ACT

College of Computer Studies

COURSE NOTES iN

APPDEV

Application Development

Course Description:

A course in application programming with an emphasis on software development. Integrated development tools, software development kits, and software subsystems are to be employed to develop database, eCommerce, and mobile applications. Learning activities in this course include classroom, laboratory, and online tasks to develop the knowledge and skills necessary to write effective computer programs for information system applications.

Learning Outcomes:

- Learn on how to interpret a data
- Learn different ways on how to manage a data
- Describe the requirements analysis process and techniques
- Use the correct usability testing method for a software development
- Distinguish a proper deployment plan based on real-life scenarios.

Tools or Application to Use:

- Student Achievement Monitoring System (SAMS)
- Facebook Messenger
- Google Classroom
- Google Meet

Mode of Assessment:

- Online Quiz
- Activities
- Project
- Presentation

1 WHAT IS A SOFTWARE?

Software is capable of performing many tasks, as opposed to hardware which can only perform mechanical tasks that they are designed for.
The software provides the means for accomplishing many different tasks with the same basic hardware.

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries, and documentation. Software, when made for a specific requirement is called **software product**.

Classes of Software

- **System software** – helps run the computer hardware and computer system itself. System software includes operating systems, device drivers, diagnostic tools, and more. System software is almost always pre-installed on your computer.

- **Application software** – allows users to accomplish one or more tasks. It includes word processing, web browsing, and almost any other task for which you might install the software. (Some application software is pre-installed on most computer systems.)

- **Programming software** – a set of tools to aid developers in writing programs. The various tools available are compilers, linkers, debuggers, interpreters, and text editors.

Basic Principles

1. Software, commonly known as programs or apps, consists of all the instructions that tell the hardware how to perform a task.
2. These instructions come from a software developer in the form that will be accepted by the platform (operating system + CPU) that they are based on.
3. For example, a program that is designed for the Windows operating system will only work for that specific operating system. The compatibility of the software will vary as the design of the software and the operating system differ. Software that is designed for Windows 10 may experience a compatibility issue when running under Windows 11.
4. Software, in its most general sense, is a set of instructions or programs instructing a computer to do specific tasks. Software is a generic term used to describe computer programs, Scripts, applications, programs and a set of instructions are all terms often used to describe software.

Software Evolution

The process of developing a software product using software engineering principles and methods is referred to as software evolution. This includes the initial development of software and its maintenance and updates, till desired software product is developed, which satisfies the expected requirements.
Evolution starts from the requirement gathering process. After which developers create a prototype of the intended software and show it to the users to get their feedback at the early stage of product development. The users suggest changes, on which several consecutive updates and maintenance keep on changing too. This process changes to the original software, till the desired software is accomplished. Even after the user has desired software in hand, the advancing technology and the changing requirements force the software product to change accordingly. Re-creating software from scratch and going one-on-one with requirements is not feasible. The only feasible and economical solution is to update the existing software so that it matches the latest requirements.

Laws in Software Evolution

Eight laws for software evolution

- **Continuing change** – a software system must continue to adapt to the real world changes, else it becomes progressively less useful.

- **Increasing complexity** – a software system evolves, its complexity tends to increase unless work is done to maintain or reduce it.

- **Conversation of familiarity** – the familiarity with the software or the knowledge about how it was developed, why was it developed in that particular manner etc. must be retained at any cost, to implement the changes in the system.

- **Continuing growth** –for a system intended to resolve some business problem, its size of implementing the changes grows according to the lifestyle changes of the business.

- **Reducing quality** – a software system declines in quality unless rigorously maintained and adapted to a changing operational environment.

- **Feedback systems** – the software systems constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved.

- **Self-regulation** – a system evolution processes are self-regulating with the distribution of product and process measures close to normal.

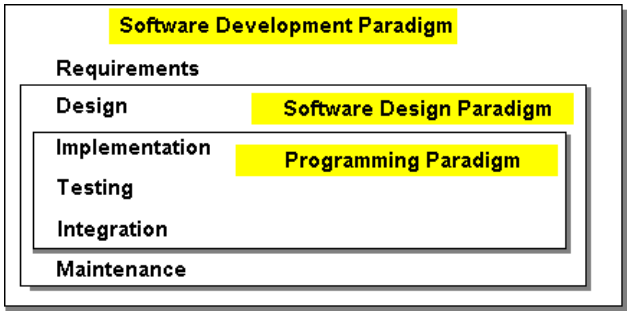
- **Organizational stability** – the average effective global activity rate in an evolving system is invariant over the lifetime of the product.

2 SOFTWARE PARADIGMS

Paradigm

"Paradigm" (a Greek word meaning example) is commonly used to refer to a category of entities that share a common characteristic.
We can distinguish between three different kinds of Software Paradigms:

- Programming Paradigm is a model of how programmers communicate a calculation to computers
- Software Design Paradigm is a model for implementing a group of applications sharing common properties
- Software Development Paradigm is often referred to as Software Engineering, may be seen as a management model for implementing big software projects using engineering principles.



Programming Paradigm

A Programming Paradigm is a model for a class of Programming Languages that share a set of common characteristics.
A programming language is a system of signs used to communicate a task/algorithm to a computer, causing the task to be performed. The task to be performed is called a computation, which follows precise and unambiguous rules.
At the heart of it all is a fundamental question: What does it mean to understand a programming language? What do we need to know to program in a language? There are three crucial components to any language.
The language paradigm is a general principle that is used by a programmer to communicate a task/algorithm to a computer.
The syntax of the language is a way of specifying what is legal in the phrase structure of the language; knowing the syntax is analogous to knowing how to spell and form sentences in a natural language like English. However, this doesn't tell us anything about what the sentences mean.
The third component is the semantics, or meaning, of a program in that language. Ultimately, without semantics, a programming language is just a collection of meaningless phrases; hence, semantics is a crucial part of a language.
There have been a large number of programming languages. Back in the '60s, there were over 700 of the – most were academic, special purpose, or developed by an organization for their own needs.

Fortunately, there are just four major programming language paradigms:

- Imperative (Procedural) Paradigm (Fortran, C, Ada, etc.)
- Object-Oriented Paradigm (Smalltalk, Java, C++)
- Logic Paradigm (Prolog)
- Functional Paradigm (Lisp, ML, Haskell)

Generally, a selected Programming Paradigm defines the main property of software developed utilizing a programming language supporting the paradigm.

