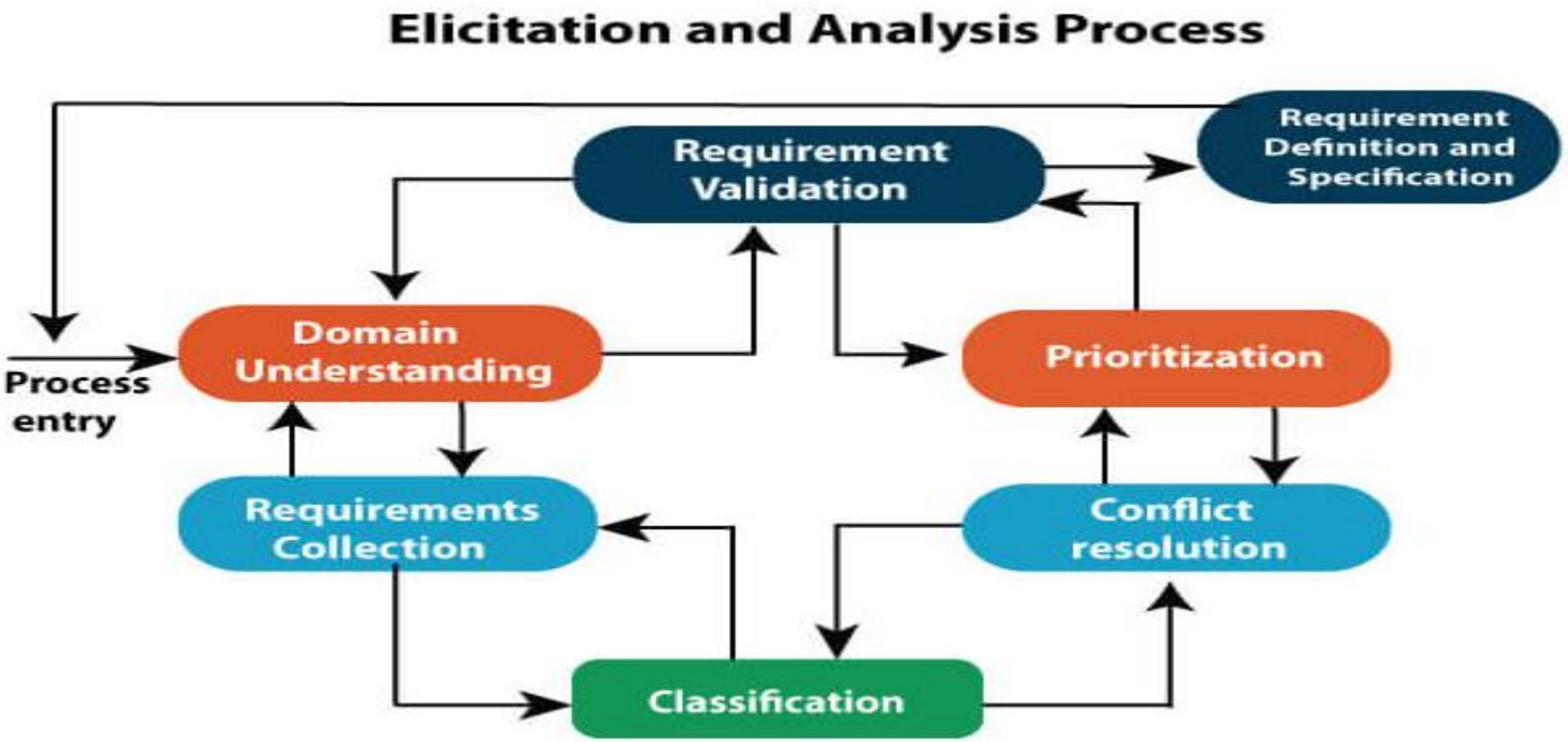
2. **Requirement Elicitation and Analysis**

- This is also known as the **gathering of requirements**. Here, requirements are identified with the help of customers and existing systems processes, if available.
- Analysis of requirements starts with requirement elicitation.
- The requirements are analyzed to identify inconsistencies, defects, omission, etc.
- We describe requirements in terms of relationships and also resolve conflicts if any.

Problems of Elicitation and Analysis

- Getting all, and only, the right people involved.
- Stakeholders often don't know what they want.
- Stakeholders express requirements in their terms.
- Stakeholders may have conflicting requirements.
- Requirement change during the analysis process.
- Organizational and political factors may influence system requirements.

Elicitation and Analysis Process



3. Software Requirement Specification

- Software requirement specification is a kind of document which is created by a software analyst after the requirements collected from the various sources.
- The requirement received by the customer written in ordinary language.
- It is the job of the analyst to write the requirement in technical language so that they can be understood and beneficial by the development team.
- The models used at this stage include ER diagrams, data flow diagrams (DFDs), function decomposition diagrams (FDDs), data dictionaries, etc.

4. Software Requirement Validation

- After requirement specifications developed, the requirements discussed in this document are validated.
- The user might demand illegal, impossible solution or experts may misinterpret the needs.

Requirements can be the check against the following conditions

- If they can practically implement
- If they are correct and as per the functionality and specially of software

- If there are any ambiguities
- If they are full
- If they can describe

Requirements Validation Techniques

Requirements reviews/inspections**:** systematic manual analysis of the requirements.

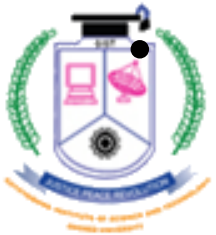
Prototyping**:** Using an executable model of the system to check requirements.

Test-case generation**:** Developing tests for requirements to check testability.

Automated consistency analysis**:** checking for the consistency of structured requirements descriptions.

Software Requirement Management:

- Requirement management is the process of managing changing requirements during the requirements engineering process and system development.
- New requirements emerge during the process as business needs a change, and a better understanding of the system is developed.
- The priority of requirements from different viewpoints changes during development process.
- The business and technical environment of the system changes during the development.



Prerequisite of Software requirements

- Collection of software requirements is the basis of the entire software development project. Hence they should be clear, correct, and well-defined.
- A complete Software Requirement Specifications should be:

Clear	Correct	Consistent
Coherent	Comprehensible	Modifiable
Verifiable	Prioritized	Unambiguous
Traceable	Credible source	

Software Requirements

Software requirements must be categorized into two categories:

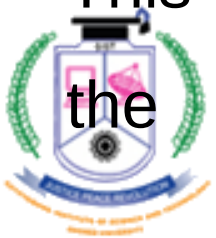
Functional Requirements:

Functional requirements define a function that a system or system element must be qualified to perform and must be documented in different forms.

The functional requirements are describing the behavior of the system as it correlates to the system's functionality.

Non-functional Requirements:

This can be the necessities that specify the criteria that can be used to decide the operation instead of specific behaviors of the system.



Software Requirements

Non-functional requirements are divided into two main categories:

Execution qualities

like security and usability, which are observable at run time.

Evolution qualities

like testability,

maintainability,

extensibility, and

scalability

that embodied in the static structure of the software system.

Functional and Non-Functional Requirements

- The analyst starts requirements gathering and analysis activity by collecting all information from the customer which could be used to develop the requirements of the system.
- He then analyses the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to removing all ambiguities and inconsistencies from the initial customer perception of the problem.

Functional and Non-Functional Requirements

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the likely complexities that might arise while solving the problem?
- What are types of data interchange format for interaction with external software or hardware?

Functional and Non-Functional Requirements

- The requirements are identified and prepared in a textual format called Software Requirement Specification (SRS) document.

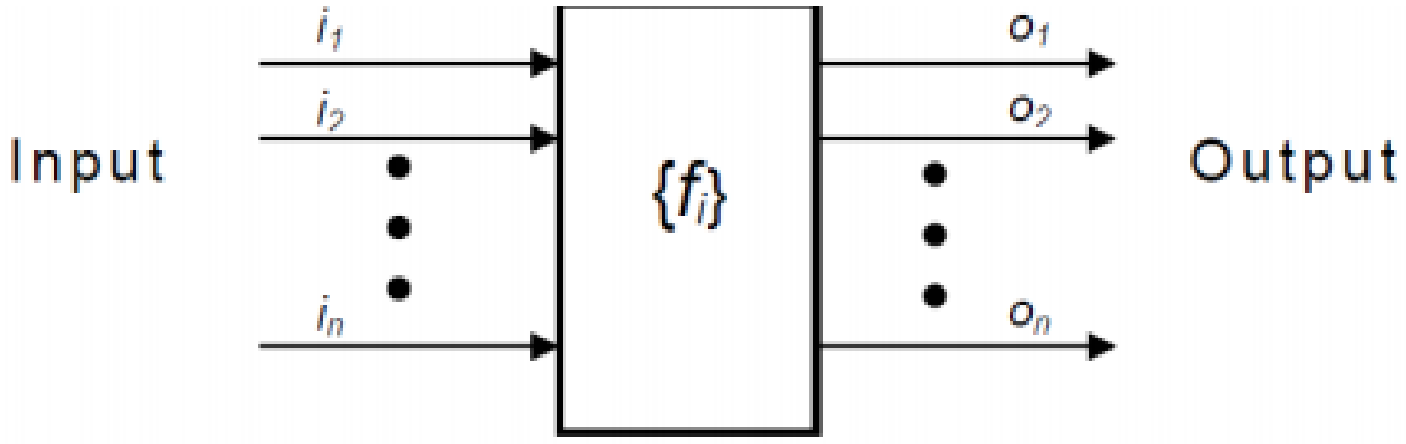
Parts of SRS document are:

- Functional requirements of the system
- Non-functional requirements of the system, and
- Goals of implementation

Functional requirements

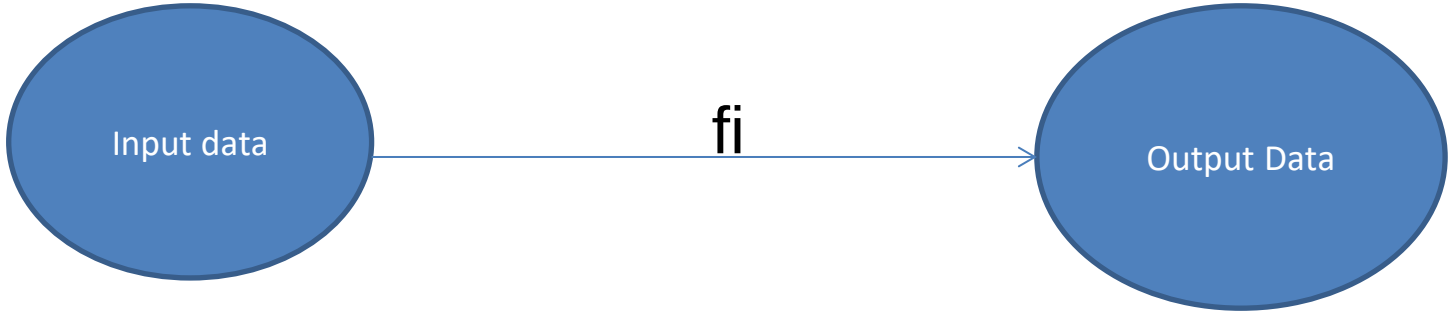
- The functional requirements part discusses the functionalities required from the system.
- The system is considered to perform a set of high- level functions {fi}.
- Each function fi of the system can be considered as a transformation of a set of input data (ii) to the corresponding set of output data (oi).
- The user can get some meaningful piece of work done using a high-level function.
- The functional view of the system is shown in figure.

Functional requirements



Identifying Functional requirements

- Functional requirements need to be identified either from informal problem description document or from a conceptual understanding of the problem.
- Identify different types of users who might use the system and try to identify the requirements from each user perspective.



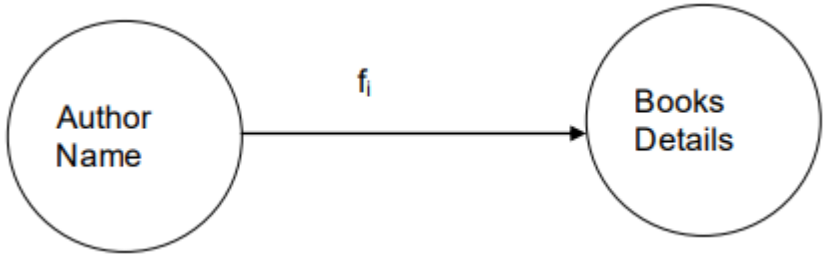
Identifying Functional requirements

Example:- Consider the case of the library system, where –

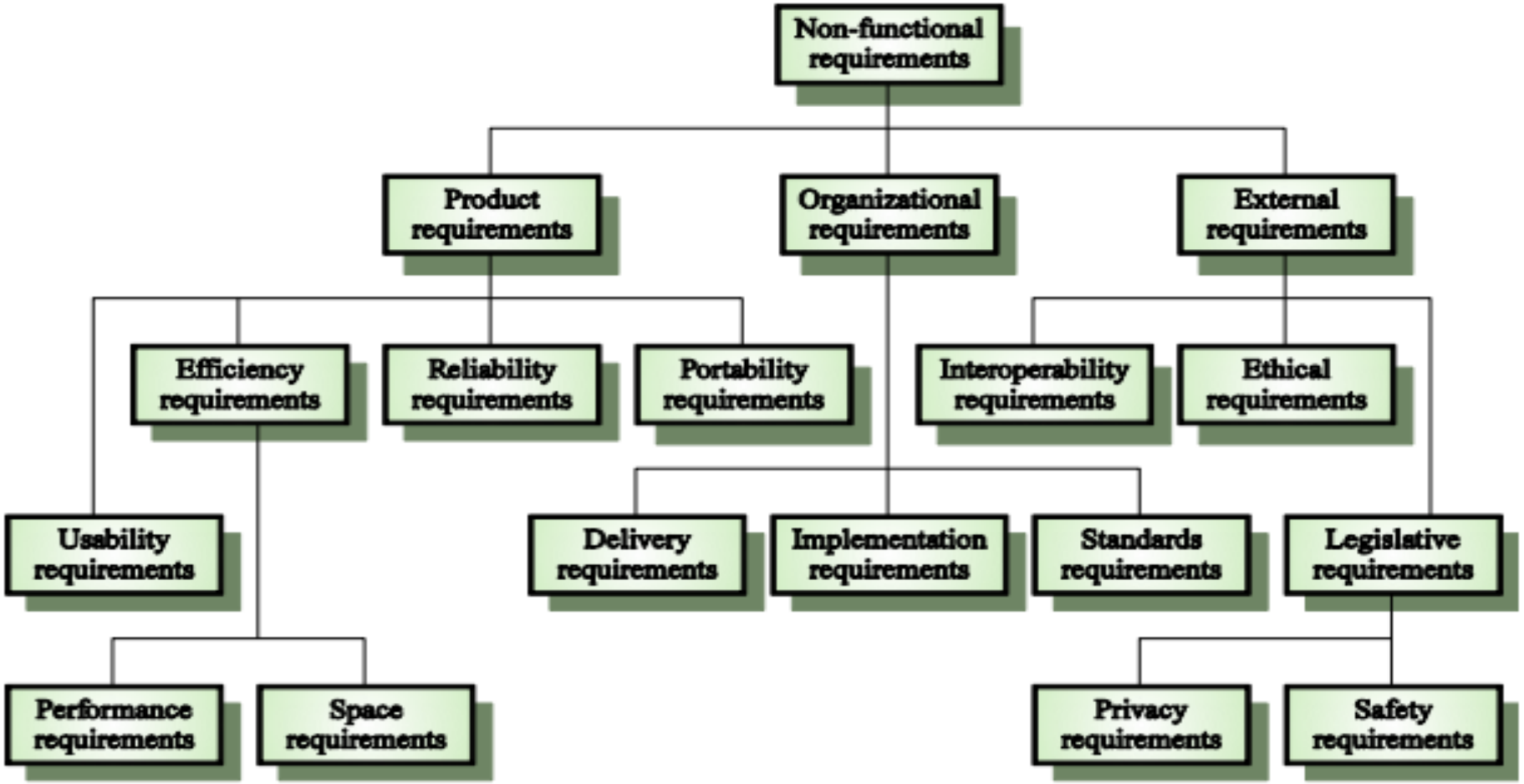
F1: Search Book function

Input: an author‘s name

Output: details of the author‘s books and the location of these books in the library



Non Functional Requirements



Properties of a good SRS document

The important properties of a good SRS document are the following:

- **Concise**. The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.
- **Structured**. It should be well-structured, easy to understand and modify.
- **Black-box view** SRS document should specify the external behavior of the system and not discuss the implementation issues.

Properties of a good SRS document

- **Conceptual integrity**. It should show conceptual integrity so that the reader can easily understand it.
- **Response to undesired events**. It should characterize acceptable responses to undesired events.
- **Verifiable**. All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.

