Elements of the analysis model.

1. Scenario based element This type of element represents the system user point of view. Scenario based elements are use case diagram, user stories.

2. Class based elements The object of this type of element manipulated by the system.

It defines the object, attributes and relationship.

The collaboration is occurring between the classes.

Class based elements are the class diagram, collaboration diagram.

Behavioural elements

Behavioural elements represent state of the system and how it is changed by the external events.

The behavioural elements are sequenced diagram, state diagram.

4. Flow oriented elements

An information flows through a computer-based system it gets transformed.

It shows how the data objects are transformed while they flow between the various system functions.

The flow elements are data flow diagram, control flow diagram.

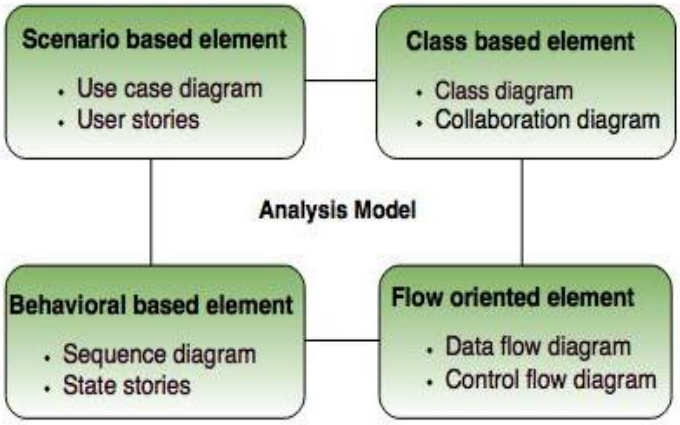


Fig. - Elements of analysis model

Incremental Model Processing

Incremental Model is a process of software development where requirements divided into multiple standalone modules of the software development cycle. In this model, each module goes through the requirements, design, implementation and testing phases. Every subsequent release of the module

adds function to the previous release. The process continues until the complete system achieved.

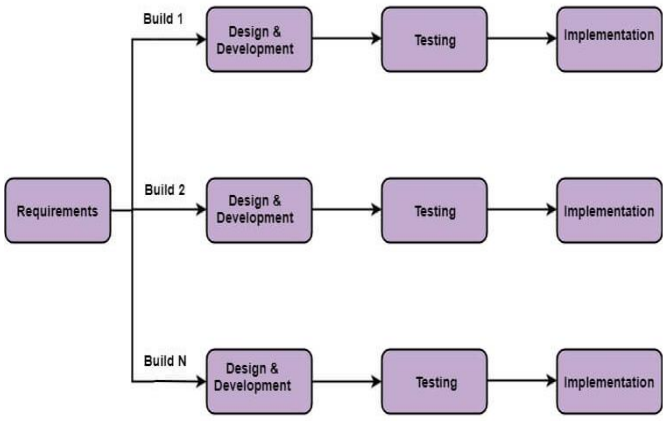


Fig: Incremental Model

The various phases of incremental model are as follows:

1. Requirement analysis: In the first phase of the incremental model, the product analysis expertise identifies the requirements. And the system functional requirements are understood by the requirement analysis team. To develop the software under the incremental model, this phase performs a crucial role.

2. Design & Development: In this phase of the Incremental model of SDLC, the design of the system functionality and the development method are finished with success. When software develops new practicality, the incremental model uses style and development phase.

The various phases of incremental model are as follows:

3. Testing: In the incremental model, the testing phase checks the performance of each existing function as well as additional functionality. In the testing phase, the various methods are used to test the behavior of each task.

4. Implementation: Implementation phase enables the coding phase of the development system. It involves the final coding that design in the designing and development phase and tests the functionality in the testing phase. After completion of this phase, the number of the product working is enhanced and upgraded up to the final system product

When we use the Incremental Model?

When the requirements are superior.

A project has a lengthy development schedule.

When Software team are not very well skilled or trained.

When the customer demands a quick release of the product.

You can develop prioritized requirements first.

Advantage of Incremental Model

Errors are easy to be recognized.

Easier to test and debug

More flexible.

Simple to manage risk because it handled during its iteration.

The Client gets important functionality early.

Negotiating requiremnts

The inception, elicitation, and elaboration tasks in an ideal requirements engineering setting determine customer requirements in sufficient depth to proceed to later software engineering activities. You might have to negotiate with one or more stakeholders. Most of the time, stakeholders are expected to balance functionality, performance, and other product or system attributes against cost and time-to-market.

The goal of this discussion is to create a project plan that meets the objectives of stakeholders while also reflecting the real-world restrictions (e.g., time, personnel, and budget) imposed on the software team. The successful negotiations aim for a "win-win" outcome. That is, stakeholders, benefit from a system or product that meets the majority of their needs, while you benefit from working within realistic and reasonable budgets and schedules.

At the start of each software process iteration, Boehm defines a series of negotiating actions. Rather than defining a single customer communication activity, the following are defined:

Negotiating requiremnts

1.Identifying the major stakeholders in the system or subsystem.

2.Establishing the stakeholders' "win conditions."

3.Negotiation of the win conditions of the stakeholders in order to reconcile them into a set of win-win conditions for all people involved

Validating Requirements

Each aspect of the requirements model is checked for consistency, omissions, and ambiguity as it is developed. The model's requirements are prioritised by stakeholders and bundled into requirements packages that will be implemented as software increments.

The following questions are addressed by an examination of the requirements model:

Is each requirement aligned with the overall system/product objectives?

Were all requirements expressed at the appropriate level of abstraction? Do some criteria, in other words, give a level of technical information that is inappropriate at this stage?

Is the requirement truly necessary, or is it an optional feature that may or may not be critical to the system's goal?
Is each requirement attributed? Is there a source noted for each requirement?
Are there any requirements that conflict with others?
Is each requirement attainable in the technical environment in which the system or product will be housed?
Is each requirement, once implemented, testable?
Does the requirements model accurately represent the information, functionality, and behaviour of the system to be built?
Has the requirements model been "partitioned" in such a way that progressively more detailed information about the system is exposed?
Have requirements patterns been used to reduce the complexity of the requirements model?
Have all patterns been validated properly? Are all patterns in accordance with the requirements of the customers?

QFD

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. Originally developed in Japan and first used at the Kobe Shipyard of Mitsubishi Heavy Industries, Ltd., in the early 1970s, QFD "concentrates on maximizing customer satisfaction from the software engineering process [ZUL92]." 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, specific system functions, and defined levels of performance.

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.. actuality, QFD spans the entire engineering process [AKA90]. However, many QFD concepts are applicable to the requirements elicitation activity. We present an overview of only these concepts (adapted for computer software) in the paragraphs that follow.

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.

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.

Requirement Engineering

Requirements engineering (RE) refers to the process of defining, documenting, and maintaining requirements in the engineering design process. 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. 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.

Requirement Engineering Process

It is a four-step process, which includes -

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

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:

1. **Technical Feasibility** - Technical feasibility evaluates the current technologies, which are needed to accomplish customer requirements within the time and budget.
2. **Operational Feasibility** - Operational feasibility assesses the range in which the required software performs a series of levels to solve business problems and customer requirements.
3. **Economic Feasibility** - Economic feasibility decides whether the necessary software can generate financial profits for an organization.

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

o Getting all, and only, the right people involved.
o Stakeholders often don't know what they want
o Stakeholders express requirements in their terms.
o Stakeholders may have conflicting requirements.

o Requirement change during the analysis process.

o Organizational and political factors may influence system requirements.