- scalability/modifiability
- integrability/reusability
- portability
- performance
- reliability
- ease of creation

Software Design Paradigm

Software Design Paradigm embodies the results of people's ideas on how to construct programs, combine them into large software systems, and formal mechanisms for how those ideas should be expressed.
Thus, we can say that a Software Design Paradigm is a model for a class of problems that share a set of common characteristics.

Software design paradigms can be sub-divided as:

- Design Patterns
- Components
- Software Architecture
- Frameworks

It should be especially noted that a particular Programming Paradigm essentially defines software design paradigms. For example, we can speak about Object-Oriented design patterns, procedural components (modules), functional software architecture, etc.

Design Patterns
A design pattern is a proven solution for a general design problem. It consists of communicating 'objects' that are customized to solve the problem in a particular context.
Patterns have their origin in object-oriented programming where they began as collections of objects organized to solve a problem. There isn't any fundamental relationship between patterns and objects; it just happens they began there. Patterns may have arisen because objects seem so elemental, but the problems we were trying to solve with them were so complex.

- Architectural Patterns: An architectural pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- Design Patterns: A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly within a particular context.
- Idioms: An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Components

Often equated to design patterns with Emphasis on reusability
Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning program.
A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces...typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations
A component must be compatible and interoperate with a whole range of other components.
Examples of components: “Window”, “Push Button”, “Text Editor”, etc.
Two main issues arise concerning interoperability information:

1. how to express interoperability information (e.g. how to add a “push button” to a “window”;
2. how to publish this information (e.g. library with API reusable via an “include” statement)?

Software Architecture

Software architecture is the structure of the components of the solution. A particular software architecture decomposes a problem into smaller pieces and attempts to find a solution (Component) for each piece. We can also say that architecture defines a software system component, its integration, and interoperability:

- Integration means the pieces fit together well.
- Interoperation means that they work together effectively to produce an answer.

There are many software architectures. Choosing the right one can be a difficult problem in itself.

Frameworks

A software framework is a reusable mini-architecture that provides the generic structure and behavior for a family of software abstractions, along with a context of metaphors that specifies their collaboration and use within a given domain.
Frameworks can be seen as an intermediate level between components and software architecture.
Example: Suppose an architecture of a WBT system reuse such components as “Text Editing Input object” and “Pushbuttons”. A software framework may define an “HTML Editor” which can be further reused for building the architecture.

Software Engineering and Software Paradigms

The term "software engineering" was coined in about 1969 to mean "the establishment and use of sound engineering principles to economically obtain software that is reliable and works efficiently on real machines".
This view opposed the uniqueness and "magic" of programming to move the development of software from "magic" (which only a select few can do) to "art" (which the talented can do) to "science" (which supposedly anyone can do!).
There have been numerous definitions given for software engineering (including those above and below).
Software Engineering is not a discipline; it is an aspiration, as yet unachieved. Many approaches have been proposed including reusable components, formal methods, structured methods, and architectural studies. These approaches chiefly emphasize the engineering product; the solution rather than the problem it solves.
Software Development current situation:

- People developing systems were consistently wrong in their estimates of time, effort, and costs
- Reliability and maintainability were difficult to achieve
- Delivered systems frequently did not work
    - 1979 study of a small number of government projects showed that:

    - 2% worked
    - 3% could work after some corrections
    - 45% delivered but never successfully used
    - 20% used but extensively reworked or abandoned
    - 30% paid and undelivered
- Fixing bugs in delivered software produced more bugs
- Increase in size of software systems
    - NASA
    - StarWars Defense Initiative
    - Social Security Administration
    - financial transaction systems
- Changes in the ratio of hardware to software costs
    - early 60's - 80% hardware costs
    - middle 60's - 40-50% software costs
    - today - less than 20% hardware costs
- The increasingly important role of maintenance
    - Fixing errors, modification, adding options
    - Cost is often twice that of developing the software
- Advances in hardware (lower costs)
- Advances in software techniques (e.g., users interaction)
- Increased demands for software
    - Medicine, Manufacturing, Entertainment, Publishing
- Demand for larger and more complex software systems
    - Airplanes (crashes), NASA (aborted space shuttle launches),
    - "ghost" trains, runaway missiles,
    - ATMs (have you had your card "swallowed"?), life-support systems, car systems, etc.
    - US National security and day-to-day operations are highly dependent on computerized systems.

Manufacturing software can be characterized by a series of steps ranging from concept exploration to final retirement; this series of steps is generally referred to as a software lifecycle.
Steps or phases in a software lifecycle fall generally into these categories:

- Requirements (Relative Cost 2%)
- Specification (analysis) (Relative Cost 5%)
- Design (Relative Cost 6%)
- Implementation (Relative Cost 5%)
- Testing (Relative Cost 7%)
- Integration (Relative Cost 8%)
- Maintenance (Relative Cost 67%)
- Retirement

Software engineering employs a variety of methods, tools, and paradigms.
Paradigms refer to particular approaches or philosophies for designing, building, and maintaining software. Different paradigms each have their advantages and disadvantages which make one more appropriate in a given situation than perhaps another (!).
A method (also referred to as a technique) is heavily dependent on a selected paradigm and may be seen as a procedure for producing some result. Methods generally involve some formal notations and processes.
Tools are automated systems implementing a particular method.

Thus, the following phases are heavily affected by selected software paradigms

- Design
- Implementation
- Integration
- Maintenance

The software development cycle involves the activities in the production of a software system. Generally, the software development cycle can be divided into the following phases:

- Requirements analysis and specification
- Design
    - Preliminary design
    - Detailed design
- Implementation
    - Component Implementation
    - Component Integration
    - System Documenting
- Testing
    - Unit testing
    - Integration testing
    - System testing
- Installation and Acceptance Testing
- Maintenance
    - Bug Reporting and Fixing
    - Change requirements and software upgrading
- Software lifecycles that will be briefly reviewed include:
    - Build and Fix model
    - Waterfall and Modified Waterfall models
    - Rapid Prototyping
    - Boehm's spiral model

Build and Fix model

This works OK for small, simple systems, but is completely unsatisfactory for software systems of any size. It has been shown empirically that the cost of changing a software product is relatively small if the change is made at the requirements or design phases but grows large at later phases.
The cost of this process model is far greater than the cost of a properly specified and designed project. Maintenance can also be problematic in a software system developed under this scenario.