Problems without a SRS document

- The system would not be implemented according to customer needs.
- Software developers would not know whether what they are developing is what exactly required by the customer.
- Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.
- It will be very much difficult for user document writers to write the users‘ manuals properly.

Problems with an unstructured specification

- It would be very much difficult to understand that document.
- It would be very much difficult to modify that document.
- Conceptual integrity in that document would not be shown.
- The SRS document might be unambiguous and inconsistent

User requirements

- Statements of what services the system is expected to provide to system users and the constraints under which it must operate, in a natural language plus diagrams.

System requirements

- More detailed descriptions of the software system‘s functions, services and operational constraints.
- The system requirements document / functional specification defines exactly what is to be implemented.

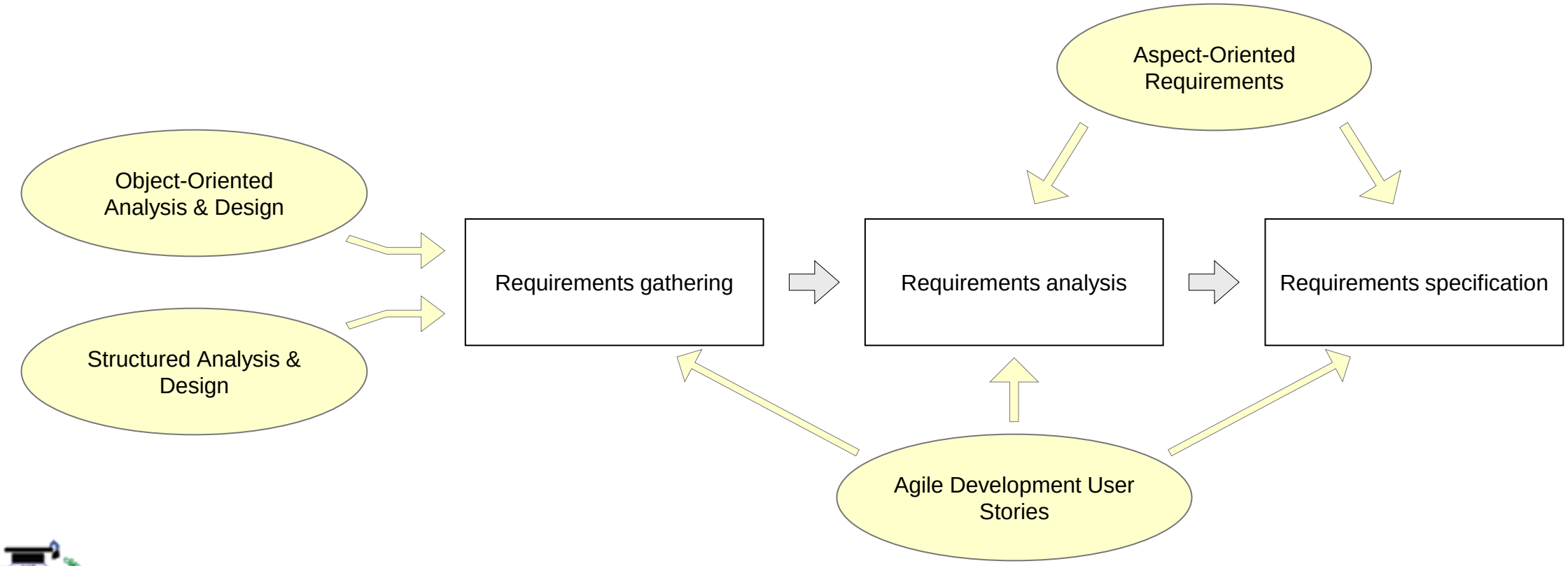
It may be part of a contract between the system buyer and the software developers.

Requirement Engineering Process

Requirements engineering provides the appropriate mechanism for understanding

- what the customer wants, analyzing need,
- assessing feasibility,
- negotiating a reasonable solution,
- specifying the solution unambiguously,
- validating the specification, and
- managing the requirements as they are transformed into an operational system.

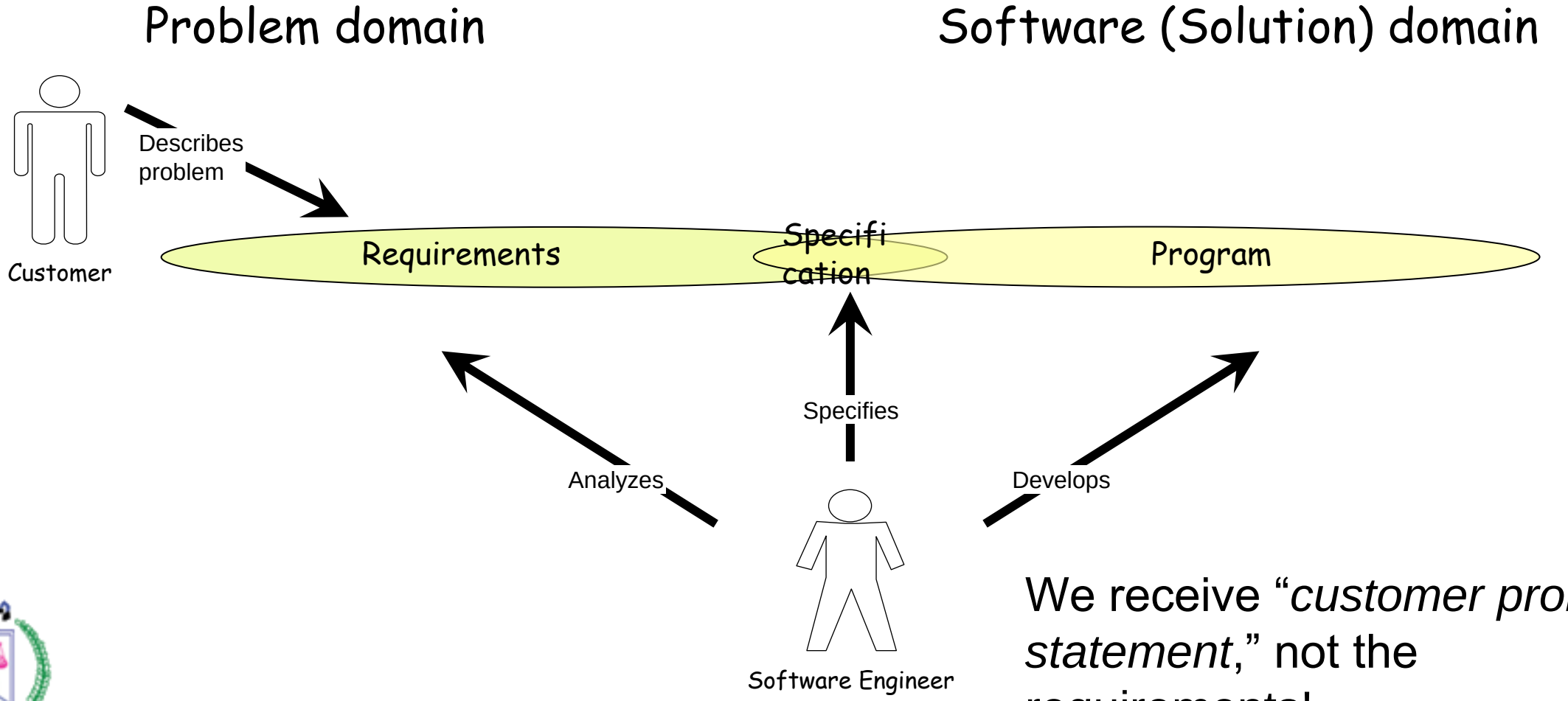
Requirements Process



Requirements Engineering Components

- **Requirements gathering**
  - (a.k.a. “requirements elicitation”) helps the customer to define what is required: what is to be accomplished, how the system will fit into the needs of the business, and how the system will be used on a day-to-day basis
- **Requirements analysis**
  - refining and modifying the gathered requirements
- **Requirements specification**
  - documenting the system requirements in a semiformal or formal manner to ensure clarity, consistency, and completeness

Requirements and Specification



We receive "*customer problem statement*," not the requirements!

Problem Example: Safe Home Access

- **Problem detected**:
  difficult access or unwanted intrusion
  (plus: operating household devices and minimizing living expenses)
- **Analysis of the Causes**:
  - User forgets to lock the door or turn off the devices
  - User loses the physical key
  - Inability to track the history of accesses
  - Inability to remotely operate the lock and devices
  - Intruder gains access
- **System Requirements**: based on the selected causes

Requirements as User Stories

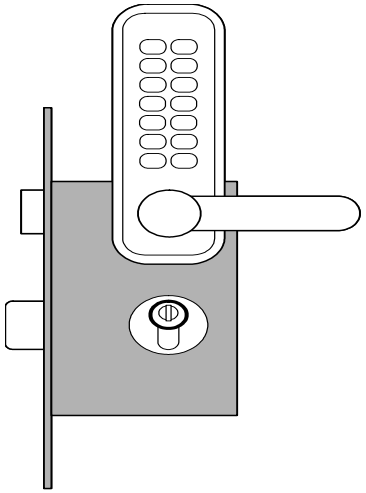
As a tenant, I can unlock the doors to enter my apartment.

user-role (benefactor) | capability | business-value

- Preferred tool in agile methods.
- Focus on the *user benefits*, instead on *system features* as in older, IEEE Standard 830 type “requirements”.
- User stories are *less formal* and emphasize automated acceptance tests.

Example User Story Requirements

Identifier	User Story	Size
REQ-1	As an authorized user, I will be able to keep the doors by default always locked.	**4 points**
REQ-2	As an authorized user, I will be able to unlock the doors using a valid key.	**7 points**
REQ-3	An intruder will not be able to unlock the doors by guessing a valid key; the system will block upon a “dictionary attack.”	**7 points**
REQ-4	The door will be automatically locked after being open for a defined period of time.	**6 pts**
REQ-5	As a user, I will have the door keypad backlit when dark for visibility.	**3 pts**
REQ-6	Anyone will be able to lock the doors on demand.	**2 pts**
REQ-7	As a landlord, I will be able at runtime to manage user authorization status.	**10 pts**
REQ-8	As an authorized user, I will be able to view the history of accesses and investigate “suspicious” accesses.	**6 pts**
REQ-9	As an authorized user, I will be able to configure the preferences for activation of household devices.	**6 pts**



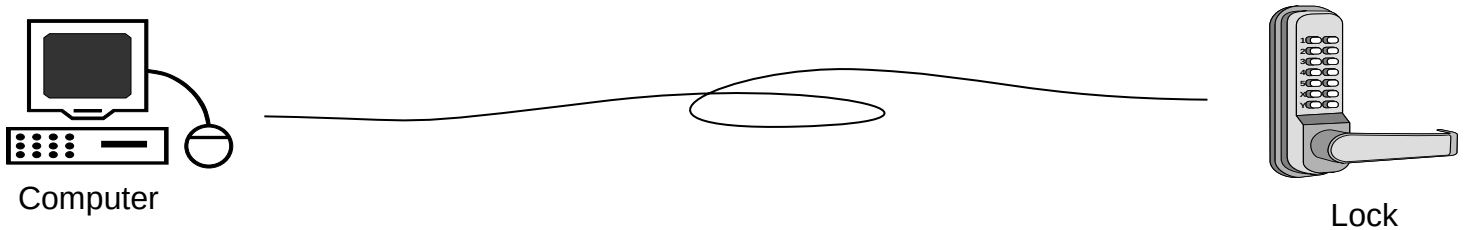
- Note no priorities for user stories
- Story priority is given by its order of appearance on the work backlog (described later)
- Estimated size points (last column) will be described later (also see 1st lecture)

Requirements Analysis

- Requirement REQ-3 states that intruders will not be able to succeed with a "dictionary attack," but many details need to be considered and many parameters determined ("business policies")
  - What distinguishes user's mistakes from "dictionary attacks"
    - The number of allowed failed attempts, relative to a predefined threshold value
      - The threshold shall be small, say three ← business policy!
  - How is the mechanical lock related to the "blocked" state?
    - Can the user use the mechanical key when the system is "blocked"?
- Requirement REQ-5 states that the keypad should be backlit when dark
  - Is it cost-effective to detect darkness vs. keep it always lit?
- Etc.

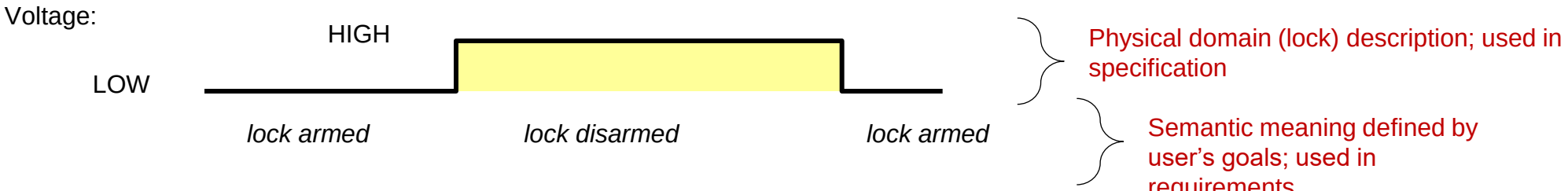
Requirements analysis should not be exhaustive, but should neither be avoided.

Problem Domain:
How Electronic Lock Works



We may need separate descriptions/models of door vs. lock.

Door state is what the user cares about; lock is one way of achieving it.



The behavior of the system-to-be determined not only by user's actions but also by the context ("situation").

E.g., what in case of power failure?
- By default armed
- By default disarmed (e.g., fire exit)

Analyst's Task: Three Descriptions

The *requirement*

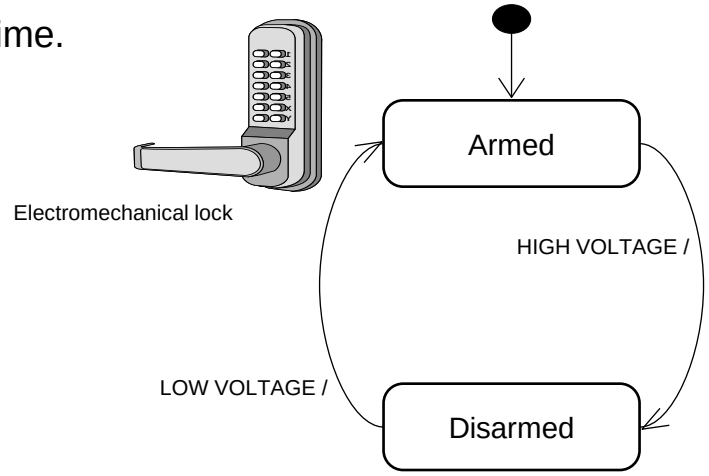
What user wants:

When valid keycode entered + Unlock pressed, open the lock;

Automatically lock after a period of time.

The *problem domain*

How problem domain behaves:



The *specification*

What software-to-be will do (at interface):

If entered number matches one of stored numbers + Button-1 pressed, put HIGH voltage on Output-port-1;

Start a timer countdown;

When the timer expires,
put LOW voltage on Output-port-1.

Concern:

It is not obvious that this is the only or even "correct" solution to the requirement-posed problem.

Problem Frames tell us what each description should contain and how to verify the concern.