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.

o **Data Flow Diagrams:** Data Flow Diagrams (DFDs) are used widely for modeling the requirements. DFD shows the flow of data through a system. The system may be a company, an organization, a set of procedures, a computer hardware system, a software system, or any combination of the preceding. The DFD is also known as a data flow graph or bubble chart.

o **Data Dictionaries:** Data Dictionaries are simply repositories to store information about all data items defined in DFDs. At the requirements stage, the data dictionary should at least define customer data items, to ensure that the customer and developers use the same definition and terminologies.

o **Entity-Relationship Diagrams:** Another tool for requirement specification is the entityrelationship
diagram, often called an "***E-R diagram***." It is a detailed logical
representation of the data for the organization and uses three main constructs i.e. data entities, relationships, and their associated attributes.

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 -

o If they can practically implement

o If they are correct and as per the functionality and specially of software

o If there are any ambiguities

o If they are full

o If they can describe

Requirements Validation Techniques

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

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

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

o **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:

o Clear

o Correct

o Consistent

o Coherent

o Comprehensible

o Modifiable

o Verifiable

o Prioritized

o Unambiguous

o Traceable

o Credible source

Software Requirements: Largely software requirements must be categorized into two categories:

1. **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.
2. **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.

Non-functional requirements are divided into two main categories:

o **Execution qualities** like security and usability, which are observable at run time.

o **Evolution qualities** like testability, maintainability, extensibility, and scalability that embodied in the static structure of the software system.

Analysis and Design Modelling

I. Analysis Modelling

1. Concept and need of Analysis Modelling
2. Objectives of Analysis Modelling

II. Analysis Modelling approaches

1. Structured Analysis (Concept)
2. Object Oriented Analysis (Concept)

III. Domain Analysis

1. Concept of Technical Domain of the software
2. Concept of Application Domain of the Software
3. Inputs and Output of Domain analysis

IV. Building the Analysis Model

1. Data Modelling Concepts
2. Flow- Oriented Modelling

i. DFD

ii. Data Dictionary

iii. Creating a Control Flow Model

iv. Creating Control Specifications (CSPEC)

v. Creating Process Specifications (PSPEC)

3. Scenario- Based Modelling

i. Developing Use Cases

ii. What is a Use Case?

iii. Purpose of a Use Case

iv. Use Case Diagram

I. Analysis Modelling

i. The development process starts with the analysis phase.

ii. This phase results in a specification document that shows what the software will do without specifying how it will be done.

iii. The analysis phase can use two separate approaches, depending on whether the implementation phase is done using a procedural programming language or an object-oriented language.

2. Analysis Principles

i. The information domain must be represented and understood.

ii. Models should be developed to give emphasis on system information, function and behavior.

iii. Models should uncover and give details of all the layers of the development process.

iv. The function and the problem statement must be defined.

v. The various analysis models are flow oriented modelling, scenario based modelling, class based modelling, and behavioral modelling.

3. Need

i. Analysis modelling describes the operational and behavioral characteristics.

ii. Shows the relationship between software interface and other software elements.

iii. Provides the software developer the representation of the information, function, and behavior.

iv. Coverts the design into the more descriptive models like use case, ER diagram.

v. Provides the customer and the developer the means to maintain the quality.

4. Objective

i. Describe what the customer requires.

ii. Establish a basis for the creation of a software design.

iii. Devise a set of requirements that can be validated once the software is built.

iv. Analysis model bridges the gap between system level description and the overall system functionality.

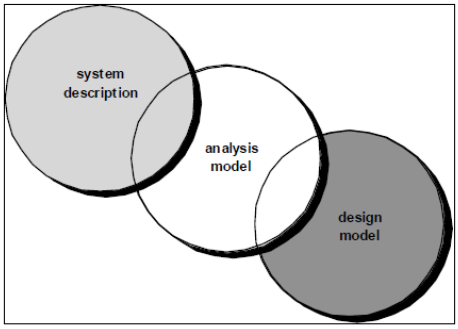


Figure 1. Analysis Model

5. Analysis Rules Of Thumb

i. The model should focus on requirements that are visible within the problem or business domain and be written as a relatively high level of abstraction.

ii. Each element of the analysis model should add to the understanding of the requirements and provide insight into the information domain, function, and behavior of the system.

iii. Delay consideration of infrastructure and other non-functional models until design.

iv. Minimize coupling throughout the system.

v. Be certain the analysis model provides value to all stakeholders.

vi. Keep the model as simple as possible.

vii. The Figure 2 shows the structure of analysis modelling.

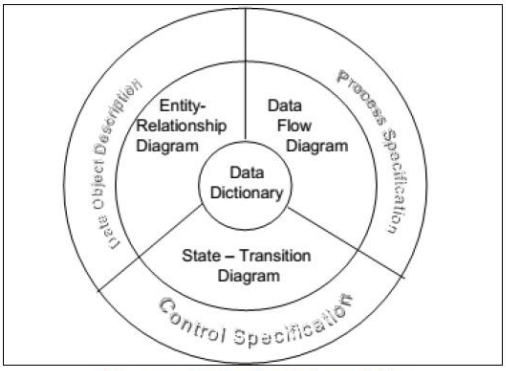


Figure 2.Structure of analysis modelling

II. Analysis Modeling Approaches

i. Considers data and processes that transform data as separate entities.

ii. Structure analysis is a top down approach.

iii. It focuses on refining the problem with the help of the functions performed on the problem domain.

2. Object-oriented analysis

i. Focuses on the definition of classes and the manner in which they collaborate to effect the customer requirements.

ii. Defines the system as a set of objects which interact with each other with the services provided.

iii. Analyses the problem domain and then partitions the problem with the help of objects.

iv. The concept of object, attributes, class, operation, inheritance, and polymorphism should be known to work on object oriented modelling.

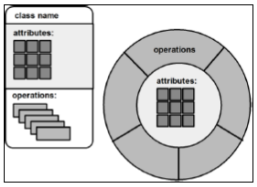


Figure 3: Class Domain

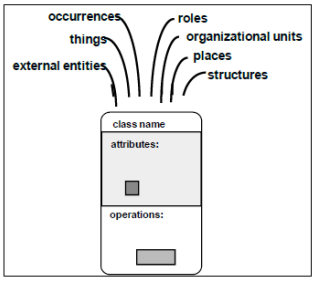


Figure 4: Class, attributes and operations

1. **Domain Analysis**
2. **Definition**
    i. Domain Analysis is the process that identifies the relevant objects of an application domain.
    ii. The goal of Domain Analysis is Software Reuse.

iii. The higher is the level of the life-cycle object to reuse, the larger are the benefits coming from its reuse, and the harder is the definition of a workable process.