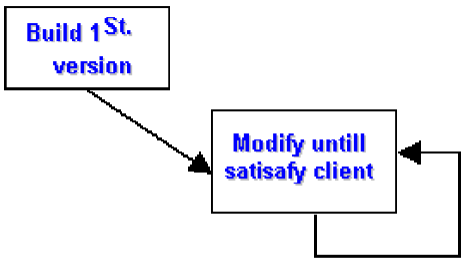


Figure: Build and Fix model

Waterfall and Modified Waterfall models

Waterfall Model

Derived from other engineering processes in 1970. Offered a means of making the development process more structured. Expresses the interaction between subsequent phases.

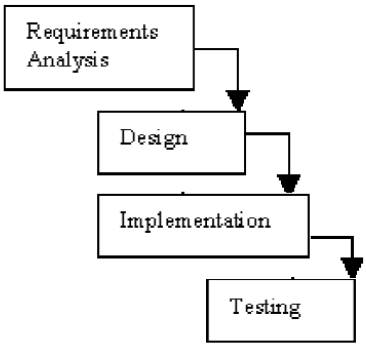


Figure: Waterfall model

Each phase cascades into the next phase. In the original waterfall model, a strict sequentially was at least implied. This meant that one phase had to be completed before the next phase was begun.

It also did not provide feedback between phases or for updating/re-definition of earlier phases. Implies that there are definite breaks between phases, i.e., that each phase has a strict, non-overlapping start and finish and is carried out sequentially.

Critical point is that no phase is complete until the documentation and/or other products associated with that phase are completed.

2.2.2 Modified Waterfall Model

Needed to provide for overlap and feedback between phases. Rather than being a simple linear model, it needed to be an iterative model. To facilitate the completion of the goals, milestones, and tasks, it is normal to freeze parts of the development after a certain point in the iteration. Verification and validation are added. Verification checks that the system is correct (building the system right). Validation checks that the system meets the users' desires (building the right system).

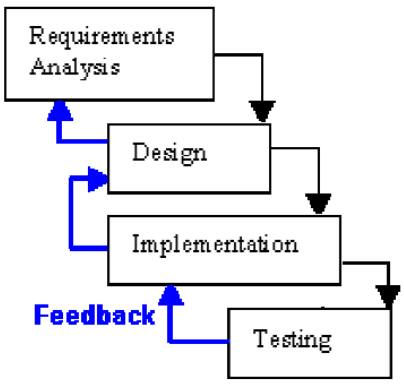


Figure: Modified Waterfall model

The waterfall model (and modified waterfall model) are inflexible in the partitioning of the project into distinct phases. However, they generally reflect engineering practice.

Considerable emphasis must be placed on discerning users' needs and requirements before the system is built. The identification of users' requirements as early as possible, and the agreement between user and developer concerning those requirements, often is the deciding factor in the success or failure of a software project. These requirements are documented in the requirements specification, which is used to verify whether subsequent phases are complying with

the requirements. Unfortunately specifying users' requirements is very much an art, and as such is extremely difficult. Validation feedback can be used to prevent the appearance of a strong divergence between the system under development and the users' expectations for the delivered system. Unfortunately, the waterfall lifecycle (and the modified waterfall lifecycle) are inadequate for realistic validation activities. They are exclusively document-driven models. The resulting design reality is that only 50% of the design effort occurs during the actual design phase with 1/3 of the design effort occurring during the coding activity! This is topped by the fact that over 16% of the design effort occurs after the system is supposed to be completed! In general, the behavior of many individuals in this type of process is opportunistic. The boundaries of phases are indiscriminately crossed with deadlines being somewhat arbitrary.

Rapid Prototyping

Prototyping also referred to as evolutionary development, prototyping aims to enhance the accuracy of the designer's perception of the user's requirements. Prototyping is based on the idea of developing an initial implementation for user feedback, and then refining this prototype through many versions until a satisfactory system emerges. The specification, development, and validation activities are carried out concurrently with rapid feedback across the activities. Generally, prototyping is characterized by the use of very high-level languages, which probably will not be used in the final software implementation but which allow rapid development, and the development of a system with less functionality concerning quality attributes such as robustness, speed, etc.

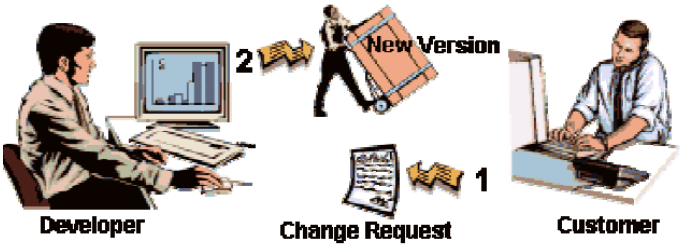


Figure: Rapid Prototyping model

Prototyping allows the clarification of users' requirements through, particularly, the early development of the user interface. The user can then try out the system, albeit a (sub) system of what will be the final product. This allows the user to provide feedback before a large investment has been made in the development of the wrong system.
There are two types of prototypes:

- Exploratory programming: The objective is to work with the user to explore their requirements and deliver a final system. Starts with the parts of the system which are understood, and then evolves as the user proposes new features.
- Throw-away prototyping: The objective is to understand the users' requirements and develop a better requirements definition for the system. Concentrates on poorly understood components.

Experiments with prototyping showed that this approach took 40% less time and resulted in 45% less code; however, it produced code that was not as robust, and therefore more difficult to maintain. Documentation was often sacrificed or done incompletely. The schedule expectations of users and managers tended to be unrealistic, especially for throw-away prototypes.

Boehm's Spiral Model

Need an improved software lifecycle model which can subsume all the generic models discussed so far. Must also satisfy the requirements of management.
Boehm proposed a spiral model where each round of the spiral

- identifies the subproblem which has the highest risk associated with it
- finds a solution for that problem.

Imperative (Procedural) Programming Paradigm

Any imperative program consists of

- Declarative statements give a name to a value. A named value is called a variable. Thus, declarative statements create variables. In procedural languages, it is common for the same variable to keep changing value as the program runs.
- Imperative statements which assign new values to variables
- Program flow control statements which define the order in which imperative statements are evaluated.

Example:

```
var factorial = 1; /*Declarative statement*/
var argument = 5;
var counter = 1;
while (counter <= argument) /*Program flow statement*/
{
factorial = factorial*counter; /*Imperative statement*/
counter++;
}
```

Variables and Types

Different variables in a program may have different types. For example, a language may treat two bytes as a string of characters and as a number. Dividing a string '20' by number '2' may not be possible. A language like this has at least two types - one for strings and one for numbers.