Types of Requirements

- Functional Requirements
- Non-functional requirements (or quality requirements)
  - FURPS+
  - Functionality (security), Usability, Reliability, Performance , Supportability
- User interface requirements

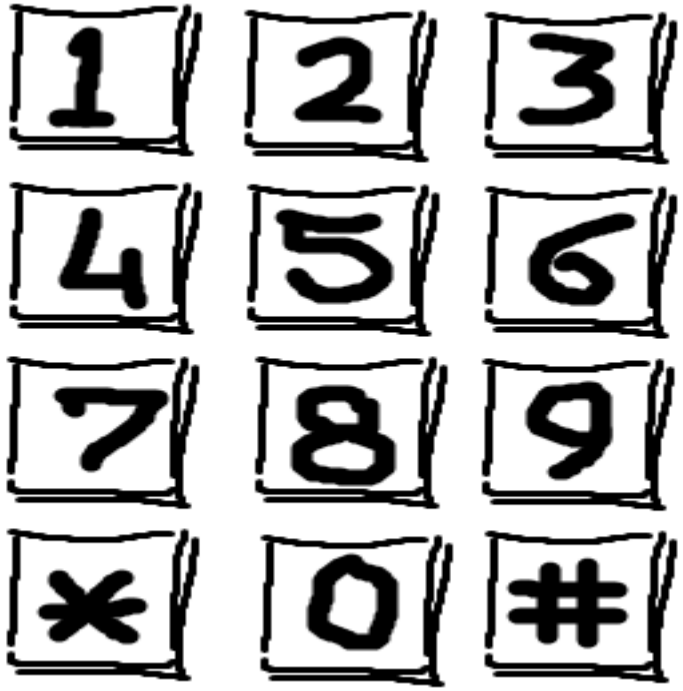
User Interface Requirements

- Do not waste your time and your customer's time by creating elaborate screen shots with many embellishments, coloring, shading, etc., that serves only to distract attention from most important aspects of the interface
- Hand-drawing the proposed interface forces you to ***economize*** and focus on the most important features
- Only when there is a consensus that a good design is reached, invest effort to prototype the interface

https://arstechnica.com/gadgets/2018/01/with-ink-to-code-microsoft-is-turning-back-of-napkin-sketches-into-software/

User interface Requirement: Example: Safe Home Access

Initial design of the door keypad:

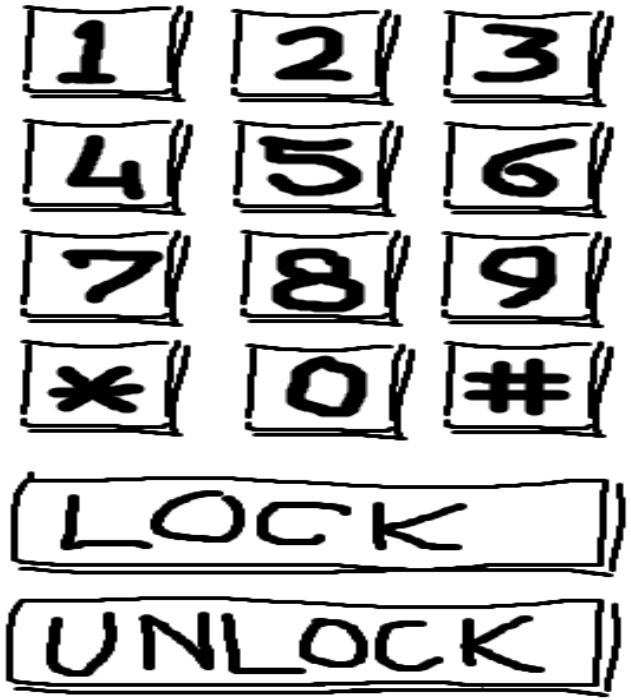


Analysis:

- If “Unlock” button is *not* available, then the system could simply take the first *N* digits and validate as the key.
- If “Unlock” button *is* available, then the system could take the last *N* digits and validate as the key (and ignore any number >*N* of previously entered digits).
- The advantage of the latter approach is that it allows correcting for unintentional mistakes during typing, so the *legitimate* user can have more opportunities to attempt.
- Note that this feature will not help the burglar trying a dictionary attack.

User interface Requirement: Example: Safe Home Access

Redesigned door keypad: includes the “Unlock ”& “Lock” buttons

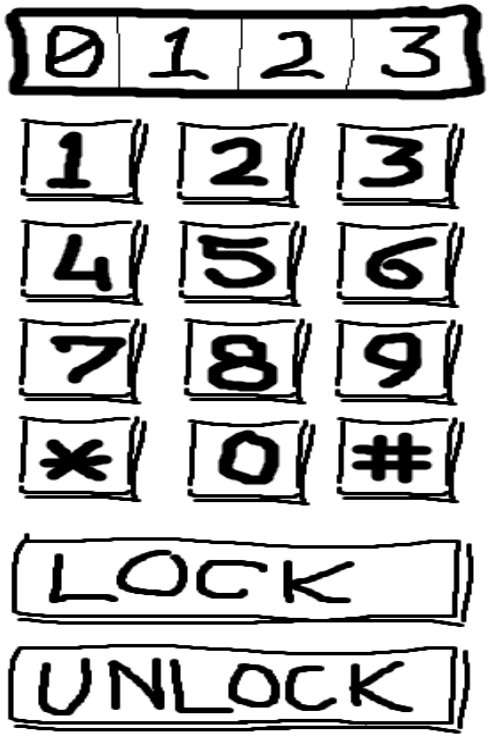


Analysis:

- When a user types in the key and the system reports an invalid key, the user may not know whether the problem is in his fingers or in his memory: “*did I mistype the correct key, or I forgot the key*?”
- To help the user in such a situation, we may propose to include a numerical display that shows the digits as the user types.

Example: Safe Home Access

Re-redesigned door keypad: includes the keycode display



Analysis:

- There are several issues to consider about the display feature:
  - How much this feature would increase the overall cost?
  - Would the display make the door device bulky and inconvenient to install?
  - Would it be significantly more sensitive to rain and other elements?
  - Would the displayed information be helpful to an intruder trying to guess a valid key?

Tools for Requirements Eng.

- **Tools, such as user stories and use cases,used for**:
  - Determining what exactly the user needs (“requirements analysis”)
  - Writing a description of what system will do (“requirements specification”)
- Difficult to use the same tool for different tasks (analysis vs. specification)

Requirement Engineering Process

The Requirement engineering process is accomplished through the execution of seven distinct functions.

- Inception
- Elicitation
- Elaboration
- Negotiation
- Specification
- Validation and management

Inception

- How does a software project get started?
- At project inception,

you establish a basic understanding of the problem,

the people who want a solution,

the nature of the solution that is desired, and

the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

Elicitation

- Ask the customer, the users, and others

✓ what the objectives for the system or product are,

✓ what is to be accomplished,

✓ how the system or product fits into the needs of the business, and

✓ how the system or product is to be used on a day-to-day basis.

Elicitation

 Problems of scope. The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives

 Problems of understanding. The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don‘t have a full understanding of the problem domain, have trouble communicating needs to the system engineer.

 Problems of volatility. The requirements change over time.

Elaboration

- The information obtained from the customer during inception and elicitation is expanded and refined during elaboration.
- This task focuses on developing a refined requirements model that identifies various aspects of

  software function,

  behavior, and

  information.

Negotiation

- It isn‘t unusual for customers and users to ask for more than can be achieved, given limited business resources.
- It‘s also relatively common for different customers or users to propose conflicting requirements, arguing that their version is —essential for our special needs.
- You have to reconcile these conflicts through a process of negotiation.
- Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

Specification

- A specification can be a written document,

  a set of graphical models,

  a formal mathematical model,

  a collection of usage scenarios,

  a prototype, or any combination of these.

- The specification is the final work product produced by the requirement engineer.

Validation

- Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.
- The primary requirements validation mechanism is the formal technical review.

Requirements management

Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds

- Once requirements have been identified, traceability tables are developed among many possible traceability tables are the following:

Requirements management

Features traceability table. Shows how requirements relate to important customer observable system/product features.

Source traceability table - Identifies the source of each requirement.

Dependency traceability table - Indicates how requirements are related to one another.

Subsystem traceability table - Categorizes requirements by the subsystem(s) that they govern.

Interface traceability table - Shows how requirements relate to both internal and external system interfaces.

Traceability Table

Requirement \ Specific aspect of the system or its environment	A01	A02	A03	A04	A05				Aii
R01			✓		✓				
R02	✓		✓						
R03	✓			✓					✓
R04		✓			✓				
R05	✓	✓		✓					✓
Rnn	✓		✓						

Fundamental of requirement 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,

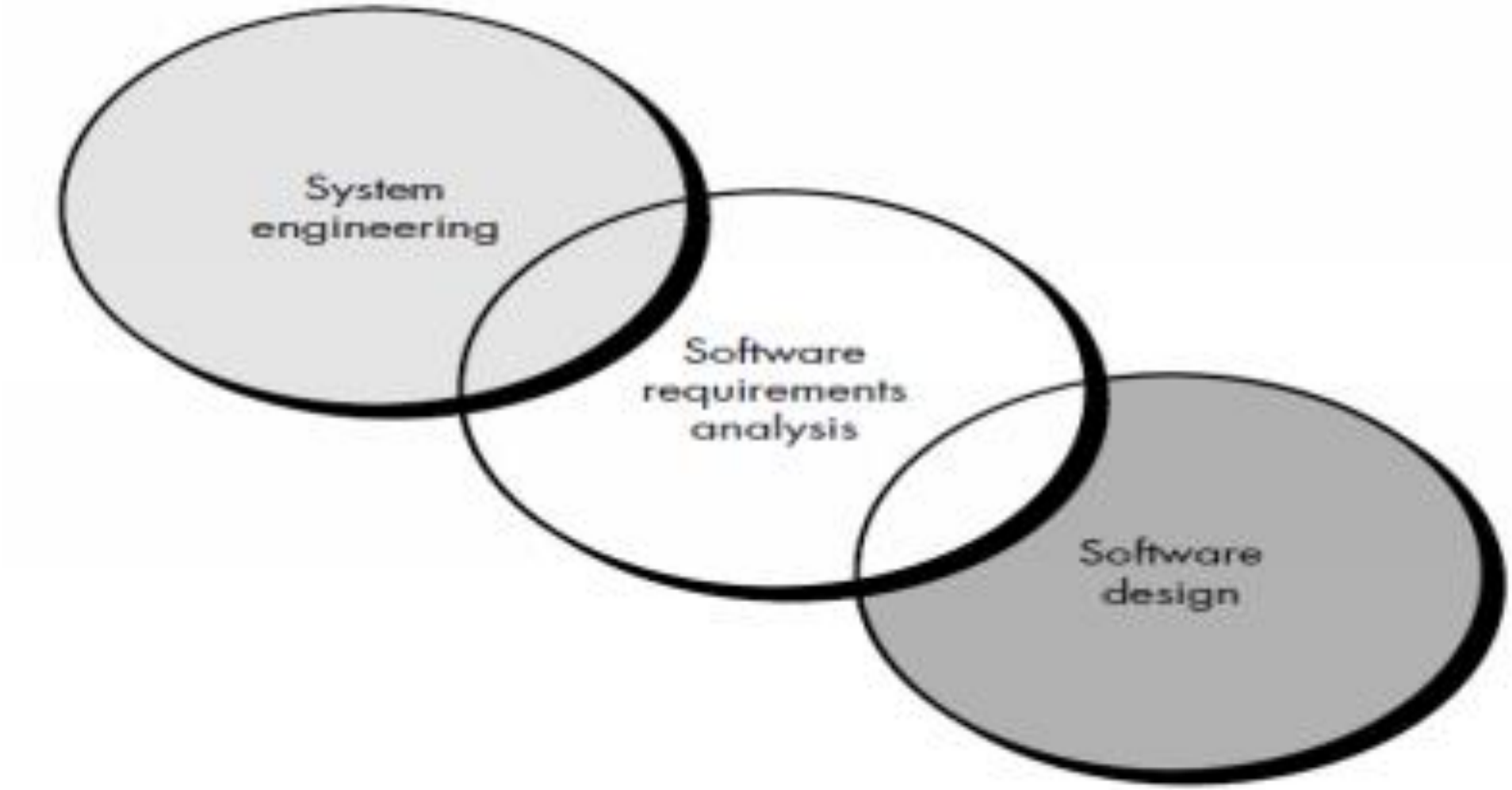
✓ indicate software's interface with other system elements, and

✓ establish constraints that software must meet.

Fundamental of requirement analysis

- Requirements analysis allows the software engineer (called 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.

Fundamental of requirement analysis



Requirements Elicitation For Software

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.

For example, the analyst might ask:

a) Who is behind the request for this work?

b) Who will use the solution?

c) What will be the economic benefit of a successful solution?

d) Is there another source for the solution that you need?

Requirements Elicitation For Software

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:

a) How would you characterize "good" output that would be generated by a successful solution?

b) What problem(s) will this solution address?

c) Can you show me (or describe) the environment in which the solution will be used?

d) Will special performance issues or constraints affect the way the solution is approached?

Requirements Elicitation For Software

The final set of questions focuses on the effectiveness of the meeting.

a) Are you the right person to answer these questions? Are your answers "official"?

b) Are my questions relevant to the problem that you have?

c) Am I asking too many questions?

d) Can anyone else provide additional information?

e) Should I be asking you anything else?

Facilitated Application Specification Techniques (FAST)

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 .

Facilitated Application Specification Techniques (FAST)

The 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.

Facilitated Application Specification Techniques (FAST)

- 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.

Quality Function Deployment (QFD)

- 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‖
- QFD identifies three types of requirements.

✓ Normal requirements.

✓ Expected requirements.

✓ Exciting requirements.

Quality Function Deployment (QFD)

I. 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,

  Specific system functions, and

  Defined levels of performance.

Quality Function Deployment (QFD)

II. 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.

Quality Function Deployment (QFD)

III. Exciting requirements

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

• For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multi-touch screen, visual voice mail) that delight every user of the product

Use-Cases

It is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users.

• To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called **use cases** ,provide a description of how the system will be used.

Use-Cases

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.

• An actor is anything that communicates with the system or product and that is external to the system itself.

Use Cases

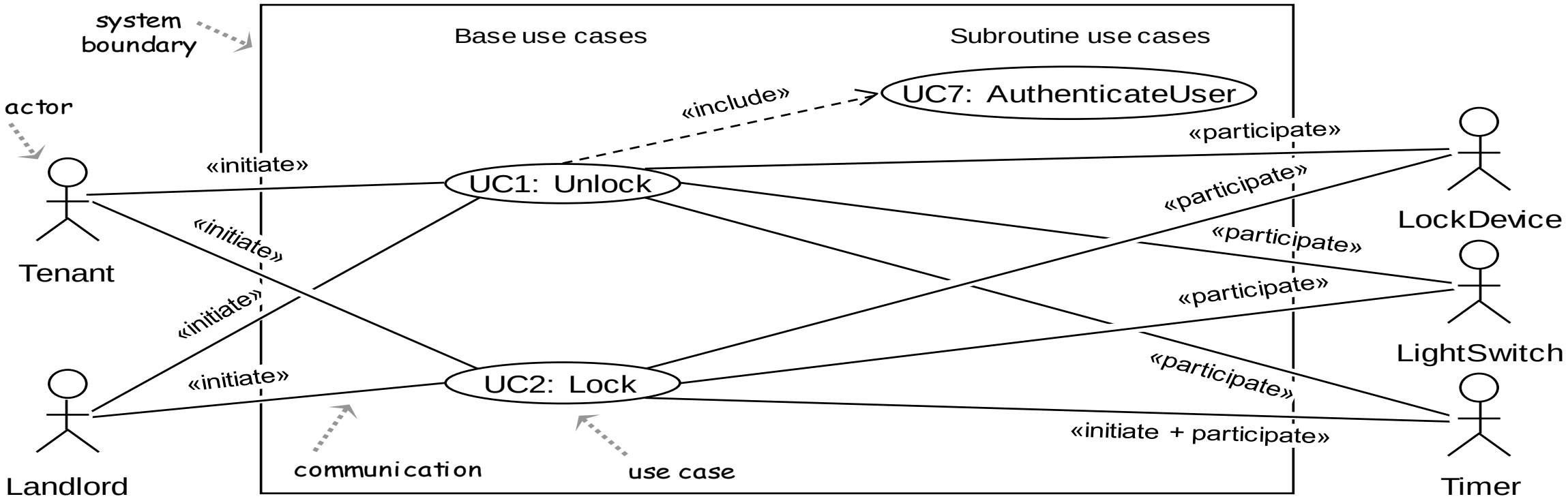
- Used for **Functional Requirements** Analysis and Specification
- A ***use case*** is a **step-by-step description** of how a user will use the system-to-be to accomplish business goals
  - Detailed use cases are usually written as *usage scenarios* or *scripts*, showing an envisioned sequence of actions and interactions between the external actors and the system-to-be

Types of Actors

- ***Initiating actor*** (also called ***primary actor*** or simply “user”): initiates the use case to achieve a goal
- ***Participating actor*** (also called ***secondary actor***): participates in the use case but does not initiate it.
- **Subtypes of participating actors:**
  - ***Supporting actor***: helps the system-to-be to complete the use case
  - ***Offstage actor***: passively participates in the use case, i.e., neither initiates nor helps complete the use case, but may be notified about some aspect of it (e.g., for keeping records)

Use Case Diagram: Device Control

UC1: Unlock
UC2: Lock
UC3: AddUser
UC4: RemoveUser
UC5: InspectAccessHistory
UC6: SetDevicePrefs
UC7: AuthenticateUser
UC8: Login



Interestingly, we keep mentioning “home” and “doors,” but they do not appear as actors!
This issue will be discussed in a later lecture (Domain Model)

Analysis Principles

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

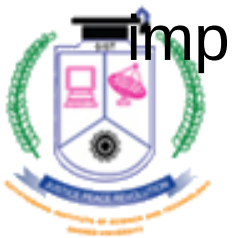
a) The information domain of a problem must be represented and understood.

b) The functions that the software is to perform must be defined.

c) The behavior of the software (as a consequence of external events) must be represented.

d) The models that depict information function and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.

e) The analysis process should move from essential information toward implementation detail.