3. Concept and technical application domain of the software

i. Frameworks are excellent candidates for Domain Analysis: they are at a higher level than code but average programmers can understand them.

ii. Umbrella activity involving the Identification, analysis, and specification ofcommon requirements from a specific application domain, typically for reuse in multiple projects

iii. Object-oriented domain analysis involves the identification, analysis, and specification of reusable capabilities within a specific application domain in terms of common objects, classes, subassemblies, and frameworks

4. Input and Output Structure of domain analysis

i. Figure 5 shows the flow of the input and the output data in the domain analysis module.

ii. The main goal is to create the analysis classes and common functions.

iii. The input consists of the knowledge domain.

iv. The input is based on the technical survey, customer survey and expert advice.

v. The output domain consists of using the input as the reference and developing the functional models

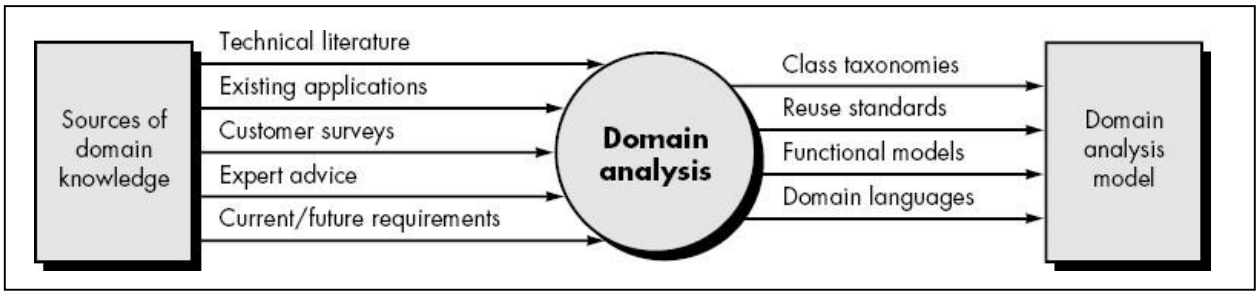


Figure 5: Domain Analysis

II. Building The Analysis Model

1. Data Modelling Concepts

Data modeling is the analysis of data objects that are used in a business orother context and the identification of the relationships among these dataobjects.

i. Data modeling is a first step in doing object-oriented programming.
ii. A data model can be thought of as a diagram or flowchart that illustrates the relationships between data.
iii. Data modelers often use multiple models to view the same data and ensure that all processes, entities, relationships and data flows have been identified.
iv. There are several different approaches to data modeling, including: Conceptual Data Modeling - identifies the highest-level relationships between different entities.
v. Enterprise Data Modeling - similar to conceptual data modeling, but addresses the unique requirements of a specific business.
vi. Logical Data Modeling - illustrates the specific entities, attributes and relationships involved in a business function. Serves as the basis for the creation of the physical data model.
vii. Physical Data Modeling - represents an application and database-specific implementation of a logical data model.
viii. Data objects are modeled to define their attributes and relationships.

a) Data objects (Entities)

i. The Figure 6 the relations between the objects and their attributes.
ii. Data objects are the representation of the most composite information of the system.
iii. Data object description incorporates the data object and all of its attributes.
iv. Data objects are all related to each other.

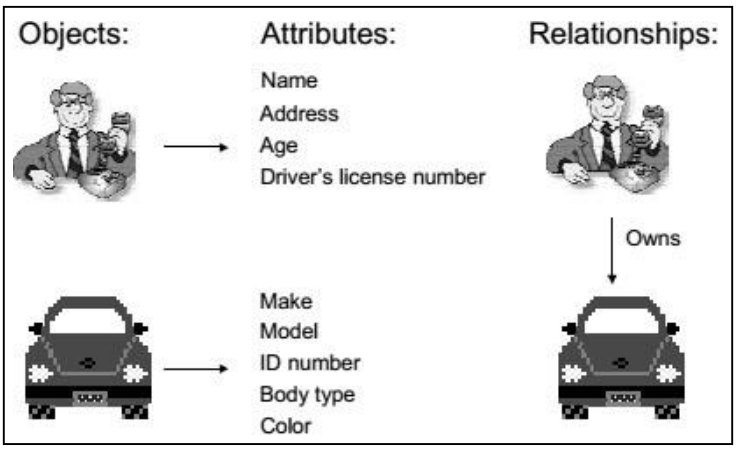


Figure 6: Relation between the object and its attributes

b) Data attributes

i. Attributes define the properties of the data object as shown in Figure 3.
ii. Attributes are used to name the instances of the data.
iii. They describe the instance of the data.
iv. It helps to make reference to the other instance in another table.

c) Relationships

i. Data objects are linked with each other in different ways. These links and connections of data objects are known as relationships.
ii. The two objects person and car are linked with the relationship as person owns the car as shown in Figure 3.

d) Cardinality (number of occurrences)

i. Cardinality is the specification of the number of occurrences of one object that can be related to the number of occurrences of another object
ii. The cardinality is referred to as “one” or “many”, One-to-one (1:1), One-to-many (1: N), Many-to-many (M: N).
iii. When one instance of object A relates to one instance of object B, its one to one cardinality.

e) Modality

i. Modality is 1 if an occurrence of the relationship is mandatory.
ii. Modality is 0 if there is no explicit need for the relationship to occur or the relationship is optional.
iii. Each faculty member advises many students, each student has only one advisor.
iv. Every faculty member may not be advisor, each student must have an advisor

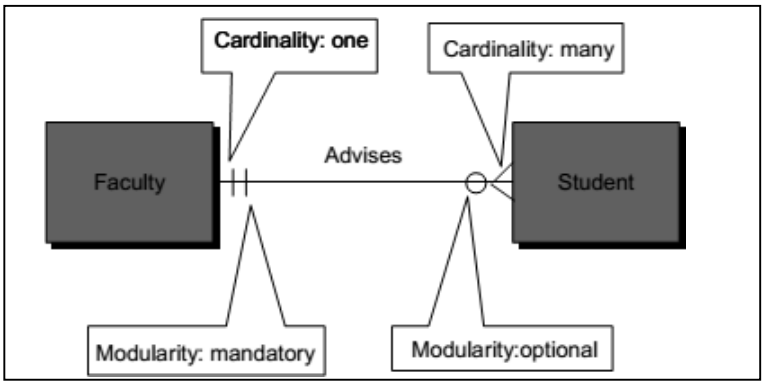

Cardinality: one
Cardinality: many
Faculty
Advises
Student
Modularity: mandatory
Modularity:optional

Figure 7: Cardinality and modularity.

2. Flow oriented modelling

This represents how the data objects are transformed as they move through the system. The flow modelling provides the view of the system in the graphical approach.

1. DFD

i. The Data Flow Diagram is a graphical technique that depicts information flow and the transforms that are applied as data move from input to output.
ii. Can be used at any level of abstraction.
iii. A level 0 DFD, also called a fundamental system model or context diagrams represents the entire software system as a single bubble with input and output data indicated by incoming and outgoing arrows respectively
iv. A level 1 DFD might contain five or six bubbles with interconnecting arrows; each of the processes represented at level 1 are sub functions of the overall system depicted in the context model.
v. Figure 8 depicts the symbols used.
vi. Depicts how input is transformed into output as data objects move through a system.
vii. Functional modeling and information flow Indicates how data are transformed as they move through the system
viii. Depicts the functions that transform the data flow.
ix. Each function description is contained in a process specification.