Example:

```
var PersonName = new String(); /*variable type "string"*/
var PersonSalary = new Integer(); /*variable type "integer"*/
```

Types can be weak or strong. Strong type means that at any point in the program when it is running, the type of a particular chunk of data (i.e. variable) is known. Weak type means that imperative operators may change a variable type.

Example:

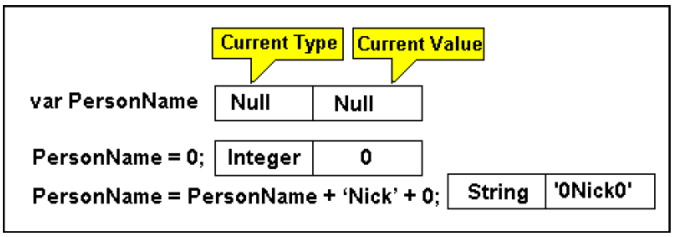
```
var PersonName; /*variable of a weak type"*/
PersonName = 0; /*PersonName is an "integer"*/
PersonName = 'Nick'; /*PersonName is a "string"*/
```

Languages supporting weak variable types need sophisticated rules for type conversions.

Example:

```
var PersonName; /*variable of a weak type"*/
PersonName = 0; /*PersonName is an "integer"*/
PersonName = PersonName + 'Nick' + 0; /*PersonName is a string "0Nick0"*/
```

To support weak typing, values are boxed together with information about their type - value and type are then passed around the program together.



Functions (Procedures)

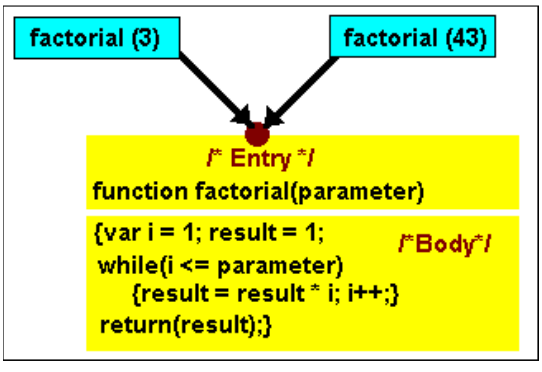
Programmers have dreamed/attempted of building systems from a library of reusable software components bound together with a little new code.

Imperative (Procedural) Programming Paradigm is essentially based on the concept of so-called "Functions" also known as "Modules", "Procedures" or "Subroutines".

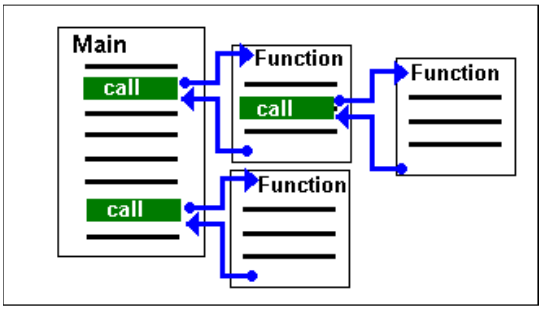
A function is a section of code that is parceled off from the main program and hidden behind an interface:

```
function factorial(parameter)
{
var i = 1;
var result = 1;
while(i <= parameter)
{
result = result * i;
i++;
}
return(result);
}
```

- The code within the function performs a particular activity, here generating a factorial value
- The idea of parceling the code off into a subroutine is to provide a single point of entry. Anyone wanting a new factorial value has only to call the "factorial" function with the appropriate parameters.



Here's what the conventional application based on the Imperative (Procedural) Programming Paradigm looks like:



- The main procedure determines the control flow for the application
- Functions are called to perform certain tasks or specific logic
- The main and sub procedures that comprise the implementation are structured as a hierarchy of tasks.
- The source for the implementation is compiled and linked with any additional executable modules to produce the application

Data Exchange between Functions (Procedures)

When a software system's functionality is decomposed into several functional modules, data exchange/flow becomes a key issue. Imperative (Procedural) Programming Paradigm extends the concept of variables to be used as such data exchange mechanism.

Thus, each procedure may have some special variables called parameters. The parameters are just named place-holders which will be replaced with particular values (or references to existing values) of arguments when the procedure is called.

Example:

```
function main()
{
var argument = 25;
var result = factorial(argument)
/* Note, the imperative operator replaces the “parameter” place holder with a current value of the variable
“argument”*/
}
-----------------------------------------------
function factorial(parameter)
{
var i = 1;
var result = 1;
while(i <= parameter)
{
result = result * i;
i++;
}
return(result);
}
```

Passing arguments to a function

There might be two different techniques for such replacement which are known as passing an argument value and passing an argument reference.

In the case of passing a value, a current argument value is duplicated as a value for a new parameter variable dynamically created for the procedure. In this case, variables used as arguments for calling sub-routines cannot be modified by imperative operators inside of the sub-routines.

In case of passing a reference, the sub-routine gets to control (i.e. reference) to a current value of the argument variable. In this case, variables used as arguments for calling sub-routines can be modified by imperative operators inside of the sub-routines.

Thus, types of variables defined as parameters of a function should be equivalent to (or at least compatible with) types of variables (constants) used as arguments.

Polymorphic Languages

When strong static typing is enforced it can be difficult to write generic algorithms - functions that can act on a range of different types. Polymorphism allows "any" to be included in the type system. For example, the types of a list of items are unimportant if we only want to know the length of the list, so a function can have a type that indicates that it takes lists of "any" type and returns an integer. Moreover, polymorphism allows combining functions implemented employing different programming languages supporting potentially different types of variables.

Pragmatically speaking, polymorphic languages allow defining new types as hidden functions which should be automatically applied to values of such “user-defined type” to convert it to values of a “standard” language type.

Variable Scope

Normally, variables that are defined within a function, are created each time the function is used and destroyed again when the function ends. The value that the function returns is not destroyed, but it is not possible to assign a value to the variable inside the function definition from outside.

Example:

```
function one()
{
var dynamicLocalVariable = 25;
two();
/* at this point just one variable “dynamicLocalVariable” exists */
alert(dynamicLocalVariable);
/* this operator displays the current value “25” */
}
function two()
{
var dynamicLocalVariable = 55;
/* at this point two variables “dynamicLocalVariable” exists */
alert(dynamicLocalVariable);
/* this operator displays the current value “55” */
}
```

Such variables are called dynamic local variables. There may be also so-called static local variables. Static local variables that are defined within a function, are created only once when the function is used for the first time. The value of such variable is not destroyed and can be reused when the function is called again.