Davis suggests a set of guiding principles for Requirements Engineering:

a) Understand the problem before you begin to create the analysis model.

b) Develop prototypes that enable a user to understand how human/machine interaction will occur.

c) Record the origin of and the reason for every requirement.

d) Use multiple views of requirements. Building data, functional, and behavioral models provide the software engineer with three different views.

e) Rank requirements. If an incremental process model is applied, those requirements to be delivered in the first increment must be identified.

f) Work to eliminate ambiguity. Because most requirements are described in a natural language. The use of formal technical reviews is one way to uncover and eliminate ambiguity

The Information Domain :

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 Information Domain :

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
2. Information flow
3. Information structure

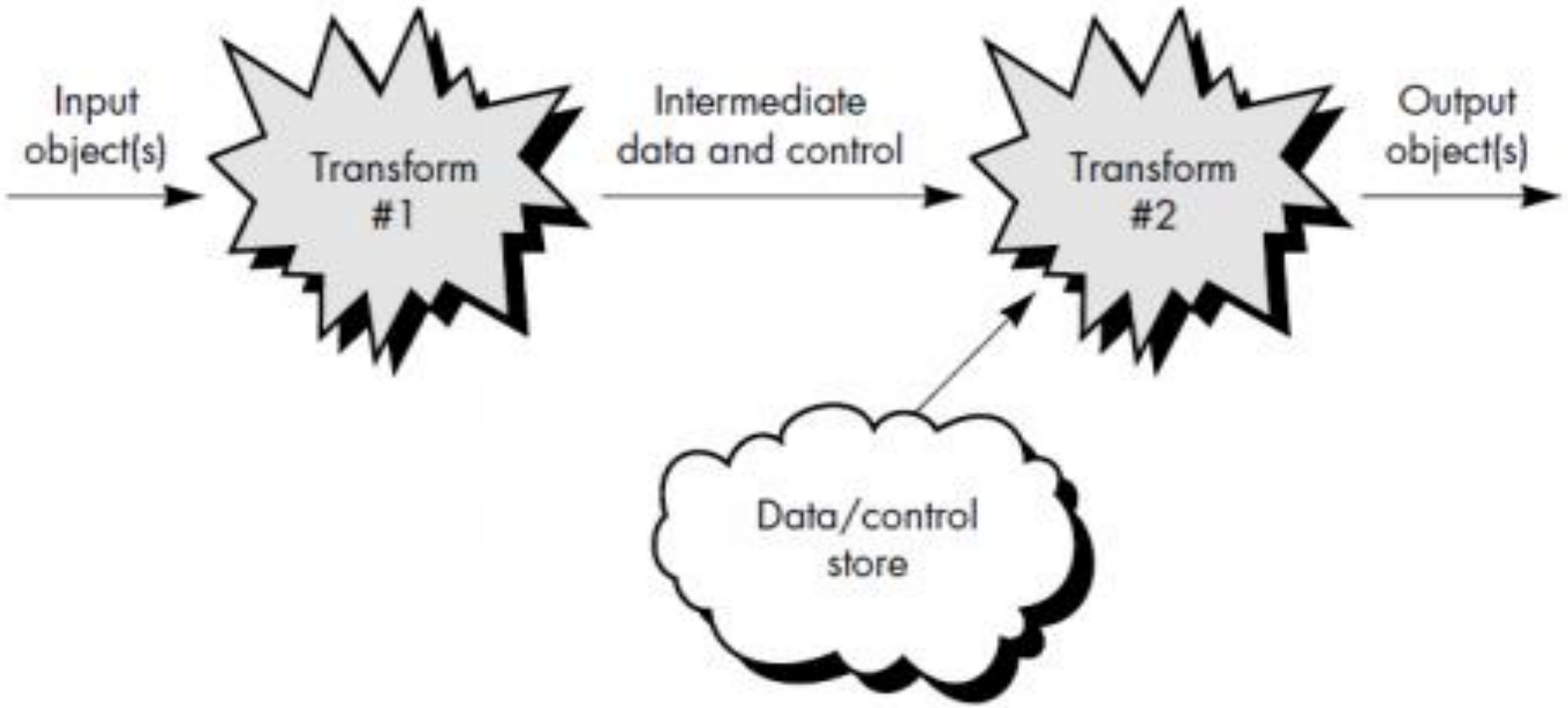
Information content represents the individual data and control objects that constitute some larger collection of information transformed by the software.

- 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.

Information flow represents the manner in which data and control change as each moves through a system.

- Information structure represents the internal organization of various data and control items.

The Information Domain



Modeling

 We create functional models to gain a better understanding of the actual entity to be built.

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

Functional models.

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

input,

processing, and

output.

 The functional model begins with a single context level model. Over a series of iterations, more and more functional detail is provided, until a thorough delineation of all system functionality is represented.

Behavioural 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 that is changed only when some event occurs

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.

Partitioning

Problems are often too large and complex to be understood as a whole.

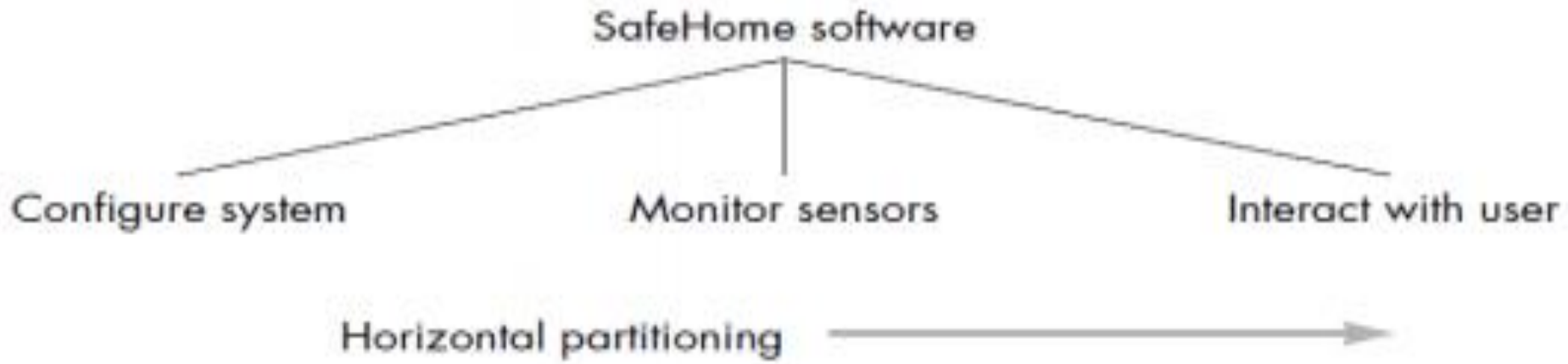
 For this reason, we tend to partition such problems into parts that can be easily understood and establish interfaces between the parts so that overall function can be accomplished.

 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

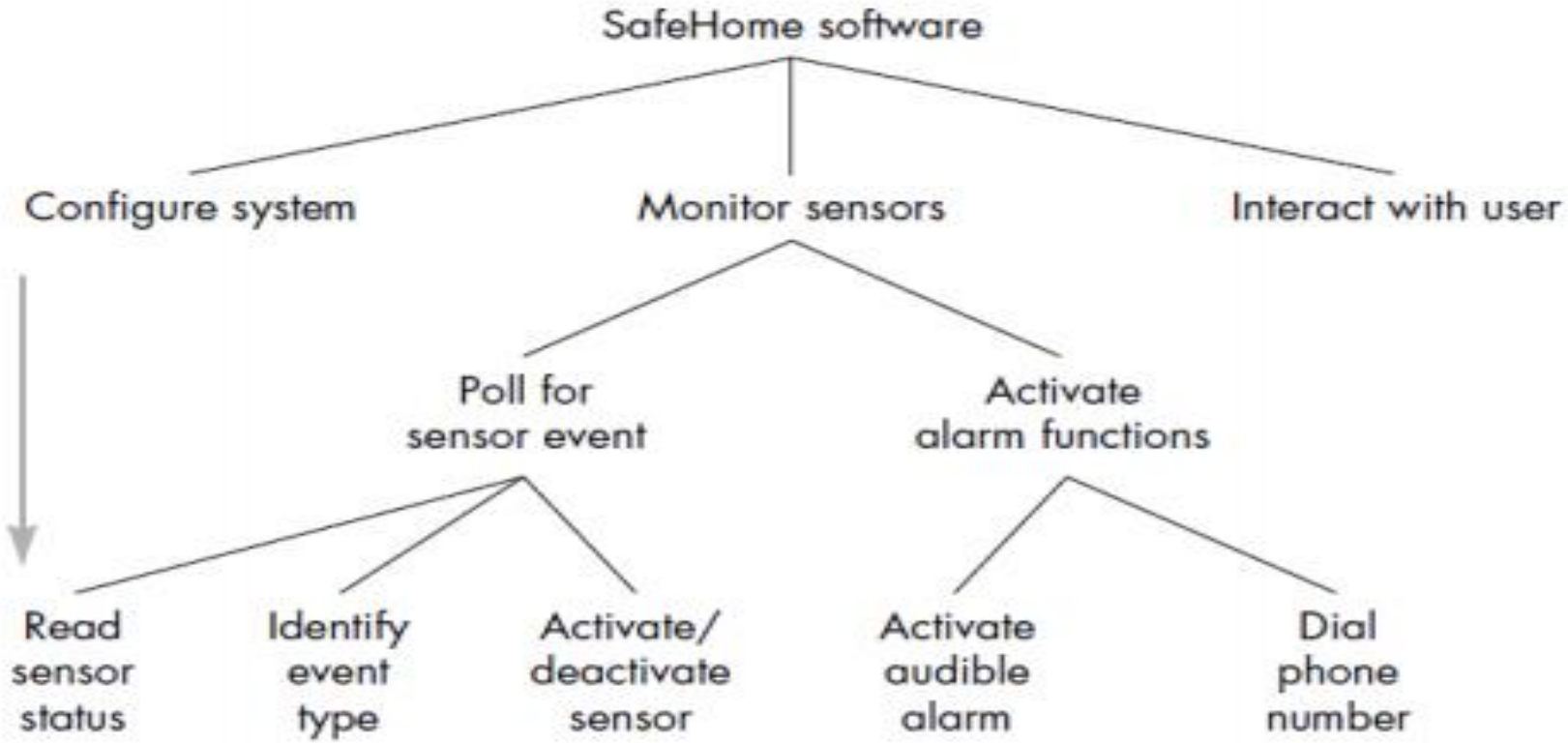
(2) Functionally decomposing the problem by moving horizontally in the hierarchy.

Horizontal Partitioning



Horizontal partitioning of SafeHome function

Vertical Partitioning



Vertical partitioning of Safe Home function

Software Prototyping

Rapid software development to validate requirements

 Prototyping is the process of quickly putting together a working model in order to test various aspects of a design, illustrate ideas or features and gather early user feedback.

Uses of prototypes

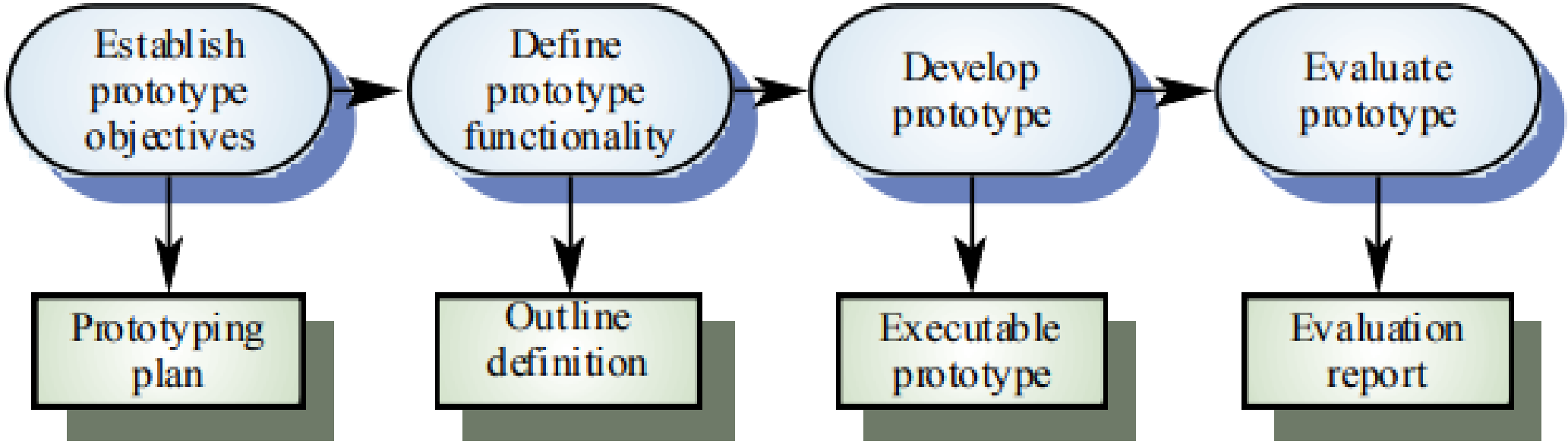
 The principal use is to help customers and developers understand the requirements for the system

 Prototyping can be considered as a risk reduction activity which reduces requirements risks

Prototyping benefits

- Misunderstandings between software users and developers are exposed
- Missing services may be detected and confusing services may be identified
- A working system is available early in the process
- The prototype may serve as a basis for deriving a system specification
- The system can support user training and system testing

Prototyping process



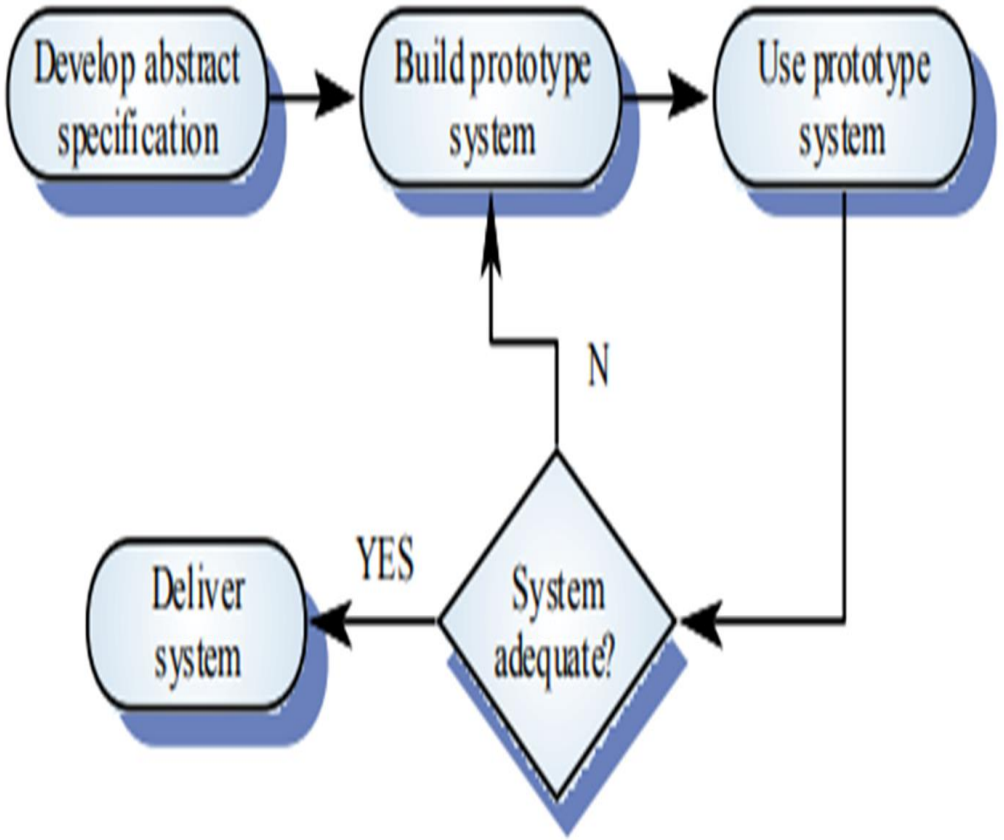
Types of prototyping

- Evolutionary prototyping
- Throw-away prototyping

Evolutionary prototyping

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.