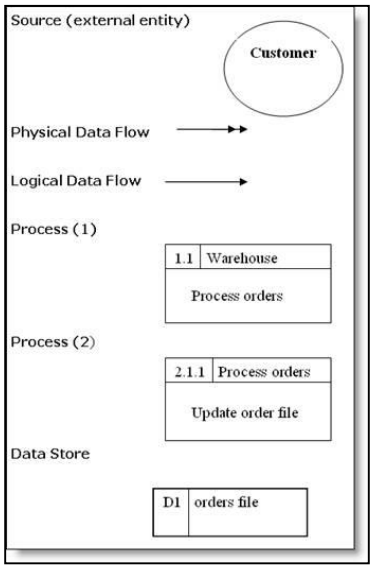


Figure 8: Symbols of DFD

Level 0 DFD of a banking system

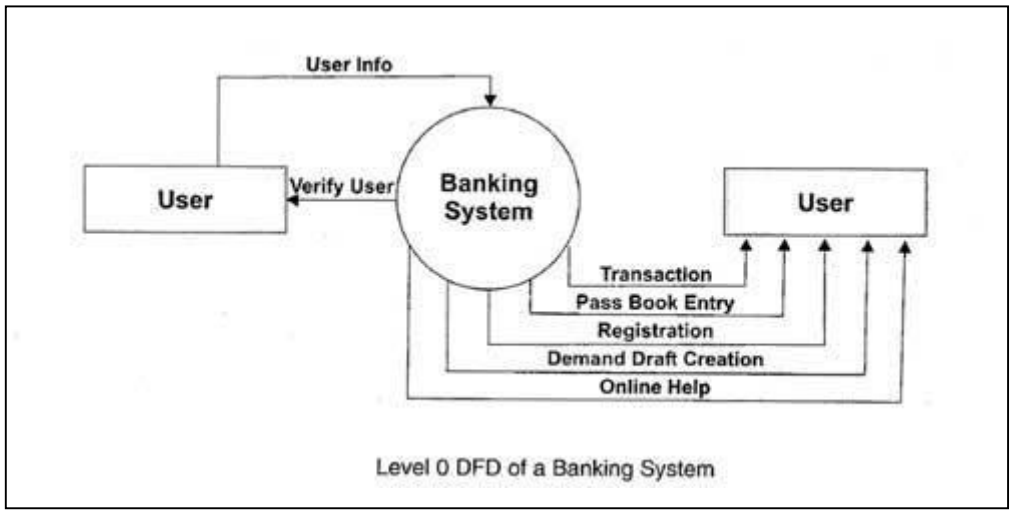


Figure 9: Banking System

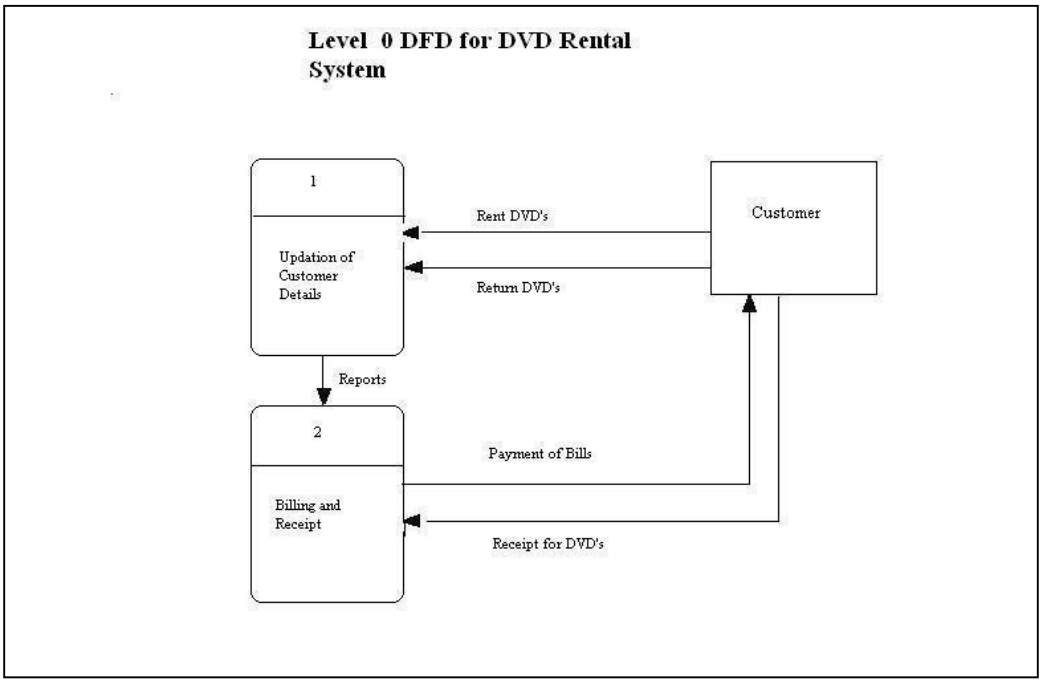


Figure 10: DVD Rental System

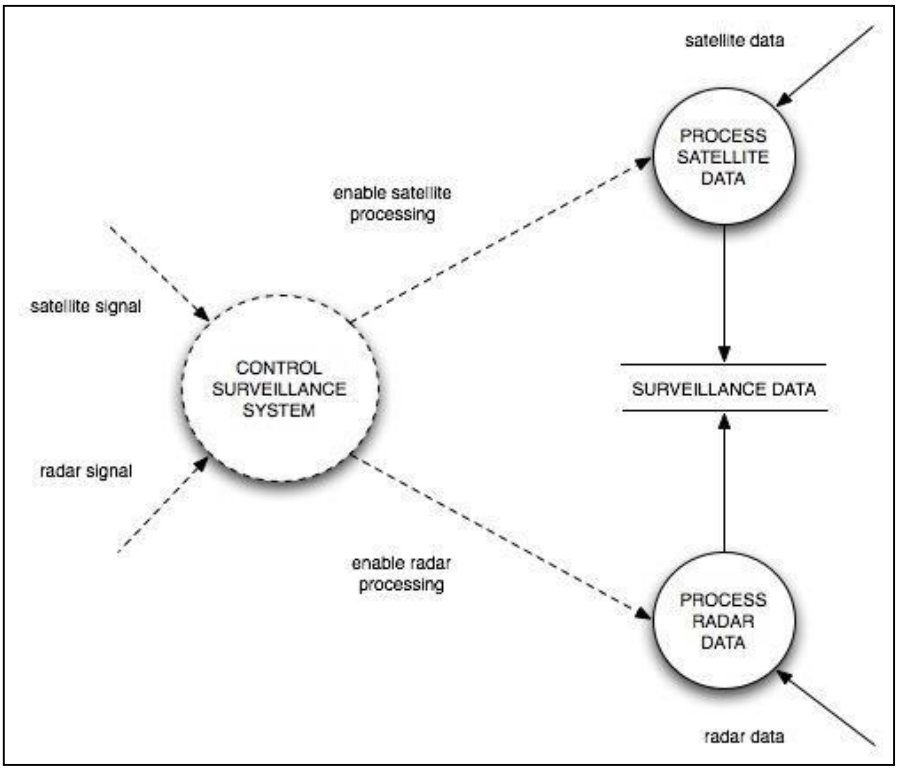


Figure 11: Control Surveillance System

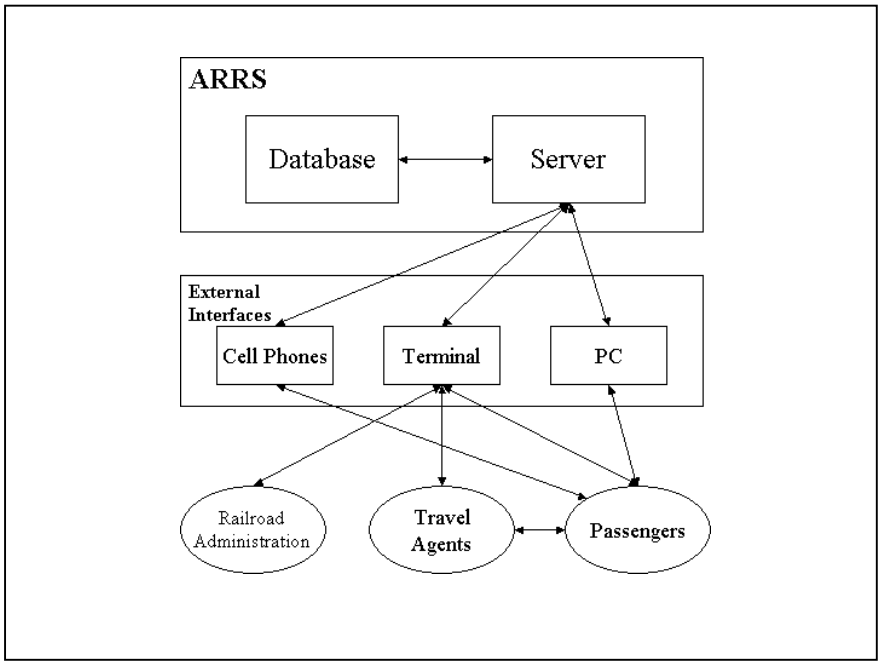


Figure 12: Level 0 DFD of Railway Reservation system

Rule 1 - Does each function have input and output?