Example:

```
function one()
{
var x = two();
alert(x);
/* this operator displays the current value “10” */
x = two();
alert(x);
/* this operator displays the current value “20” */
}
function two()
{
var static staticLocalVariable = 0;
staticLocalVariable = staticLocalVariable + 10;
return(staticLocalVariable);
}
```

Note that function “two” returns different values for the same set of arguments. Such functions are called reactive functions. Generally, testing and maintenance of projects having many reactive functions become a very difficult task. For practical reasons, many software projects do use some static data.
Note, it is still not possible to assign a value to the local static variable inside a function from outside.
There may be also so-called static global variables. Static global variables that are defined within any function, are created only once when the whole software system is initiated. The value of such a variable is never destroyed and can be reused by imperative operators inside any function.
Example:

```
function one()
{
var global globalLocalVariable = 0;
two();
alert(globalLocalVariable);
/* this operator displays the current value “10” */
two();
alert(globalLocalVariable);
/* this operator displays the current value “20” */
}
function two()
{
globalLocalVariable = globalLocalVariable + 10;
}
```

Here, the function “two” also demonstrates a “reactive” behavior. Maintaining and testing projects heavily based on global variables becomes even more difficult than in the case of local static variables. Nevertheless, for practical reasons many software development paradigms do use such global static variables.

Software Design Methodology (Procedural Paradigm)

Benefits of the Paradigm:

- Re-usability: anyone that needs a particular functionality can use an appropriate module, without having to code the algorithm from scratch.
- Specialization: one person can concentrate on writing the best possible module (function) for a particular task while others look after other areas.
- Upgradability: if a programmer comes up with a better way to implement a module then he/she simply replaces the code within the function. Provided the interface remains the same - in other words, the module name and the order and type of each parameter are unchanged - then no changes should be necessary for the rest of the application.

However procedural modules have serious limitations:

- For a start, there is nothing to stop another programmer from meddling with the code within a module, perhaps to better adapt it to the needs of a particular application.
- There is also nothing to stop the code within the function from making use of global variables, thus negating the benefits of a single interface providing a single point of entry.

The paradigm is best suited for the waterfall model of software development.

Design

A particular software system is viewed in terms of its modules and data flowing between them starting with a high-level view.

In this case, software design methodology can be categorized as a Top-down modular design (functional design viewpoint).

The basic design concepts include:

- Modularity
    - Modules are used to describe a functional decomposition of the system
    - A module is a unit containing:
        - executable statements
        - data structures
        - other modules
    - A module:
        - has a name
        - can be separately compiled
        - can be used in a program or by other modules
    - System design generally determines what goes into a module
- Cohesive
    - Single clearly defined function
    - Description of when and how used
    - Loosely Coupled Modules (Modules implement functionality, but not parts of other modules)
- Black Boxes (information hiding)
    - each module is a black box
    - each module has a set of known inputs and a set of predictable outputs
    - inner workings of the module are unknown to the user
    - can be reusable
- Preliminary and Detailed Design specifies the modules to carry out the functions in the DataFlow Diagrams (DFD).

Preliminary design deals mainly with Structure Charts

Hierarchical tree structure

- Modules - rectangle boxes
- calling relationships are shown with arrows
- arrows are labeled with the data flowing between modules

Module Design

- Title
- Module ID - from structure charts
- Purpose
- Method - algorithm
- Usage - who calls it
- External references - other modules called
- Calling sequence - parameter descriptions
- Input assertion
- Output assertion
- Local variables
- Author(s)
- Remarks

Preliminary Design Document

- Cover Page
- Table of Contents
- Design Description
- Software Structure Charts
- Data Dictionary
- Module Designs
- Module Headers
- Major Data Structures Design
- Design Reviews (Examination of all or part of the software design to find design anomalies)

Overview of Detailed Design

- select an algorithm for each module
- refine the data structures
- produce detailed design document

Implementation

Coding (for each Module)

- Source Code
- Documentation

Integration

- Decide what order the modules will be assembled
- Assemble and test the integration of modules
- After final assembly perform the system test
- Note, coding and testing are often done in parallel

Testing

Types of testing

- Unit testing
- Integration testing
- Acceptance testing

As was mentioned above, the paradigm is best suited for the waterfall model of software development. Implementing change requirements and especially rapid prototyping are weak points of the programming paradigm.

3A REQUIREMENTS AND SPECIFICATIONS

Abstract

Defining requirements to establish specifications is the first step in the development of an embedded system. However, in many situations, not enough care is taken in establishing correct requirements upfront. This causes problems when ambiguities in requirements surface later in the life cycle, and more time and money is spent on fixing these ambiguities. Therefore, requirements must be established in a systematic way to ensure their accuracy and completeness, but this is not always an easy task. This difficulty in establishing good requirements often makes it more of an art than a science. The difficulty arises from the fact that establishing requirements is a tough abstraction problem and often the implementation gets mixed with the requirements. In addition, it requires people with both communication and technical skills. As requirements are often weak about what a system should not do, this poses potential problems in the development of dependable systems, where these requirements are necessary to ensure that the system does not enter an undefined state. The development of dependable embedded systems requires even more complicated requirements as the embedded system not only interacts with the software but also with the outside world. Therefore, the importance of establishing good requirements is even greater in embedded systems design.

Introduction

Requirements and specifications are very important components in the development of any embedded system. Requirements analysis is the first step in the system design process, where a user's requirements should be clarified and documented to generate the corresponding specifications. While it is a common tendency for designers to be anxious about starting the design and implementation, discussing requirements with the customer is vital in the construction of safety-critical systems. Activities in this first stage have a significant impact on the downstream results in the system life cycle. For example, errors developed during the requirements and specifications stage may lead to errors in the design stage. When this error is discovered, the engineers must revisit the requirements and specifications to fix the problem. This leads not only to more time wasted but also the possibility of other requirements and specifications errors. Many accidents are traced to requirements flaws, incomplete implementation of specifications, or wrong assumptions about the requirements. While these problems may be acceptable in non-safety-critical systems, safety-critical systems cannot tolerate errors due to requirements and specifications. Therefore, the requirements must be specified correctly to generate clear and accurate specifications.