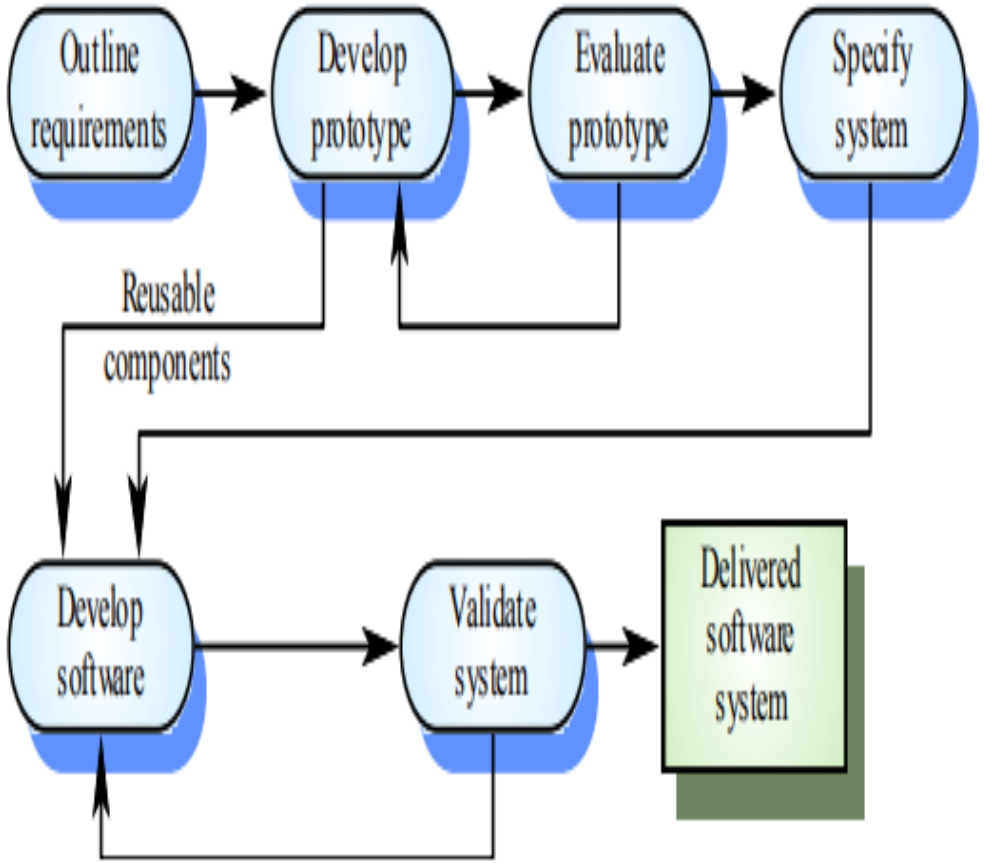
- The prototype of the software is the first evolution of the finished system.



Throw-away prototyping

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.



Selecting the appropriate prototyping approach

Question	Throwaway prototype	Evolutionary prototype	Additional preliminary work required
Is the application domain understood?	Yes	Yes	No
Can the problem be modeled?	Yes	Yes	No
Is the customer certain of basic system requirements?	Yes/No	Yes/No	No
Are requirements established and stable?	No	Yes	Yes
Are any requirements ambiguous?	Yes	No	Yes
Are there contradictions in the requirements?	Yes	No	Yes

Prototyping Methods and Tools

To conduct rapid prototyping, 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 end application generators, and
- other very high-level non procedural languages.

Because 4GT enable the software engineer to generate executable code quickly, they are ideal for rapid prototyping.

Prototyping Methods and Tools

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.

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

Prototyping Methods and Tools

3. Formal specification and prototyping environments

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.

User interface prototyping

 It is impossible to pre-specify the look and feel of a user interface in an effective way. prototyping is essential

 UI development consumes an increasing part of overall system development costs.

 User interface generators may be used to _draw‘ the interface and simulate its functionality with components associated with interface entities.

 Web interfaces may be prototyped using a web site editor.

User interface prototyping- Techniques

a) Work with the real users.

The best people to get involved in prototyping are the ones who will actually use the application when it is done.

These are the people who have the most to gain from a successful implementation;

These are the people who know their own needs best.

User interface prototyping- Techniques

b) Get your stakeholders to work with the prototype.

Just as if you want to take a car for a test drive before you buy it, your users should be able to take an application for a test drive before it is developed.

Furthermore, by working with the prototype hands-on, they can quickly determine whether the system will meet their needs.

A good approach is to ask them to work through some use case scenarios using the prototype as if it were the real system.

User interface prototyping- Techniques

c) Understand the underlying business.

You need to understand the underlying business before you can develop a prototype that will support it.

The more you know about the business, the more likely it is you can build a prototype that supports it.

Once again, active stakeholder participation is critical to your success.

d) You should only prototype features that you can actually build.

If you cannot possibly deliver the functionality, do not prototype it.

User interface prototyping- Techniques

e) **You cannot make everything simple**.

Sometimes your software will be difficult to use because the problem it addresses is inherently difficult. Your goal is to make your user interface as easy as possible to use, not simplistic.

f) **It's about what you need**.

Their point is a good user interface fulfills the needs of the people who work with it.

It isn‘t loaded with a lot of interesting, but unnecessary, features.

g) Get an interface expert to help you design it.

User interface experts understand how to develop easy-to-use interfaces, whereas you probably do not.

A generalizing specialist with solid UI skills would very likely be an ideal member of your development team.

h) Explain what a prototype is.

The biggest complaint developers have about UI prototyping is their users say —That‘s great. Install it this afternoon.‖ This happens because users do not realize more work is left to do on the system. The reason this happens is simple: From your user's point-of-view, a fully functional application is a bunch of screens and reports tied together by a menu.

User interface prototyping- Techniques

i) Consistency is critical.

Inconsistent user interfaces lead to less usable software,

more programming, and

greater support and training costs.

j) Avoid implementation decisions as long as possible.

Be careful about how you name these user interface items in your requirements documents.

Strive to keep the names generic, so you do not imply too much about the implementation technology.

User interface prototyping- Techniques

k) Small details can make or break your user interface.

Have you ever used some software, and then discarded it for the product of a competitor because you didn‘t like the way it prints, saves files, or some other feature you simply found too annoying to use? I have.

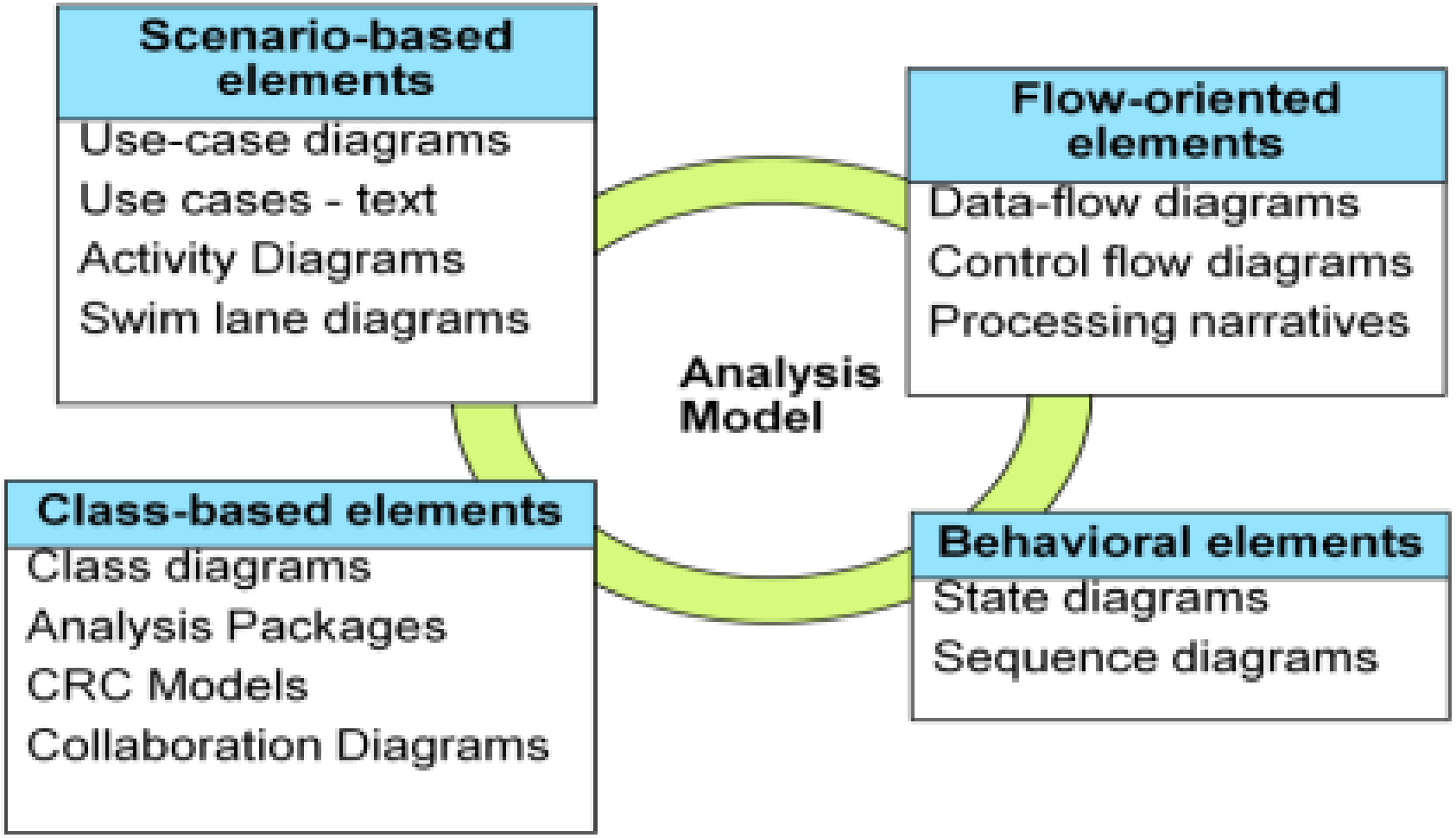
Although the rest of the software may have been great, that vendor lost my business because a portion of its product‘s user interface was deficient.

Software Document Analysis and Modeling

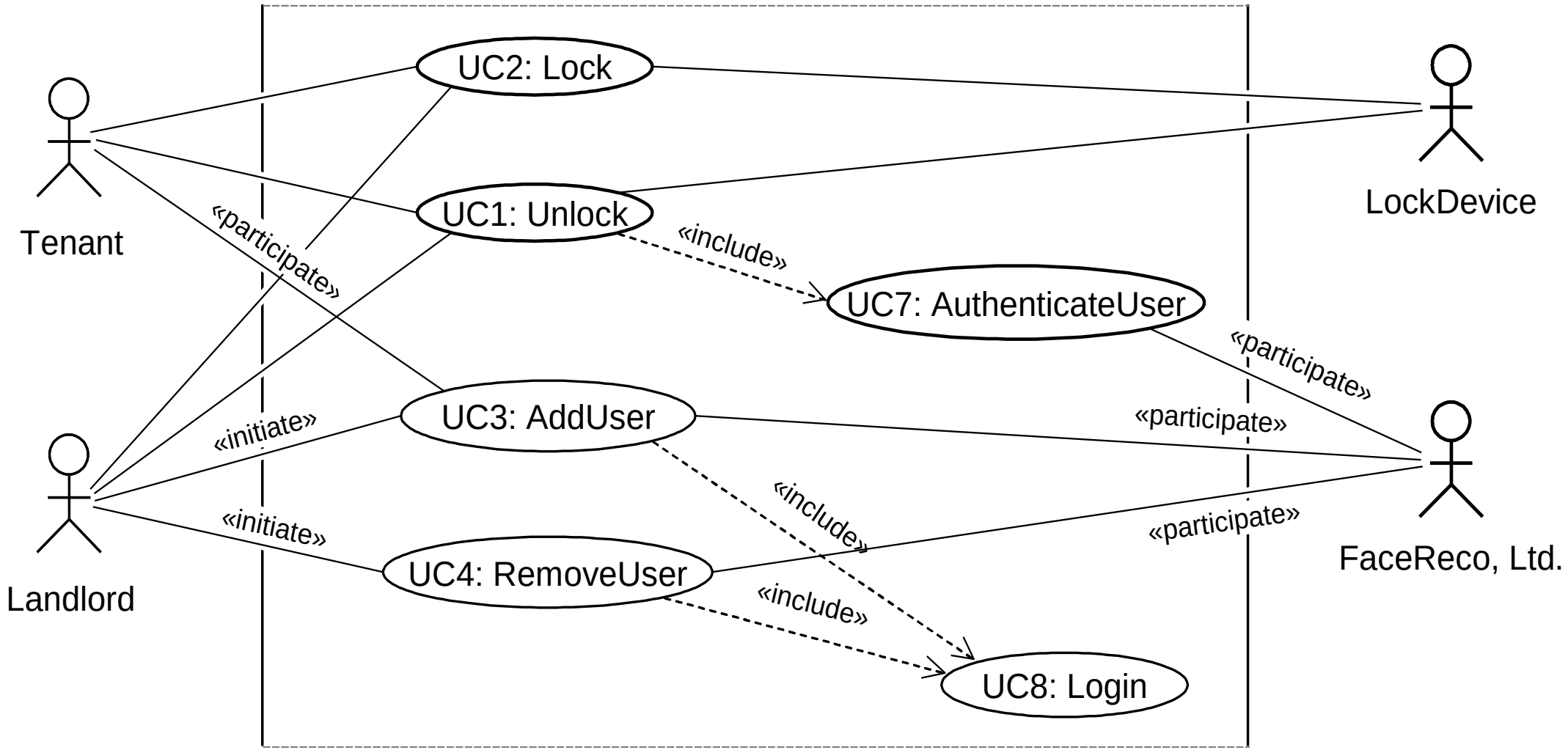
Goals of Analysis Modeling

- Provides the first technical representation of a system
- Is easy to understand and maintain.
- Deals with the problem of size by partitioning the system.
- Uses graphics whenever possible.
- Differentiates between essential information versus implementation information.
- Helps in the tracking and evaluation of interfaces
- Provides tools other than narrative text to describe software logic and policy.