1. **Rule 2** - Does each function have all the information it needs in order to produce its output?
2. **Rule 3** - If not, then what information does it need and where will it get that information from?

LEVEL 1 DFD

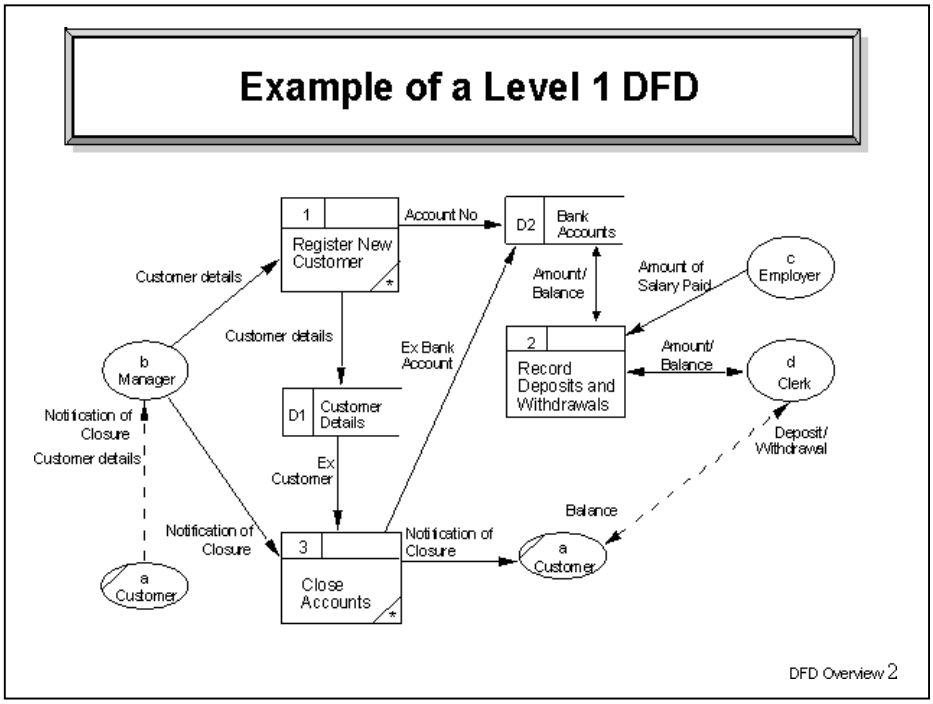


Figure 13: Banking System

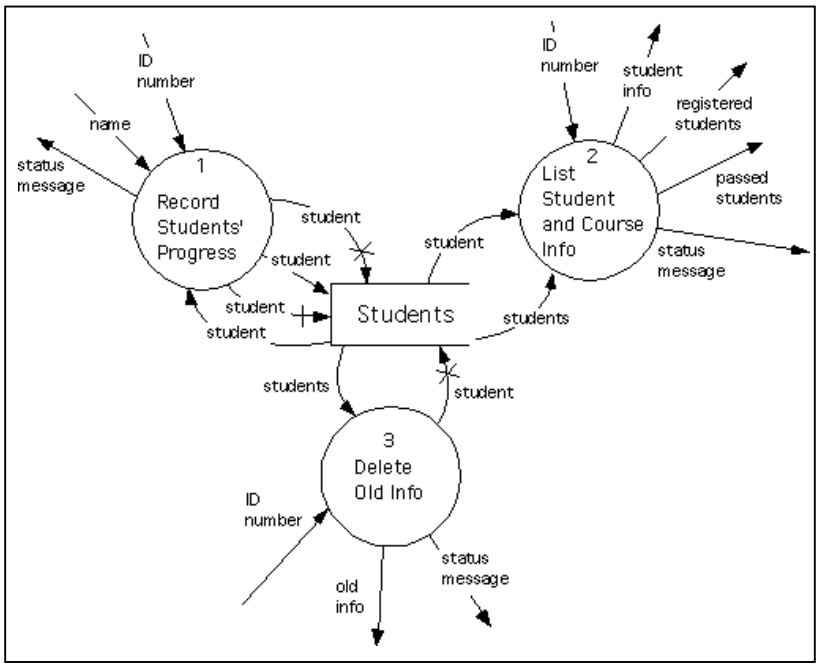


Figure 14: Student Record system

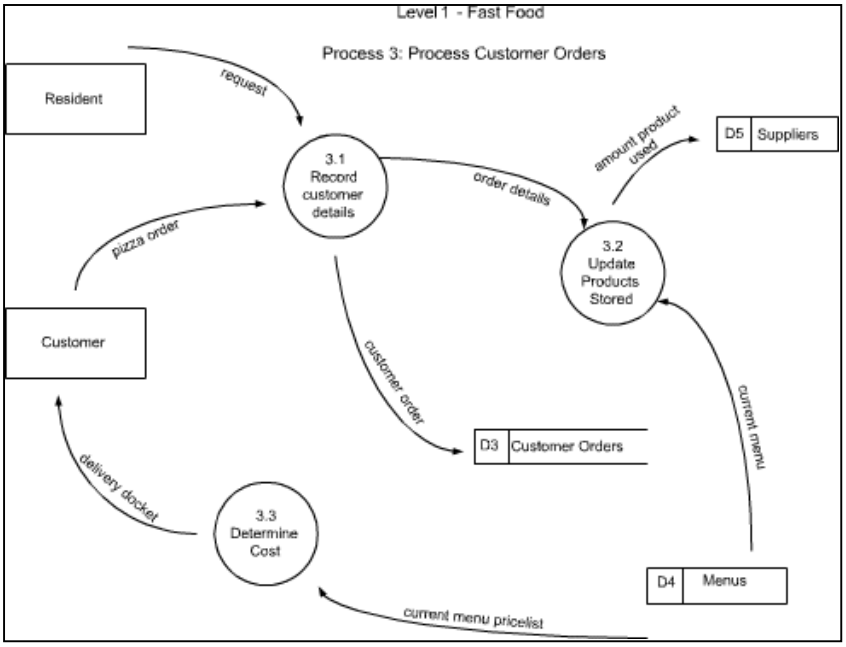


Figure 15: Fast Food System

Level 2 DFD

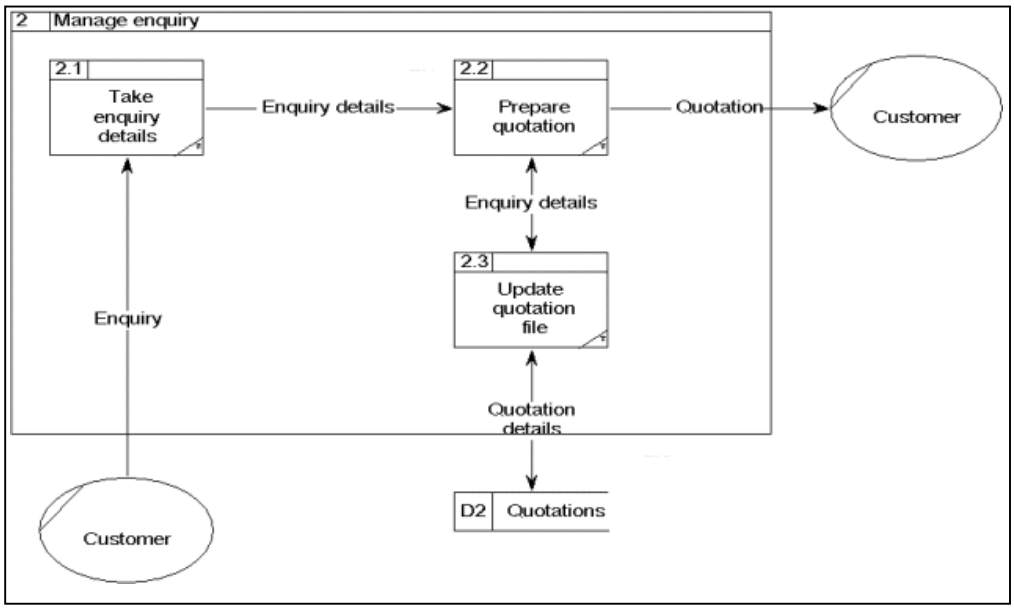
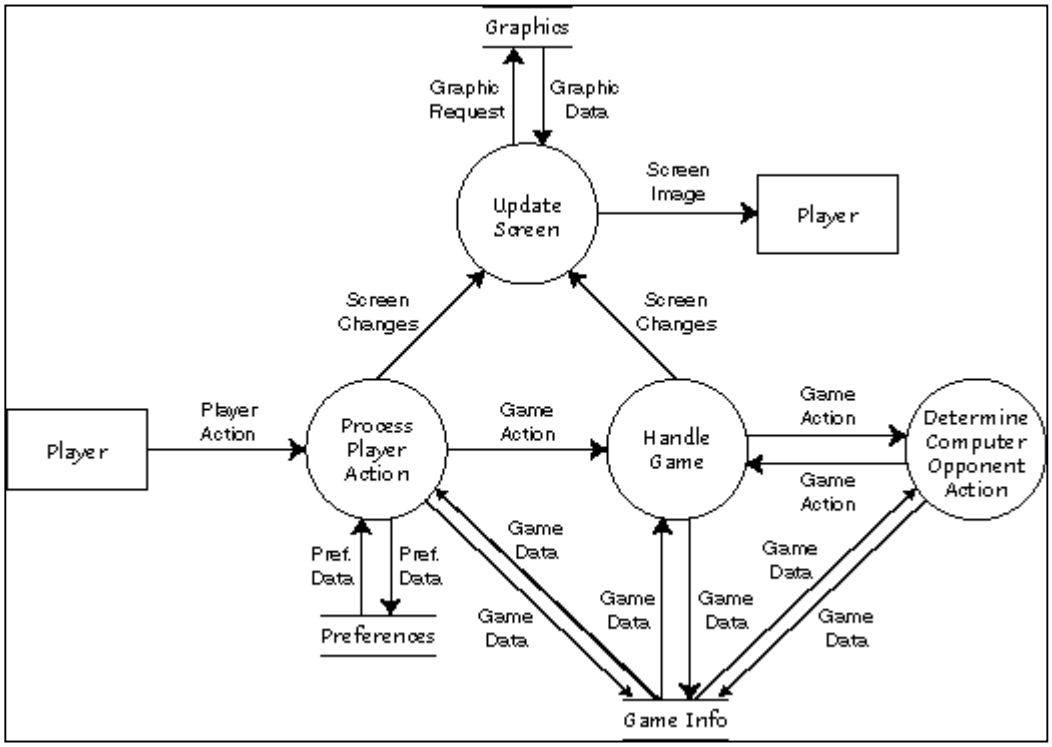


Figure 16: Tender System



Logical DFD

Figure 17: Logical DFD for computer game development

The difference between a logical and a physical data flow diagram, typically referred to as a DFD, lies primarily in how the data is identified and represented.

i. A data flow diagram in general represents the movement of data within an organization, concentrating on its information system.

ii. **A logical DFD** focuses more on the organization itself and identifies the data-generating events that take place.

iii. **A physical DFD** instead is concerned with how that data is represented.

iv. Both types of DFDs are valuable tools for allowing users to monitor the flow of information from its entry point to its movement throughout the organization, and eventually to its exit point. Interpretation of the data along the way relies partially on recognizing whether the information is processed sequentially or in a parallel fashion.

v. The benefits of a logical DFD include easy communication between employees, the potential for more stable systems, better understanding of the data and the system by analysts, and an overall flexibility.