There is a distinct difference between requirements and specifications. A requirement is a condition needed by a user to solve a problem or achieve an objective. A specification is a document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system, and often, the procedures for determining whether these provisions have been satisfied. For example, a requirement for a car could be that the maximum speed is at least 120mph. The specification for this requirement would include technical information about specific design aspects. Another term that is commonly seen in books and papers is requirements specification which is a document that specifies the requirements for a system or component. It includes functional requirements, performance requirements, interface requirements, design requirements, and development standards. So the requirements specification is simply the requirements written down on paper.

Establishing Correct Requirements

The first step toward developing accurate and complete specifications is to establish correct requirements. As easy as this sounds, establishing correct requirements is extremely difficult and is more of an art than a science. There are different steps one can take toward establishing correct requirements. Although some of the suggestions sound fairly obvious, actually putting them into practice may not be as easy as it sounds. The first step is to negotiate a common understanding. There is a quote by John von Neumann that states "There's no sense being exact about something if you don't even know what you're talking about." [Gause89] Communication between the designer and customer is vital. There is no point in trying to establish exact specifications if the designers and customers cannot even agree on what the requirements are.

The problem stems from ambiguities in state requirements. For example, say the requirement states that we want to create a means that would transport a group of people from Boston to Washington D.C. Possible interpretations of this requirement include building a bus, train, or airplane, among other possibilities. Although each of these transportation devices satisfies the requirement, they are certainly very different. Ambiguous requirements can be caused by missing requirements, ambiguous words, or introduced elements. The above requirement does not state how fast the people should be transported from Boston to Washington D.C. Taking an airplane would certainly be faster than riding a bus or train. These are also missing requirements. "a group of people" in the above requirement is an example of ambiguous words. What exactly does "group" imply? A group can consist of 5 people, 100 people, 1000 people, etc. The requirement states to "create a means" and not "design a transportation device". This is an example of introduced elements where an incorrect meaning slipped into the discussion. It is important to eliminate or at least reduce ambiguities as early as possible because the cost of them increases as we progress in the development life cycle.

Often the problem one has in establishing correct requirements is how to get started. One of the most important things in getting started is to ask questions. Context-free questions are high-level questions that are posed early in a project to obtain information about the global properties of the design problem and potential solutions. Examples of context-free questions include who is the client? what is the reason for solving this problem? what environment is this product likely to encounter? and what is the trade-off between time and value?. These questions force both sides, designer, and customer, to look at the higher issues. Also, since these questions are appropriate for any project, they can be prepared in advance. Another important point is to get the right people involved. There is no point in discussing requirements if the appropriate people are not involved in the discussion. Related to getting the right people involved is making meetings work. Having effective meetings is not as easy as it sounds. However, since they play a central role in establishing requirements it is essential to know how to make meetings work. There are important points to keep in mind when creating effective meetings, which include creating a culture of safety for all participants, keeping the meeting to an appropriate size, and other points. [Gause89]

Exploring the possibilities is another important step toward generating correct requirements. Ideas are essential in establishing correct requirements, so it is important that people can get together and generate ideas. Every project will also encounter conflicts. Conflicts can occur from personality clashes, people that cannot get along, intergroup prejudice such as those between technical people and marketing people, and level differences. A facilitator must be present to help resolve conflicts.

In establishing requirements, it is important to specifically establish the functions, attributes, constraints, preferences, and expectations of the product. Usually, in the process of gaining information, functions are the first ones to be defined. Functions describe what the product is going to accomplish. It is also important to determine the attributes of a product. Attributes are characteristics desired by the client, and while 2 products can have similar functions, they can have completely different attributes. After all the attributes have been clarified and attached to functions, we must determine the constraints on each of the attributes. Preferences, which is desirable but optional condition placed on an

attribute, can also be defined in addition to its constraints. Finally, we must determine what the client's expectations are. This will largely determine the success of the product.

Testing is the final step on the road to establishing correct requirements. There are several testing methods used, as listed below. [Gause89]

- Ambiguity poll - Used to estimate the ambiguity in a requirement. This involves asking questions such as how fast? how big? how expensive? and then determining if there is ambiguity between the high and low values.
- Technical review - A testing tool for indicating the progress of the requirements work. It can be formal or informal and generally only deals with technical issues. Technical reviews are necessary because it is not possible to produce error-free requirements and usually it is difficult for the producers to see their own mistakes.
- User satisfaction test - A test used regularly to determine if a customer will be satisfied with a product.
- Black box test cases - Constructed primarily to test the completeness, accuracy, clarity, and conciseness of the requirements.
- Existing products - Useful in determining the desirable and undesirable characteristics of a new product.

At some point it is necessary to end the requirements process as the fear of ending can lead to an endless cycle. This does not mean that it is impossible to revisit the requirements at a later point in the development life cycle if necessary. However, it is important to end the process when all the requirements have been determined, otherwise, you will never proceed to the design cycle.
Establishing good requirements requires people with both technical and communication skills. Technical skills are required as the embedded system will be highly complex and may require knowledge from different engineering disciplines such as electrical engineering and mechanical engineering. Communication skills are necessary as there is a lot of exchange of information between the customer and the designer. Without either of these two skills, the requirements will be unclear or inaccurate.

Requirements in safety critical embedded systems must be clear, accurate, and complete. The problem with requirements is that they are often weak about what a system should not do. In a dependable system, it is just as important to specify what a system is not supposed to do as to specify what a system is supposed to do. These systems have an even greater urgency that the requirements are complete because they will only be dependable if we know exactly what a system will do in a certain state and the actions that it should not perform. Requirements with no ambiguities will also make the system more dependable. Extra requirements will usually be required in developing a dependable embedded system. For example, in developing a dependable system for non-computer-literate people, extra requirements should be specified to make the system safe even in exceptional or abusive situations.