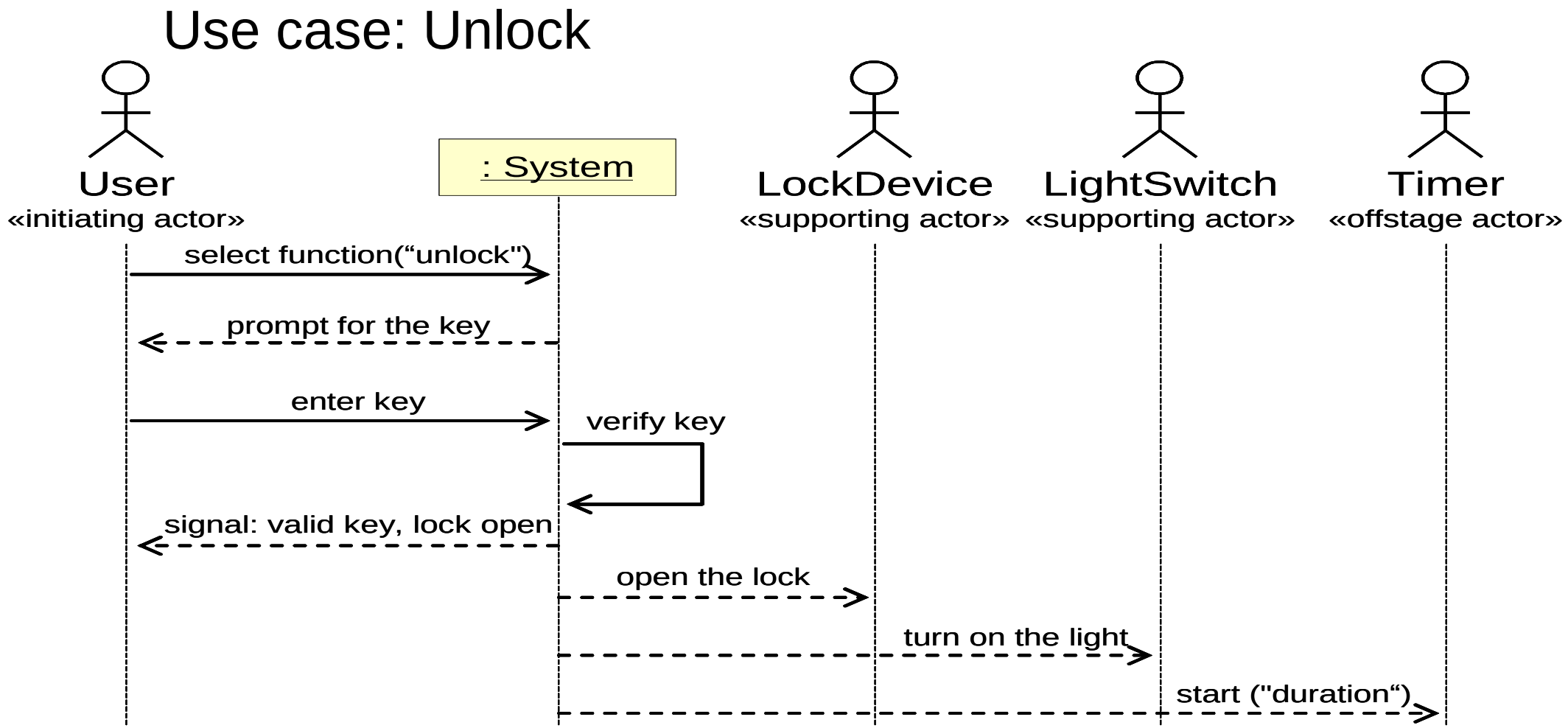
Elements of the Analysis Model



Use Case Diagram



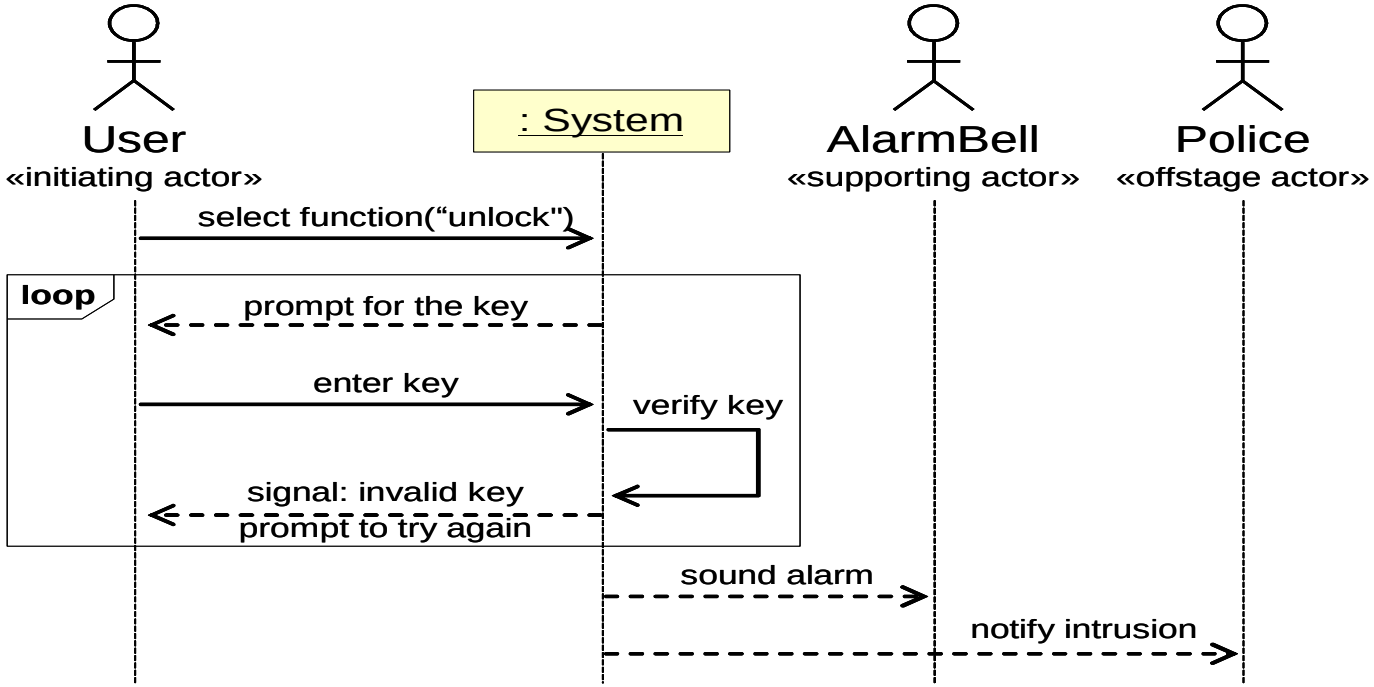
Sequence Diagram



System Sequence Diagram

[Modeling System Workflows]

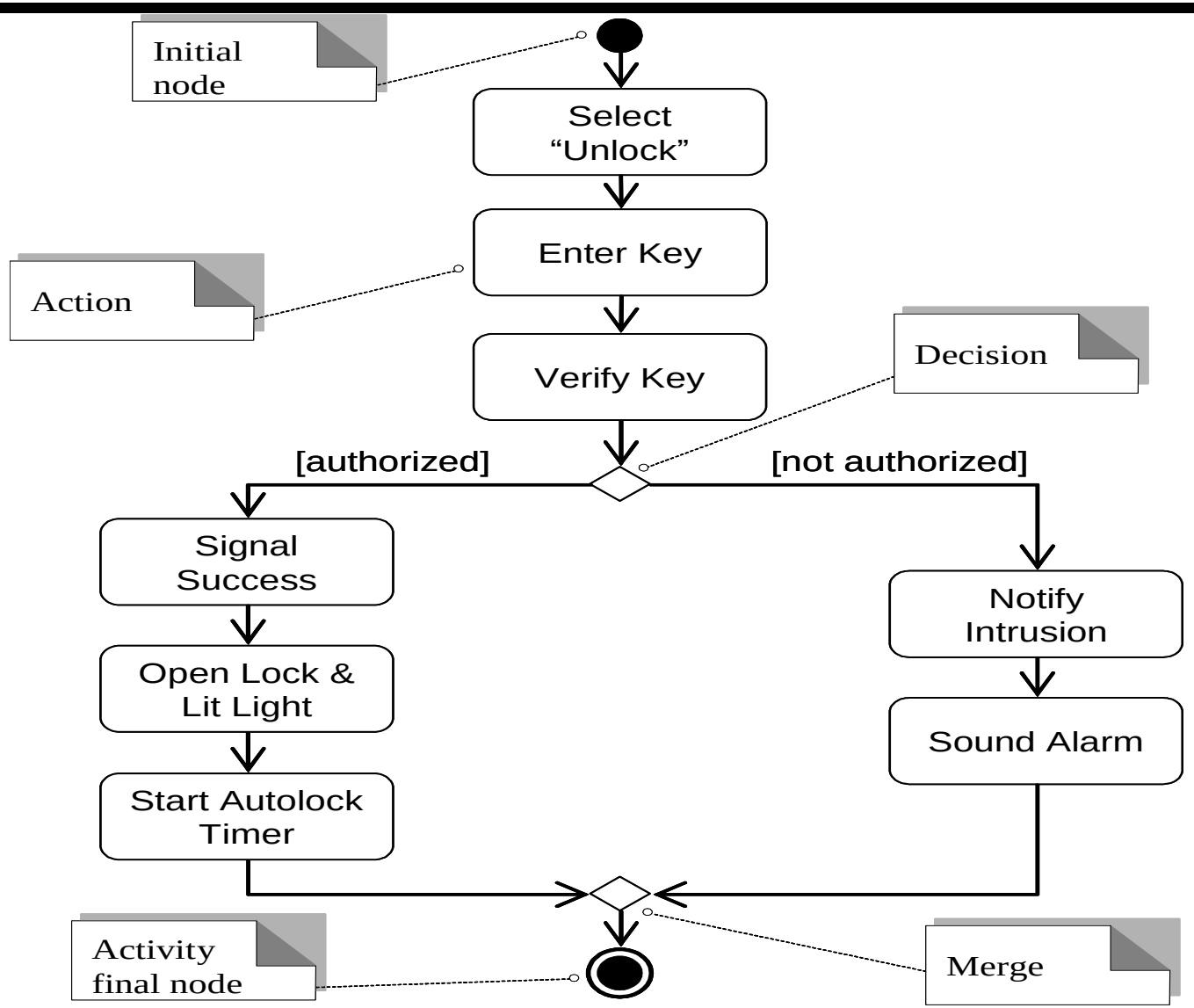
Use case: Unlock



Alternate scenario (burglary attempt)

Activity Diagram

[Modeling System Workflows]



Data Modeling

- Examines data objects independently of processing
- Focuses attention on the data domain
- Creates a model at the customer‘s level of abstraction
- Indicates how data objects relate to one another.

Data Objects

A data object can be an external entity , a thing ,an occurrence or event ,a role, an organizational unit, a place, or a structure.

Data Modeling

Attributes

- Data attributes define the properties of a data object
- They can be used to

(1) name an instance of the data object,

(2) describe the instance, or

(3) make reference to another instance in another table

- In addition, one or more of the attributes must be defined as an identifier— that is, the identifier attribute becomes a "key" when we want to find an instance of the data object.

Data Modeling

Instance

Naming attributes

Ties one data object to another, in this case, owner

Identifier

Descriptive attributes

Referential attributes

Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

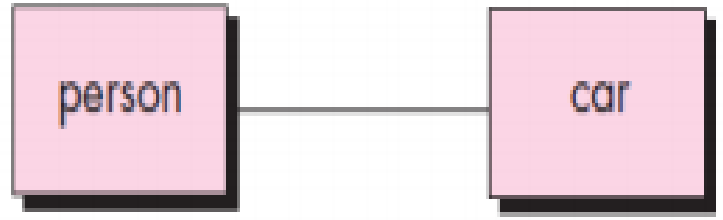
Tabular representation of data objects

Relationships

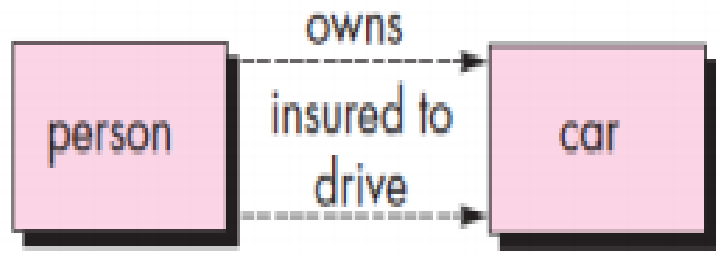
- Data objects are connected to one another in different ways.
- Consider the two objects person and car ,a connection is established between them because they are related.

For example

- A person owns a car
- A person is insured to drive a car.



(a) A basic connection between data objects



(b) Relationships between data objects

Cardinality

Cardinality and Modality We have defined a set of objects and represented the object/relationship pairs that bind them. But a simple pair that states:

object X relates to object Y does not provide enough information for software engineering purposes.

We must understand how many occurrences of object X are related to how many occurrences of object Y. This leads to a data modeling concept called cardinality.

Cardinality is usually expressed as simply 'one' or 'many.

Cardinality

- **One-to-one (I:I)**—An occurrence of [object] 'A' can relate to one and only one occurrence of [object] 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'
- **One-to-many (I:N)—**One occurrence of [object] 'A' can relate to one or many occurrences of [object] 'B,' but an occurrence of 'B' can relate to only one occurrence of 'A.'
- **Many-to-many (M:N)—**An occurrence of [object] 'A' can relate to one or more occurrences of 'B,' while an occurrence of 'B' can relate to one or more occurrences of 'A.'

Modality

The modality of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional.

The modality is 1 if an occurrence of the relationship is mandatory.

Functional and Behavioral Models

Functional Model - Data Flow Diagram (DFD)

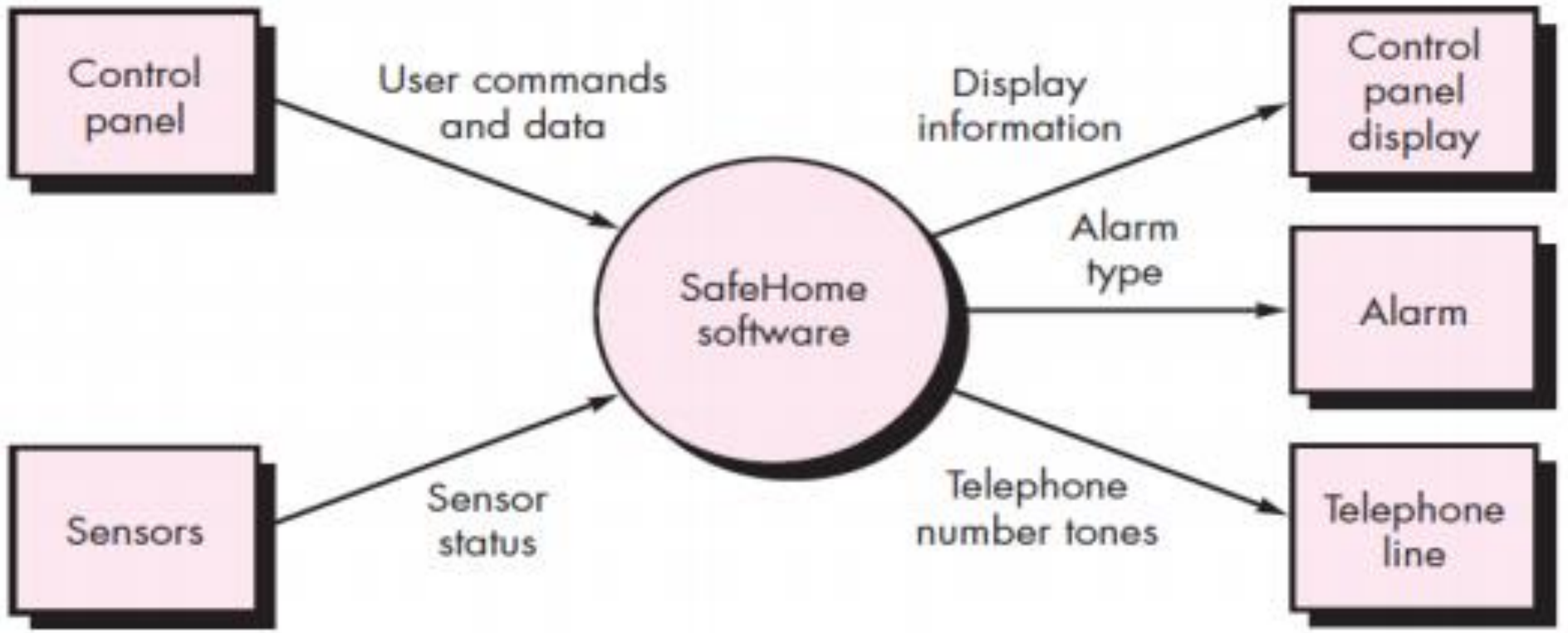
The DFD takes an input-process-output view of a system.

Data objects flow into the software ,

Transformed by processing elements, and

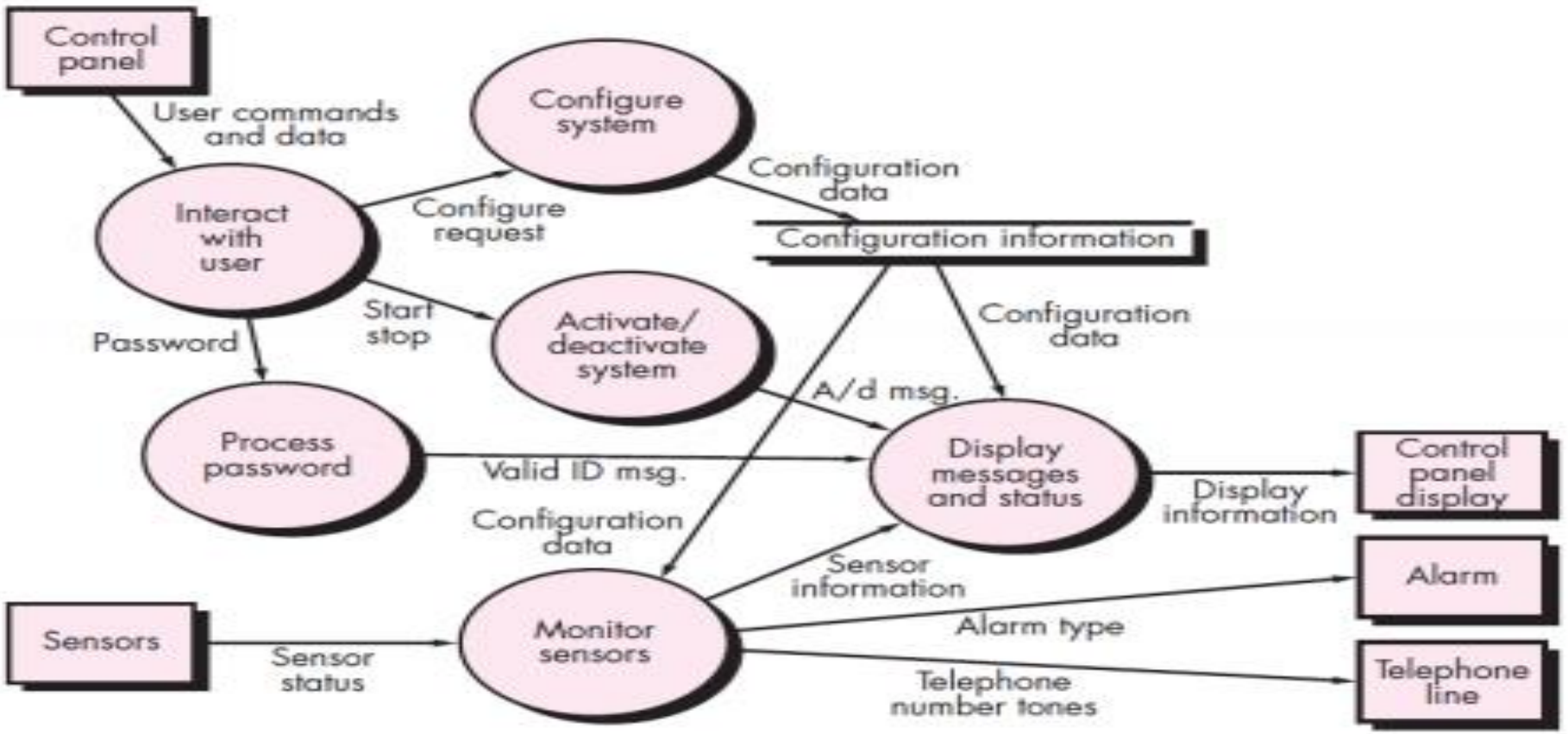
Resultant data objects flow out of the software.