vi. It is also easy to maintain and to remove redundancies as they accumulate.

vii. A physical DFD, on the other hand, has a clear distinction between manual and automated processes, provides more controls over the system, and identifies temporary data stores.

2. Data Dictionary

i. Data dictionary is a set of meta-data which contains the definition and representation of data elements.

ii. It gives a single point of reference of data repository of an organization. Data dictionary lists all data elements but does not say anything about relationships between data elements.

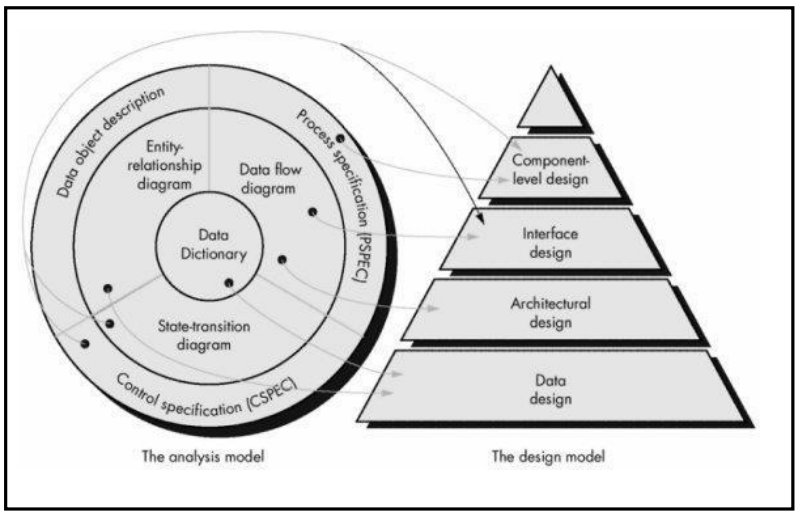
iii. A data dictionary or database dictionary is a file that defines the basic organization of a database.

iv. A database dictionary contains a list of all files in the database, the number of records in each file, and the names and types of each data field.

v. Most database management systems keep the data dictionary hidden from users to prevent them from accidentally destroying its contents.

vi. Data dictionaries do not contain any actual data from the database, only bookkeeping information for managing it.

vii. Without a data dictionary, however, a database management system cannot



access data from the database.

Figure 18: Data Dictionary

3. Creating Control Flow Model

i. Illustrates how events affect the behavior of a system through the use of state diagrams.

ii. Data flow and the control flow diagrams are necessary to obtain the meaningful insight of the software requirements.

iii. There are large class of events that are driven by events rather than data.

iv. Such applications that are driven by events require control flow model.