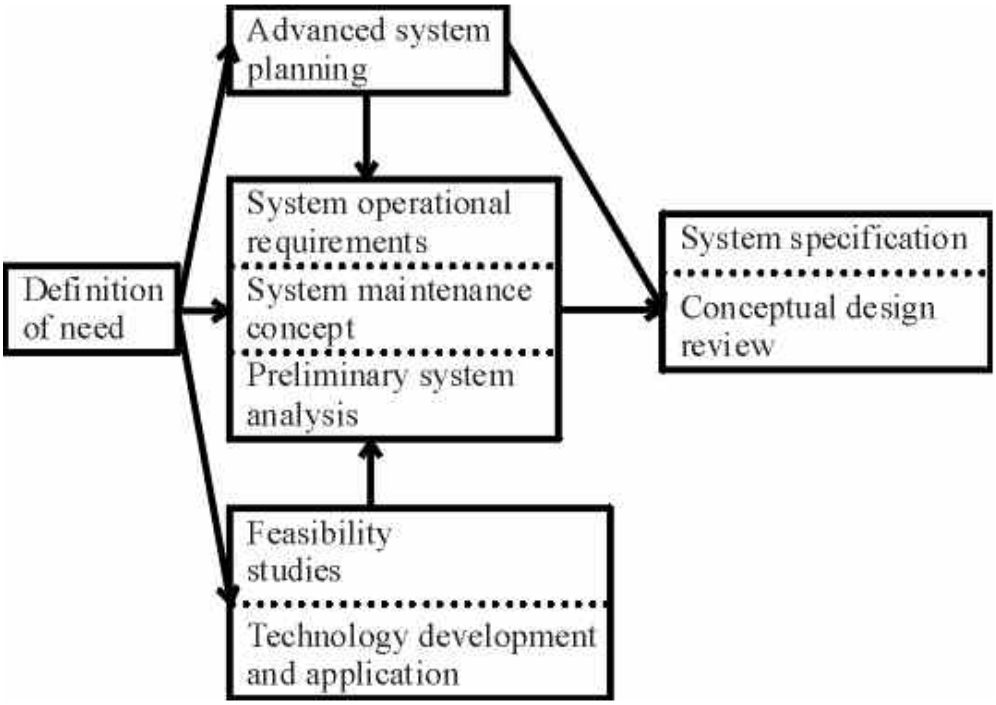
Requirements and Specification's Role in System Design

Systems exist everywhere in the universe we live in. The universe can be considered a system, and so can an atom. A system is very loosely defined and can be considered as any of the following definitions. [Blanchard90]

- A combination of elements forming a complex or unitary whole (i.e. river system or transportation system)
- A set of correlated members (i.e. system of currency)
- An ordered and comprehensive assemblage of facts, principles, or doctrines in a particular field of knowledge (i.e. system of philosophy)
- A coordinated body of methods, a complex scheme, or a plan of procedure (i.e. system of organization and management)
- Any regular or special method or plan of procedure (i.e. a system of marking)

The important characteristic of a system is that there is unity, functional relationship, and useful purpose. Systems engineering is not a technical specialty but is a process used in the evolution of systems from the point when a need is identified through production and construction to the deployment of the system for consumer use. [Blanchard90] Systems engineering requires knowledge from different engineering disciplines such as aeronautical engineering, civil engineering, and electrical engineering. The development of embedded systems also requires the knowledge of different engineering disciplines and can follow the techniques used for systems engineering. Therefore, it is appropriate that the steps used in establishing system requirements also apply to requirements for embedded systems.

The conceptual system design is the first stage in the systems design life cycle and an example of the systems definition requirements process is shown in Figure 1. Each box will be explained below.



In establishing system requirements, the first step is to define a need. This need is based on a want or desire. Usually, an individual or organization identifies a need for an item or function, and then a new or modified system is developed to fulfill the requirement. After a need is defined, feasibility studies should be conducted to evaluate various technical approaches that can be taken. The system operational requirements should also be defined. This includes the definition of system operating characteristics, maintenance support concept for the system, and identification of specific design criteria. In particular, the system operational requirements should include the following elements. [Blanchard90]

- Mission definition - Identification of the primary operating mission of the system in addition to alternative and secondary missions.
- Performance and physical parameters - Definition of the operating characteristics or functions of the system.
- Use requirements - Anticipation of the use of the system.
- Operational deployment or distribution - Identification of transportation and mobility requirements. Includes a quantity of equipment, personnel, etc., and geographical location.
- Operational life cycle - Anticipation of the time that the system will be in operational use.
- Effectiveness factors - Numbers specified for system requirements. Includes cost-system effectiveness, mean time between maintenance(MTBM), failure rate, maintenance downtime, etc.
- Environment - Definition of the environment in which the system is expected to operate.

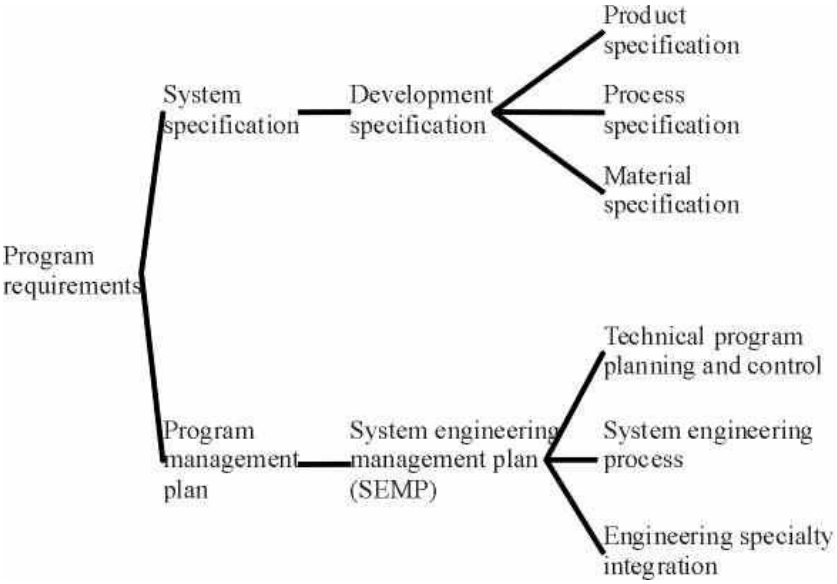
The system operational requirements define how the system will be used in the field by the customer.

Usually, in defining system requirements, the tendency is to cover areas that are related to performance as opposed to areas that are related to support. However, this means that emphasis is only placed on part of the system and not the whole system. It is essential to take into consideration the entire system when defining system requirements. The system maintenance concept describes the overall support environment that the product is supposed to exist in.

After the system operational requirements and system maintenance concept are defined, the preliminary system analysis is performed to determine which approach for system development should be adopted. The following process is usually applied. [Blanchard90]

1. Define the problem - The first step always begins with clarifying the objectives, defining the concerned issues, and limiting the problem so that it can be effectively studied.
2. Identify feasible alternatives - All the alternatives should be considered to make sure that the best approach is chosen.
3. Select the evaluation criteria - The criteria for the evaluation process can vary considerably, so the appropriate ones must be chosen.
4. Applying modeling techniques - A model or series of models should be used.
5. Generate input data - The requirements for appropriate input data should be specified.
6. Manipulate the model - After data is collected and inputted, the model may be used. Analysis after using the model will lead to a recommendation for some kind of action.