Context level DFD for the SafeHome security function

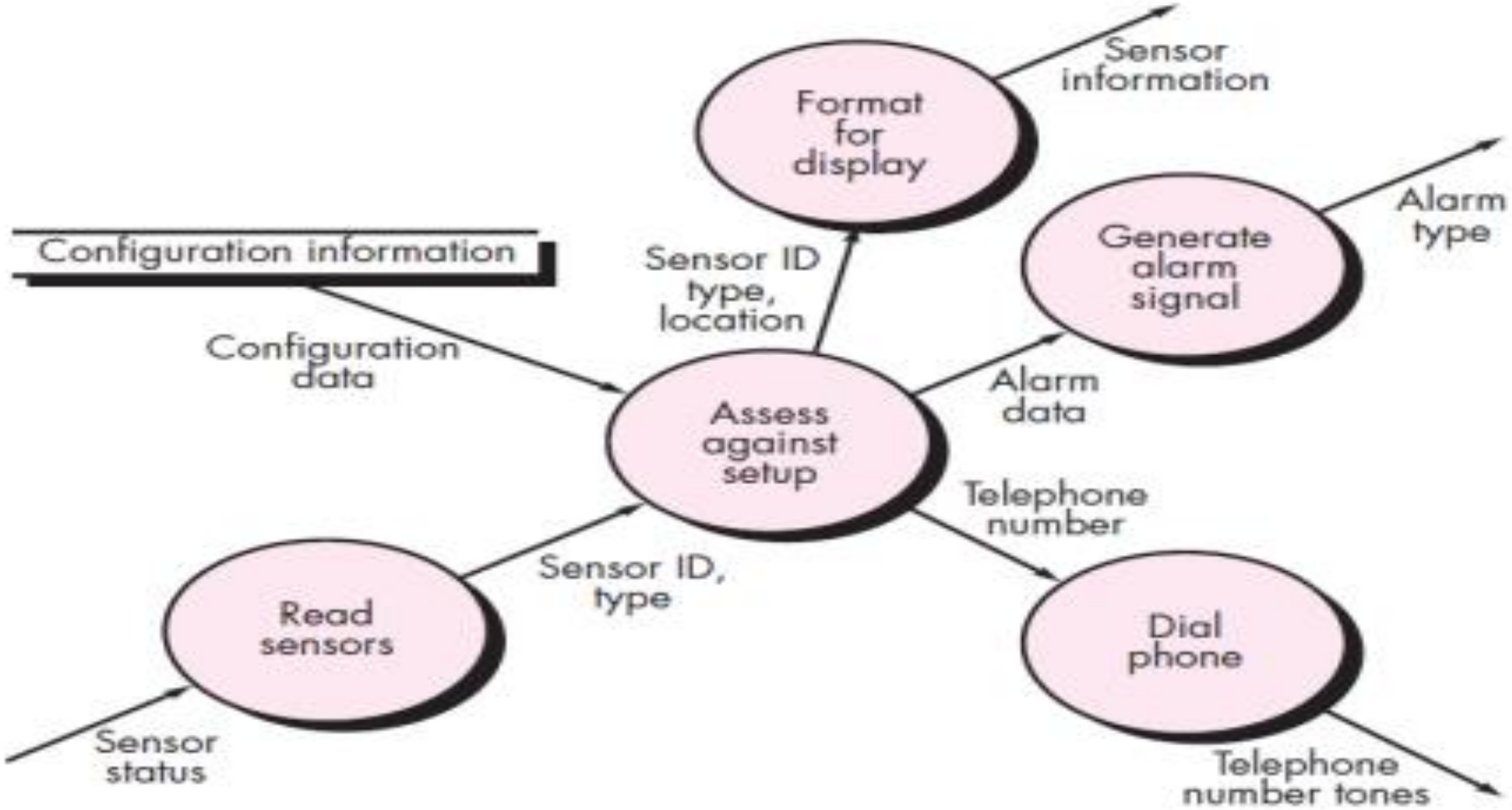


Flow Modeling Notation

Level 1 DFD for Safe Home security function



Level 2 DFD That Refines The Monitor Sensors Process



Behavioral modeling :State Transition Diagram(STD)

To create the model, you should perform the following steps:

a) Evaluate all use cases to fully understand the sequence of interaction within the system.

b) Identify events that drive the interaction sequence and understand how these events relate to specific objects.

c) Create a sequence for each use case.

d) Build a state diagram for the system.

e) Review the behavioral model to verify accuracy and consistency.

State Representations

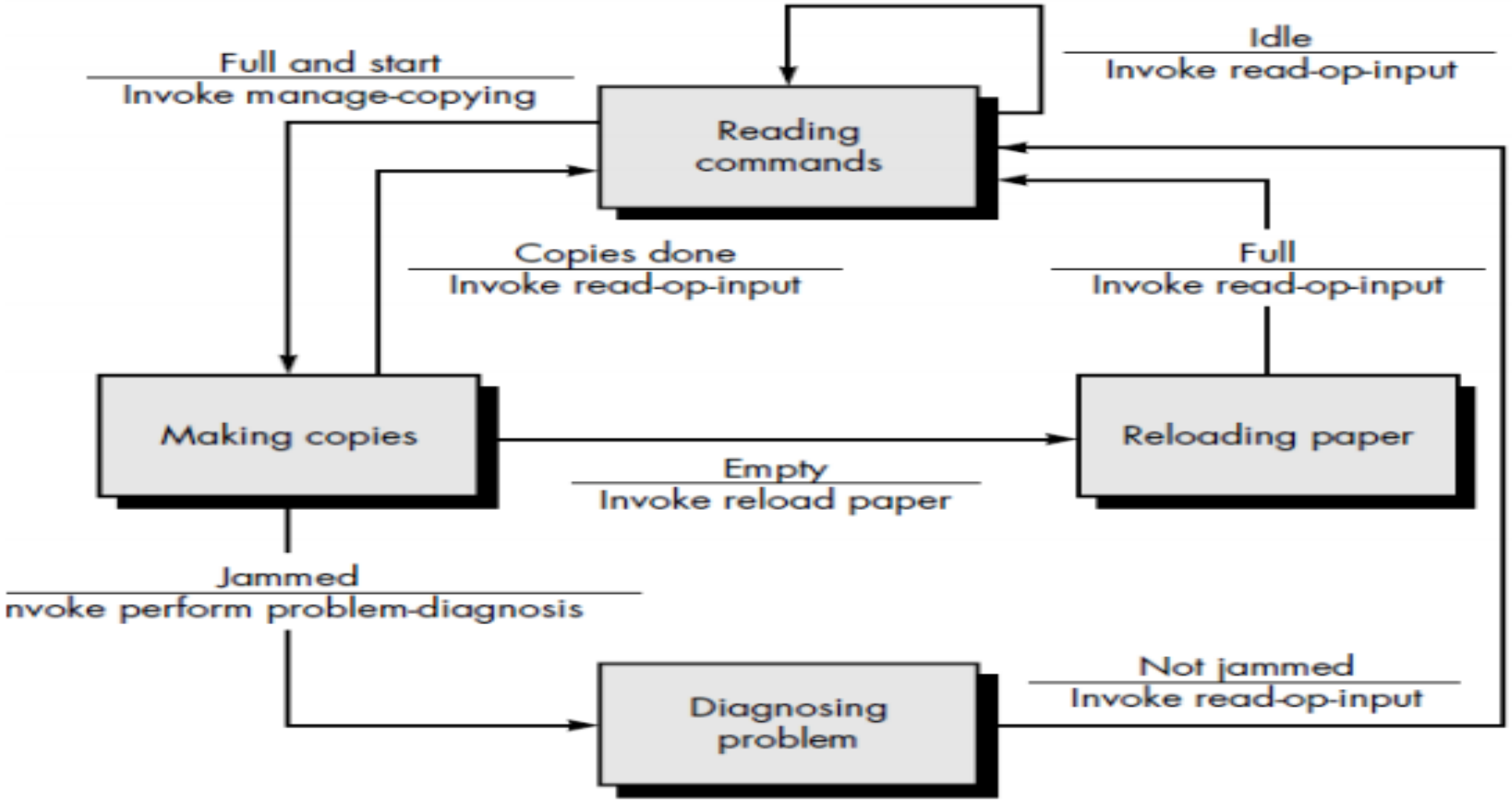
In the context of behavioral modeling, two different characterizations of states must be considered:

(1) The state of each class as the system performs its function and

(2)The state of the system as observed from the outside as the system performs its function.

 The state of a class takes on both passive and active characteristics.

State Representations



State Representations

A simplified state transition diagram for the photocopier software.

The rectangles represent system states.

The arrows represent transitions between states.

Each arrow is labeled with a ruled expression.

The top value indicates the event(s) that cause the transition to occur.

The bottom value indicates the action that occurs as a consequence of the event.

State Representations

Therefore, when the paper tray is full and the start button is pressed, the system moves from the reading commands state to the making copies state.

Structured analysis

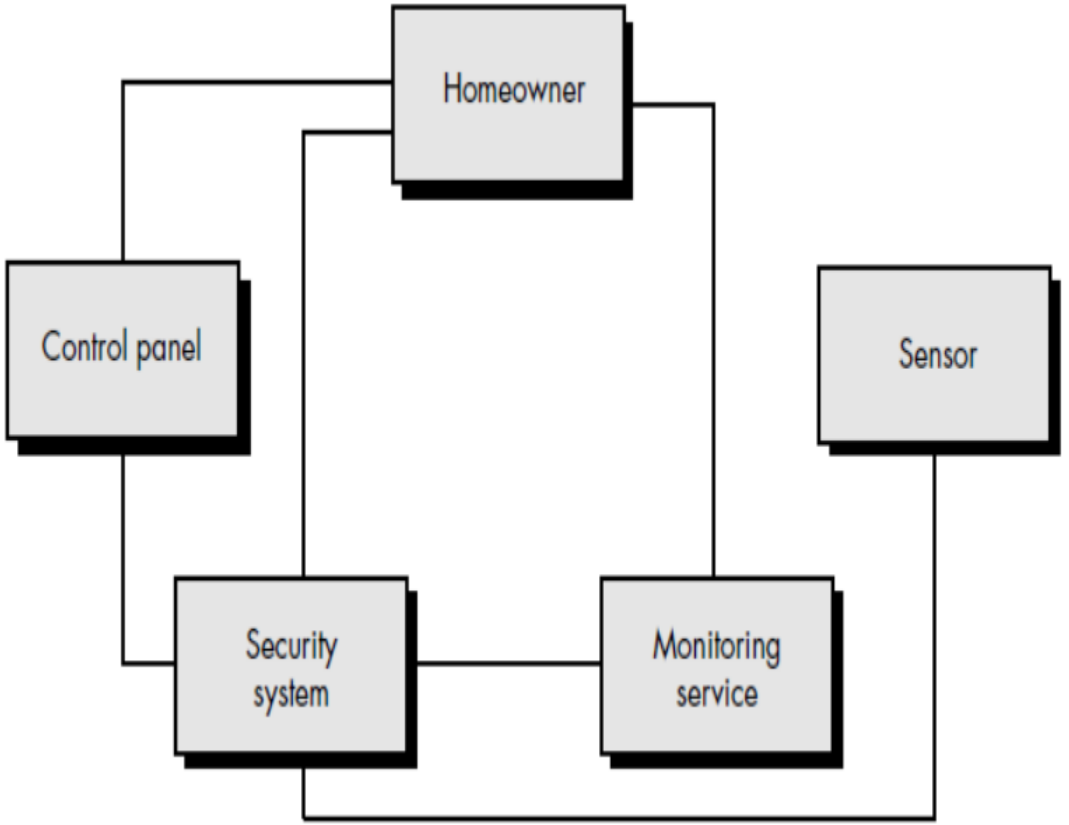
Structured analysis consist of three modeling techniques

- Functional model – Data Flow Diagram (DFD)
- Data Model – Entity Relationship Diagram (ERD)
- Behavioral Model – State Transition Diagram (STD)

Data model : Entity Relationship Diagram (ERD)

The system analyst and customer identifying the functional requirements and it will be modeled.

Creating Entity - Relationship model (ERD) ERD is constructed in a iterative fashion.



Data model : Entity Relationship Diagram (ERD)

The following steps are required.

1. During requirement elicitation, the customer asked to —Thingsll which are required. Things evolve input and output data object.
2. Taking the objects one at a time, define connection exist between data objects.
3. If connection exists, create one or more object relationship pairs.
4. For each object /relationship pair, cardinality and modality are explored.
5. Steps 2 to 4 are continue iteratively.
6. The attribute of each entity are defined.
7. ERD is formalized and reviewed.
8. Steps 1 to 7 are repeated until data modeling is complete.

Data model : Entity Relationship Diagram (ERD)

In Safe home software, the things are

- Homeowner
- Control panel
- Security System
- Monitoring service
- Sensor etc.

lines connecting the objects are noted.

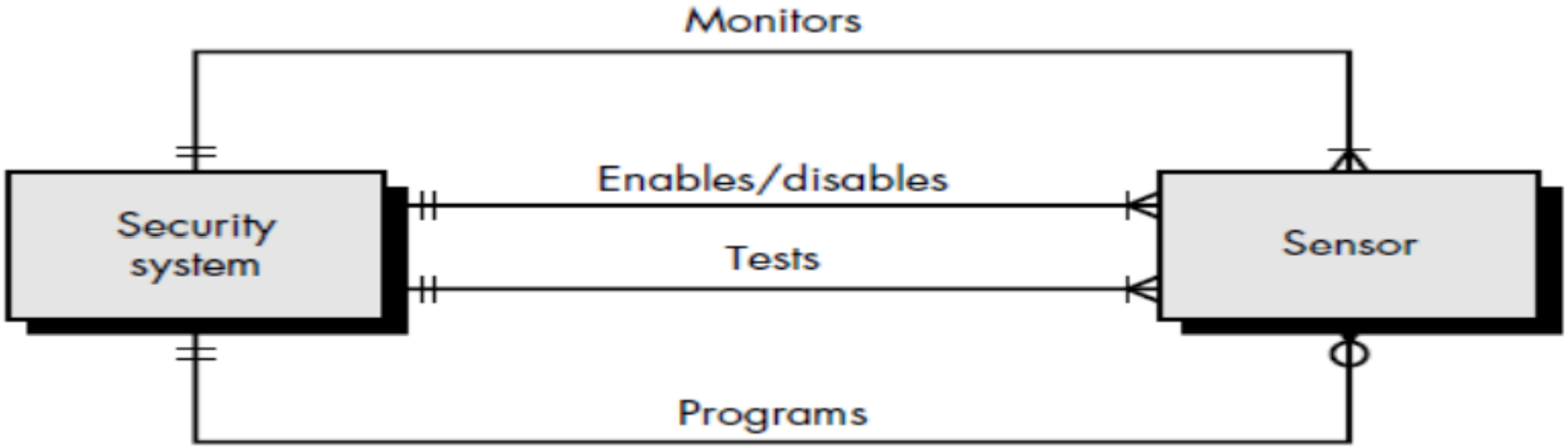
For example, direct connection exist between sensor and security system is noted.

Single connection between sensor and security system is identified.

Data model : Entity Relationship Diagram (ERD)

For example ,

- Security system monitors sensor
- Security system enables/disables sensor
- Security system tests sensor
- Security system programs sensor



Data model : Entity Relationship Diagram (ERD)

Cardinality and modality between object/relationship has been identified.

For example,

Security system to sensor is 1:m relationship exists.

Each attribute studied with its attributes.

For example, sensor object consists of sensor type, internal identification number, zone location and alarm.