4. Creating Process Specifications

i. Six class selection characteristics that retain information.

ii. Information must be remembered about the system over time.

iii. Needed services

iv. Set of operations that can change the attributes of a class.

v. Multiple attributes: Whereas, a single attribute may denote an atomic variable rather than a class.

vi. Common attributes: A set of attributes apply to all instances of a class

vii. Common operations: A set of operations apply to all instances of a class

viii. Essential requirements: Entities that produce or consume information.

3. Scenario Based modelling

1. Use case

i. "[Use-cases] are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use-cases)." Ivar Jacobson

ii. The concept is relatively easy to understand- describe a specific usage scenario in straightforward language from the point of view of a defined actor.

2. Writing Use-Cases

i. What should we write about?

ii. Inception and elicitation provide us the information we need to begin writing use cases.

iii. How much should we write about it?

iv. How detailed should we make our description?

v. How should we organize the description?

3. Developing an Activity Diagram

i. What are the main tasks or functions that are performed by the actor?

ii. What system information will the actor acquire, produce or change?

iii. Will the actor have to inform the system about changes in the external environment?

iv. What information does the actor desire from the system?

v. Does the actor wish to be informed about unexpected changes?

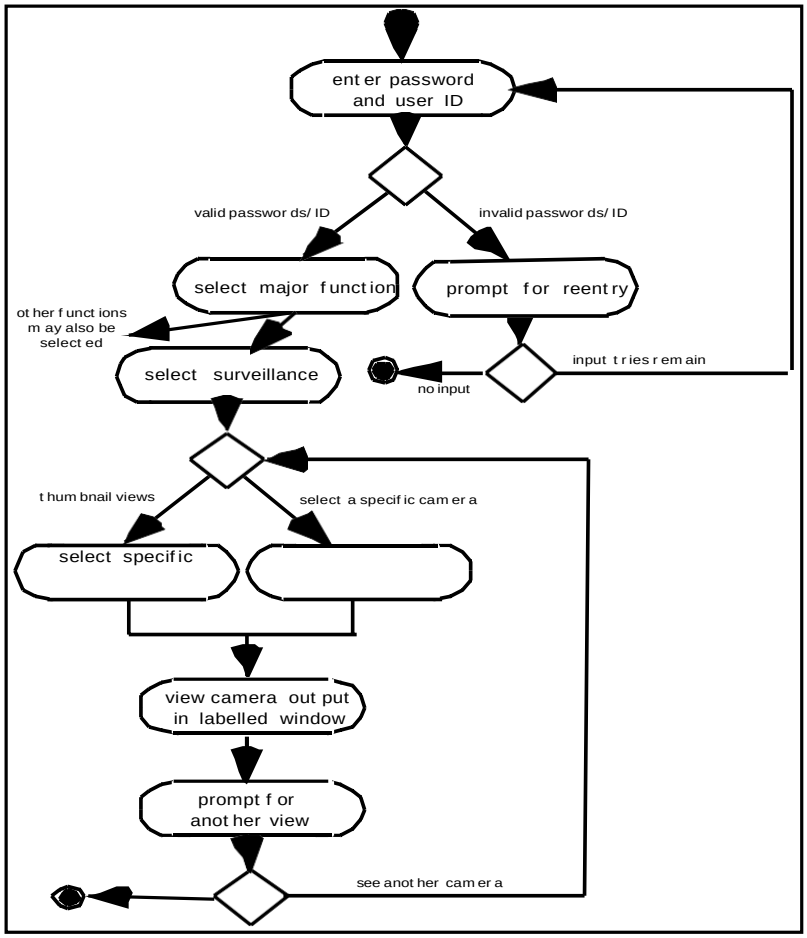


Figure 19: Activity Diagram

4. Swim lane Diagrams

i. The UML swim lane diagram is a useful variation of the activity diagram and allows the modeler to represent the flow of activities described by theuser-case and at the same time indicate which actor or analysis class has responsibility for the action described by an activity rectangle.

ii. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

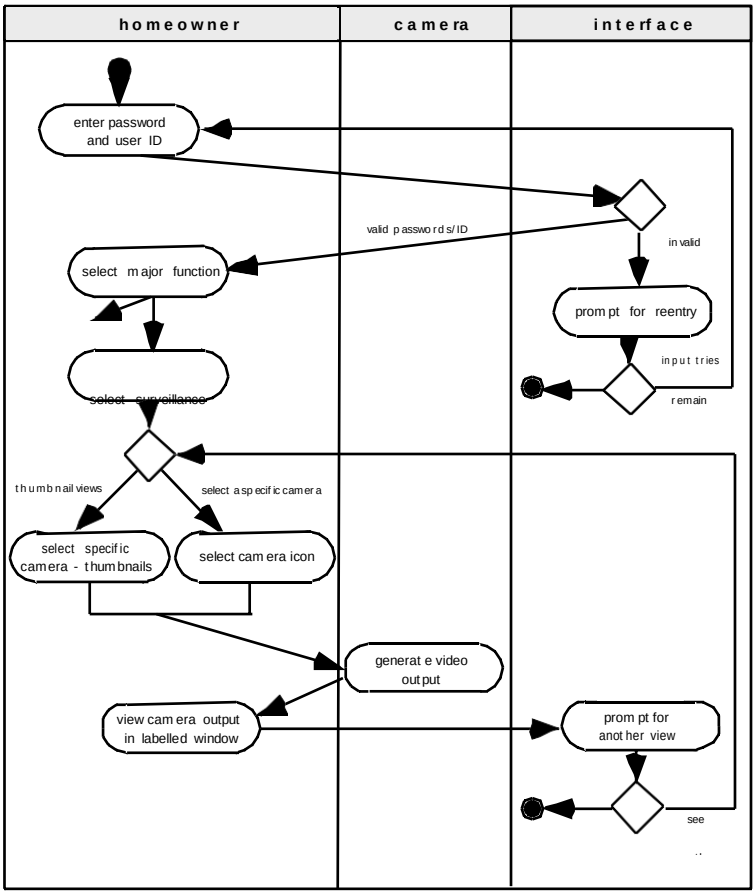


Figure 20: Swim lane Diagram

Safe Home security system

SafeHome security system Scope of the software: The main target of this software namedSafeHome security system is to create a system that works in a house .It will have some functionalities like Home security functions, home surveillance function ,home management function, communication management function etc. so for the rest of the discussion we shall maintain the software engineering processes and develop this software in a organized manner. Before we begin the process of software engineering we have to know about some processes that are used in software engineering. Requirements analysis: Requirements analysis results in the specification of software's operational characteristics. Indicates software' sinter face with other system elements, and establishes constraints that software must meet. The analysis model must achieve some primary objectives. The requirement analysis allows the software engineer to elaborate on basic requirements established during early requirement engineering tasks and build model that depict user scenarios, functional activities, problem classes and the irrelation ships, system and class behaviour and the flow of data as it is transformed. Requirements analysis provides the software designer with a representation of information, function and behaviour that can be translated to architectural, interface and component level designs .Finally the analysis model and the requirements specification provide the developer and customer with the means to asses quality Once the software is built. The analysis model must achieve three basic objectives.

1) To describe what the customer requires.

2) To establish a basis for the creation of a software design.3)To define a set of requirements that can be validated once the software is built. The analysis model bridges the gap between a system level description that describes overall system functionality as it is achieved by software ,hardware, data ,human and other system elements and a software design that describes the software's application architecture ,user interface and component level structure.

Agile overhauled requirements gathering. Under the Waterfall model, development teams gather software requirements before coding or testing. But Agile radically changes the rules for software requirements.

Agile requirements gathering is a practice teams often perform on the fly. For example, developers update requirements between iterations -- if the software project has documented requirements at all. Some Agile practice purists balk at the word *requirements*. A lack of requirements could throw many business or technical processes into chaos, but Agile development thrives in an iterative approach. Agile trades certainty for adaptability.

Let's look at the goals of Agile software development and practices that help requirements gathering keep pace.

Why requirements gathering matters

Requirements set project goals and guide developers through coding and testing. Wrong or incomplete requirements can create project delays, cost overruns and poor user acceptance or adoption -- even project failure.

What makes Agile requirements gathering unique? Traditional Waterfall projects emphasize upfront gathering, review and approval of detailed requirements. Approved requirements are unlikely to change. Change requests are often costly and disruptive. This requirements paradigm simply isn't flexible enough to work in Agile development environments.

Agile and other forms of continuous development depend on fast, iterative, customer-focused processes. The scope still defines the project's purpose, but these secluded requirements teams must rethink how they approach their work. They must determine what users actually want in a software product.

Developers should embrace a fast, flexible and dynamic approach to software creation. They can experiment, try new things and evolve a product that is more competitive and higher quality than one created with a gated development approach.

What does the Agile Manifesto say? The Agile Manifesto highlights four fundamental differences between Agile and Waterfall development:

- Individuals and interactions over processes and tools;
- Working software over comprehensive documentation;
- Customer collaboration over contract negotiation; and
- Responding to change over following a plan.

These four values mean Agile development -- and, thus, Agile requirements gathering -- supports the evolution of a final product that best addresses user needs and business goals.

What role do user stories play? The user story represents requirements in a simple written narrative told from the user's perspective, rather than a comprehensive document.

User stories enable developers to estimate and plan work. User stories change and mature throughout the software development lifecycle.

Developers compile the many user stories into a product backlog. The product manager or product owner manages this backlog.

7 ways to improve Agile requirements gathering

While Agile is a powerful development paradigm, some practitioners find Agile requirements gathering chaotic. Try these ways to bolster requirements gathering

1. Supplement user stories. User stories don't always include enough information for development decisions. Complex or mission-critical projects especially demand more detail.

Some projects must have documentation to meet compliance or process standards. In these cases, development teams can supplement user stories with relevant details, such as use cases and decision tables.

2. Involve stakeholders regularly. Developers can face enormous challenges when they're isolated from project stakeholders, such as users and product owners. Too often, stakeholders get involved early, but don't see the results of the work until late in the project.

Projects can fail when stakeholders and developers fail to communicate. Stakeholders are the experts in a project's requirements. They should share ideas, make suggestions and model and document requirements in each iteration. Developers can then build and adjust the product accordingly. Project and product owners generally prioritize those requirements for developers.

Reconsider the project if the development team cannot identify appropriate stakeholders, or if those stakeholders don't take an active role.

3. Prioritize requirements. Once a requirement is outlined, developers estimate the time and money necessary to implement the feature or functionality. The list of requirements and estimates for the iteration can take shape as a stack of index cards, or some other method. Due to time and budget limitations, not all requirements make it into a given iteration.

A project or product owner works with developers to prioritize the list of requirements. They sort the stack into high- or low-priority work items for the available time and budget. Generally, the higher priority the requirement, the more detailed the item should be. Development teams often shelve low-priority requirements for future iterations, though it is possible to add or remove items from the list at any time.

4. Focus on executable requirements. Agile teams typically model requirements, write code, and then refine and refactor it to implement those models. This process is called test-first design. Modeling translates requirements into code. Executable requirements focus on what something needs to do, and how that thing should work. When developers work through executable models upfront, they can identify cross-requirement dependencies. Developers can also spot potential problems via highly focused approaches, such as test-driven development

5. Use the INVEST principle. User stories are easy to create. Good user stories are harder to create. To write valuable user stories, Agile teams often use the INVEST mnemonic:

- Independent: Each user story should stand on its own, independent from other stories.
- Negotiable: Stories are not contracts, rather opportunities for negotiation and change.
- Valuable: Every story should add value for users and stakeholders.
- Estimable: Every story's time and budget costs should be calculable, based on domain and technical knowledge.
- Small: User stories should be small enough to estimate and implement simply.
- Testable: Make sure you can test the user story through criteria the story itself explains.

6. Think layers, not slices. With Agile requirements, it's helpful to think small. Smaller requirements and feature sets correspond to user stories that are easier to estimate, prioritize and build. Larger, more complex requirements tend to create dependencies, going against the

INVEST principle.Compose user stories with an eye toward the software stack, such as the front end, middleware, back end and database.

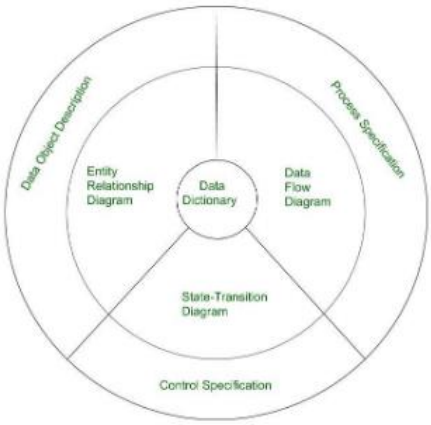
7. Prototype and update. Agile development paradigms facilitate developer experimentation, while mitigating risk through tests. Prototyping is a useful practice to test ideas and encourage discussion with stakeholders. Developers can update the prototype to refine the software and solidify its design. The resulting code often is usable for the build.

Analysis Model is a technical representation of the system. It acts as a link between system description and design model. In Analysis Modelling, information, behavior, and functions of the system are defined and translated into the architecture, component, and interface level design in the design modeling.

Objectives of Analysis Modelling:

- It must establish a way of creating software design.
- It must describe the requirements of the customer.
- It must define a set of requirements that can be validated, once the software is built.

Elements of Analysis Model:



- **Data Dictionary:**

  It is a repository that consists of a description of all data objects used or produced by the software. It stores the collection of data present in the software. It is a very crucial element of the analysis model. It acts as a centralized repository and also helps in modeling data objects defined during software requirements.

- **Entity Relationship Diagram (ERD):**

  It depicts the relationship between data objects and is used in conducting data modeling activities. The attributes of each object in the Entity-Relationship Diagram can be described using Data object description. It provides the basis for activity related to data design.

- **Data Flow Diagram (DFD):**

  It depicts the functions that transform data flow and it also shows how data is transformed when moving from input to output. It provides the additional information which is used during the analysis of the information domain and serves as a basis for the modeling of function. It also enables the engineer to develop models of functional and information domains at the same time.

- **State Transition Diagram:**

  It shows various modes of behavior (states) of the system and also shows the transitions from one state to another state in the system. It also provides the details of how the system behaves due to the consequences of external events. It represents the behavior of a system by presenting its states and the events that cause the system to change state. It also describes what actions are taken due to the occurrence of a particular event.

- **Process Specification:**

  It stores the description of each function present in the data flow diagram. It describes the input to a function, the algorithm that is applied for the transformation of input, and the output that is produced. It also shows regulations and barriers imposed on the performance characteristics that are applicable to the process and layout constraints that could influence the way in which the process will be implemented.

- **Control Specification:**

  It stores additional information about the control aspects of the software. It is used to indicate how the software behaves when an event occurs and which processes are invoked due to the occurrence of the event. It also provides the details of the processes which are executed to manage events.

- **Data Object Description:**

  It stores and provides complete knowledge about a data object present and used in the software. It also gives us the details of attributes of the data object present in the Entity Relationship Diagram. Hence, it incorporates all the data objects and their attributes.