Functional model – Data Flow Diagram (DFD)

Creating DFD contains the following guidelines.

a) Level 0 DFD depicts as a single bubble.

b) Primary input and output clearly noted.

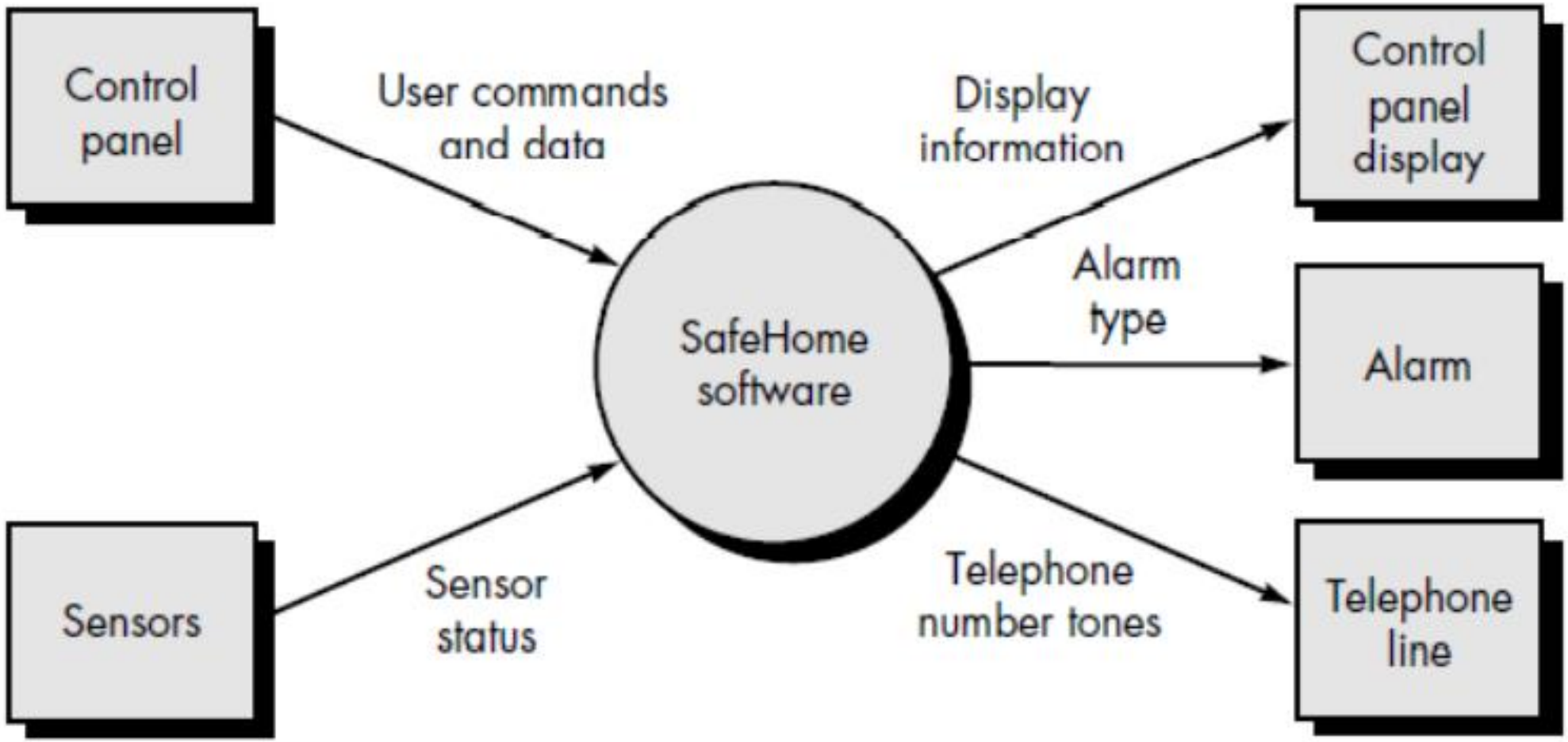
c) Refinement should begin by isolating candidate process.

d) All arrows and bubbles should be labeled with meaning

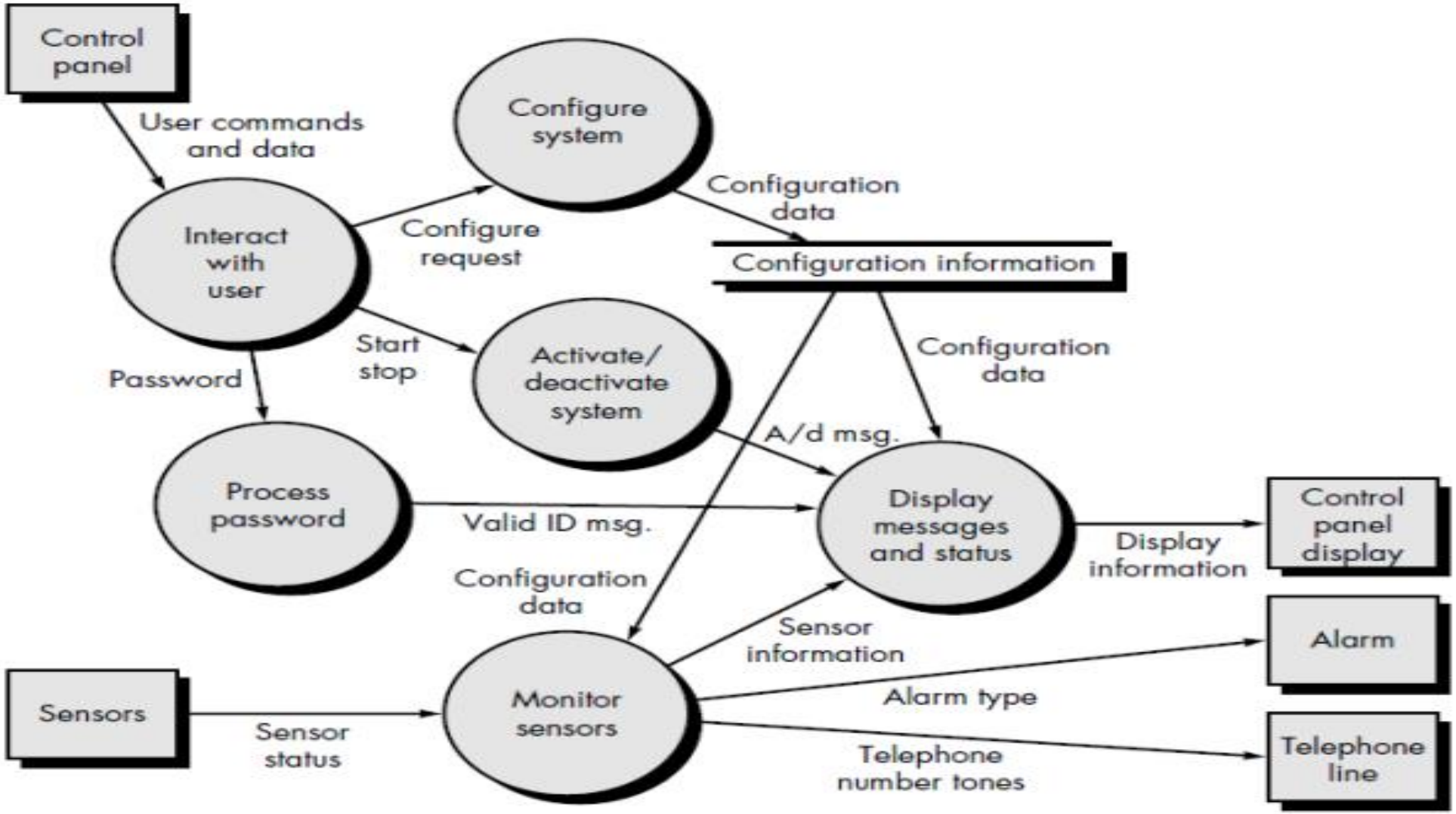
e) Information flow continuity must be maintained.

f) One bubble at a time should be refined

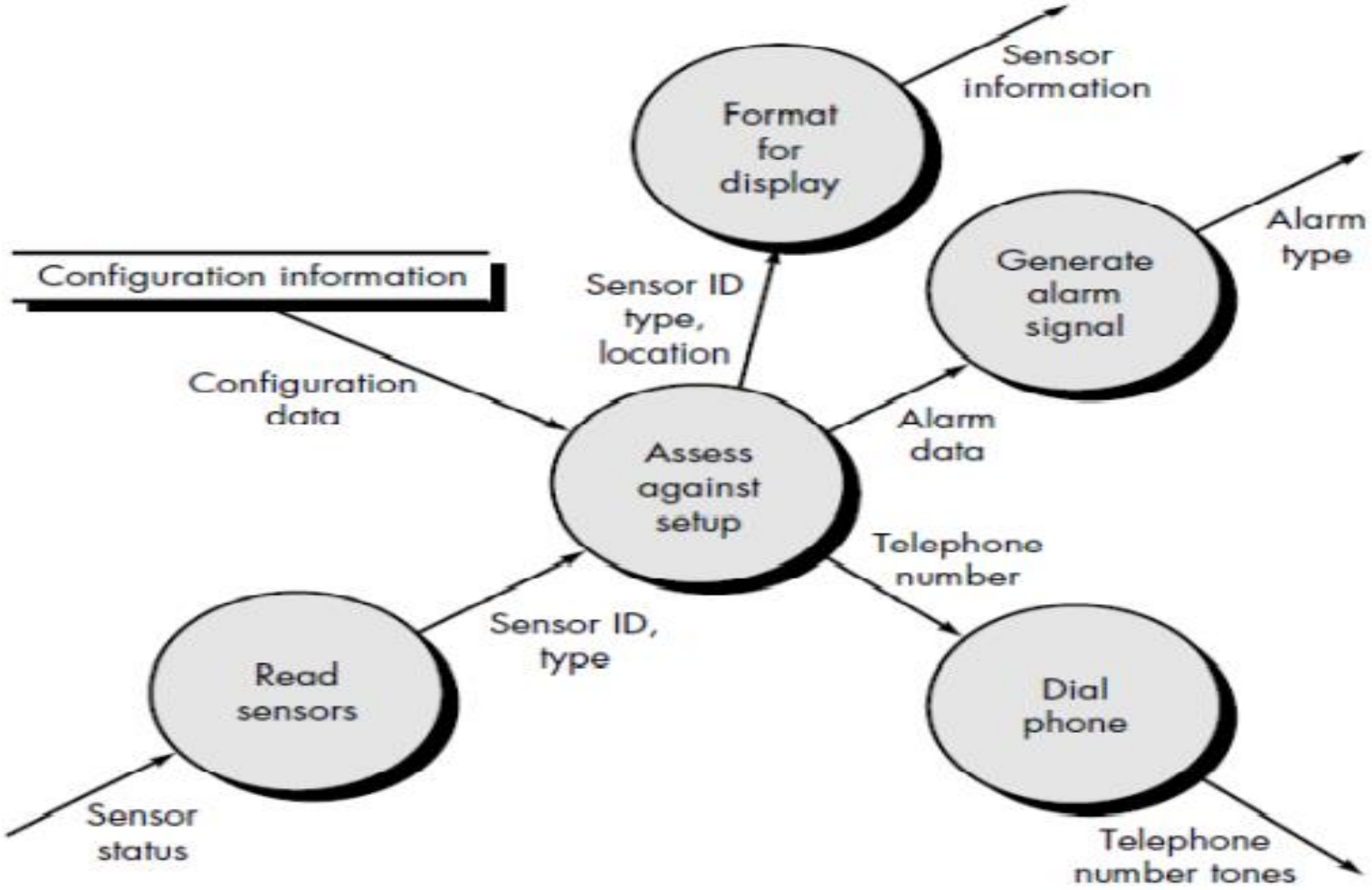
Data Flow Diagram (DFD)-Level 0 DFD of safe home software



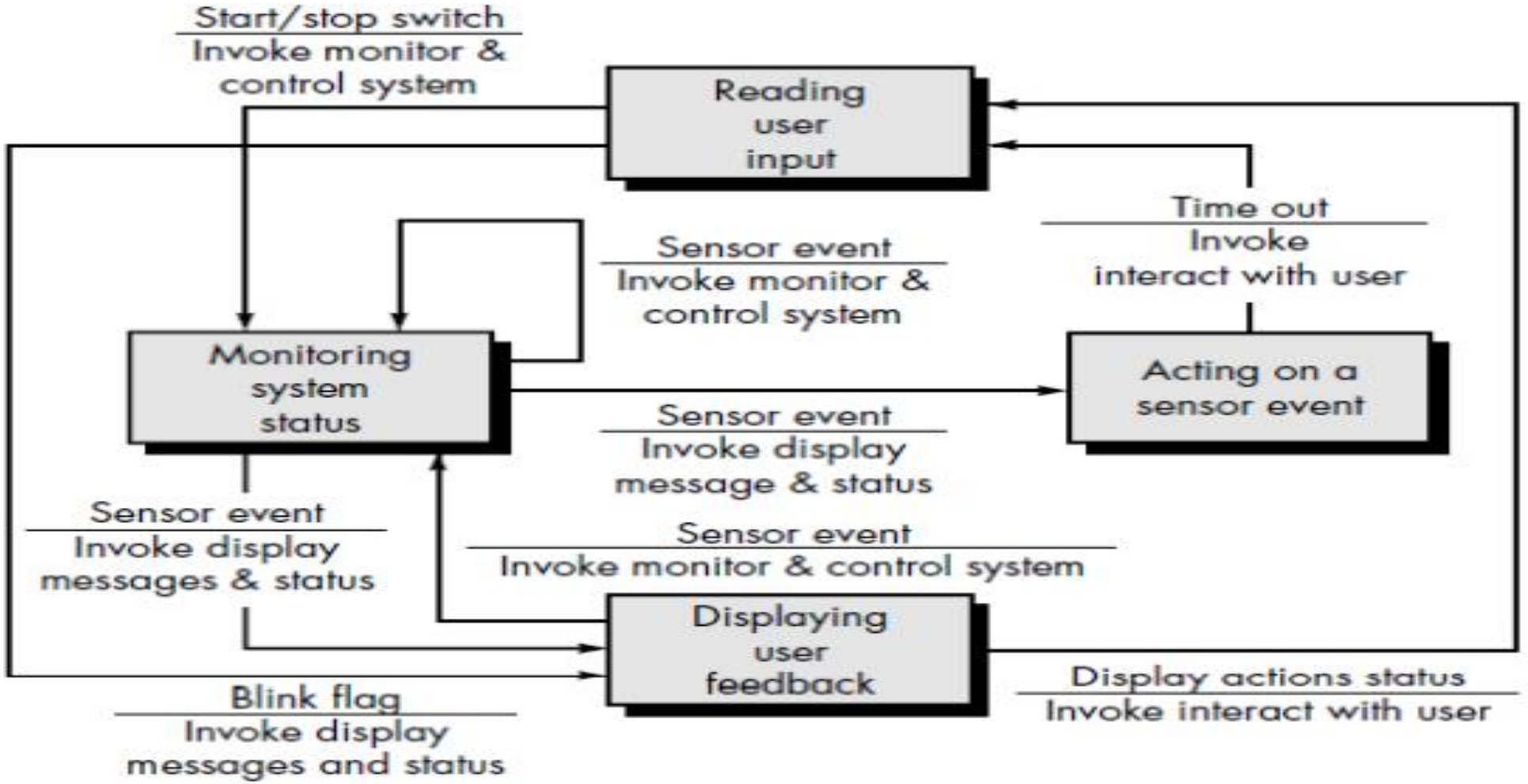
Data Flow Diagram (DFD)-Level 1 DFD of safe home software



Data Flow Diagram (DFD)-Level 2 DFD of safe home software



Behavioral Model – State Transition Diagram (STD)



Data Dictionary

The data dictionary is an organized listing of all data elements so that both user and system analyst will have a common understanding of inputs, outputs, components of stores and [even] intermediate calculations.

the data dictionary is always implemented as part of a CASE "structured analysis and design tool."

Data Dictionary

The data dictionary is an organized listing of all data elements so that both user and system analyst will have a common understanding of inputs, outputs, components of stores and [even] intermediate calculations.

The data dictionary is always implemented as part of a CASE "structured analysis and design tool.“

Although the format of dictionaries varies from tool to tool.

Data Dictionary

Most contain the following information:

- Name—the primary name of the data or control item, the data store or an external entity.
- Alias—other names used for the first entry.
- Where-used/how-used—a listing of the processes that use the data or control item and how it is used (e.g., input to the process, output from the process, as a store, as an external entity.

Data Construct	Notation	Meaning
	=	is composed of
Sequence	+	and
Selection	[|]	either-or
Repetition	{ }*n*	*n* repetitions of
	( )	optional data
	* ... *	delimits comments

The End