After the preliminary system analysis, advanced system planning will be done. Early system planning takes place from the identification of a need through the conceptual design phase. The results from these plans will be defined as either technical requirements included in the specifications or management requirements included in a program management plan. The documents associated with these requirements are shown in Figure 2. The system specification includes information from the operational requirements, maintenance concept, and feasibility analysis. The System Engineering Management Plan(SEMP) contains three sections. The technical program planning and control part describes the program tasks that have to be planned and developed to meet system engineering objectives such as work breakdown structure, organization, risk management, etc. The system engineering process part describes how the system engineering process applies to program requirements. Finally, the engineering specialty integration part describes the major system-level requirements in the engineering specialty areas such as reliability, maintainability, quality assurance, etc.



Finally, the conceptual design review is also performed during the conceptual design stage. It usually occurs early in the system engineering development life cycle after the operational requirements and the maintenance concept have been defined.

Requirements Traceability

It is very important to verify that the requirements are correctly implemented in the design. This is done with requirements traceability which is usually referred to as "the ability to follow the life of a requirement, in both forwards and backward direction (i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases.)" [Ramesh95] Requirements traceability captures the relationships between the requirements, specifications, and design. Standards for systems development such as the one from the U. S. Department of Defense (standard 2167A) require that requirements traceability be used. Although requirements traceability has been around for more than 2 decades, there has been no consensus as to what kind of information should be used as part of a traceability scheme. The problem is that the definition of traceability differs when taken from different points of view of the system. (i.e. the view of the system is different for customers, project managers, test engineers, etc.) Each organization has a different purpose and methodology for requirements tracing. While it is not the purpose of this paper to dwell on a long discussion about requirements traceability, a short example of the methodology used at one organization will be given. [Ramesh95]

The projects typically involved at Abbott Laboratories Diagnostics Division are real-time, embedded, in vitro diagnostic instruments approaching 200,000 lines of code. They have found that traceability aids project managers in verification, cost reduction, accountability, and change management. Traceability helps in verifying that software requirements are satisfied in the design process and that they are tested for during the verification process. Traceability allows the allocation of product requirements early in the development cycle thereby reducing costs of correcting defects (due to untraceable components) in the integration and system test phase. Providing quantitative traceability analyses also allows for accountability in making sure that project milestones are approved, deliverables are verified, and customers are satisfied. The documentation from traceability also keeps information organized during changes in staff or management.

A specific in vitro diagnostic instrument contained approximately 175,000 lines of source code and approximately 1,600 software requirements that needed to be traced. While the division also has an automated traceability system (ATS) that allowed them to automate many of the tasks, it was the process and not the tool that led to their success. The main purpose of the traceability program is to identify links and determine that the links are complete and accurate. The traceability analysis consists of 4 aspects: forward requirements analysis, reverse requirements trace, forward test analysis, and reverse test trace. These steps are used to trace each software requirement through its design elements and test traces. The ATS can be used to design documentation matrices and test matrices that are used to perform the different analyses required. The ATS is also able to give feedback about the design components that are not yet implemented during the life cycle. In the test phase, the ATS gives input to what requirements are covered by the test cases. [Watkins94]

Requirements Standards

There are many requirements and specification standards. They are mostly military standards as opposed to "commercial" standards. In addition, most of the standards are in the systems engineering area and particularly deal with the software aspects. A good reference to many of these standards is Standards, Guidelines, and Examples on System and Software Requirements Engineering from the IEEE Computer Society Press. [Dorfman90] This book is a compilation of international requirements standards and U. S. military standards. There is also a section on requirements analysis methodologies and examples. Listed below are several relevant standards, but the list is by no means exhaustive.

- IEEE Recommended Practice for Software Requirements Specifications (IEEE std 830-1998)

- British Standard Guide to Specifying User Requirements for a Computer-Based Standard (BS6719 - 1986)
- Canadian Standard, Basic Guidelines for the Structure of Documentation of System Design Information (Z242.15.4-1979)
- U. S. Military Standards
- System/Segment Specification (DI-CMAN-80008A, 2/29/1988)
- Software Requirements Specification (DI-MCCR-80025A, 2/29/1988)
- Interface Requirements Specification (DI-MCCR-80026A, 2/29/1998)

3B DIFFERENT REQUIREMENTS GATHERING TECHNIQUES AND ISSUES

Abstract

Project management is now becoming a very important part of our software industry. To handle projects with success is a very big deal. In the software project management process there are some phases, first phase is requirement gathering. To get the correct requirement and to handle it, is most important for the complete project successfully. Requirement management is used to ensure that product or software meets the user's needs or expectations. Requirements are defined during the planning phase and then these requirements are used throughout the project. There are some techniques for gathering requirements. These techniques are interview, prototyping, use case analysis, JAD (Joint Application Design), brainstorming questionnaires, and storyboard. While gathering requirements, we faced many issues that are not capable of a successful project. This paper, there will be discussed these techniques and issues that are faced during requirement gathering and their solution.

Introduction

Requirements analysis is critical to the achievement of a development project. Requirements should be measurable, actionable, and testable and also should be related to the user's expectations. Requirement without any ambiguity fulfill the user's requirement make the project successful. While gathering requirements focused on "what" should be required rather than "how" it is required.

"Using peer reviews, scenarios, and walkthroughs to validate and verify requirements results in a more accurate, specification and higher customer satisfaction." It is estimated that 85% of defects are found in requirements during software development. Some techniques are used to gather requirements. Every technique is not used for every project. In these techniques some are useful and some are not but it depends on the project description.

Good requirement specifications are listed.

- Complete
- Verifiable
- Unambiguous
- Modifiable
- Traceable
- Usable during operations and maintenance

These requirements specifications produce a good project. There are some requirements types. Every project has some kind of requirements like:

- Functional requirements
- Non- Functional requirements
- Domain requirements
- Inverse requirements

During requirements elicitation, there may be many issues that have to face. That issue and its solutions will be discussed in this chapter. Different techniques and which one is best for which type of project will be discussed in this chapter.

Techniques of Requirements Gathering

In reality, there are hundreds of different techniques for requirement elicitation. In this paper, some commonly used techniques are mentioned. These are the following:

- Interview
- Questionnaires
- Brainstorming
- Storyboard
- Prototyping
- Use cases
- AD (Joint Application Development)

Interview

The interview is a common technique used for gathering data or information. In this technique, the interviewer conducts a meeting with the interviewee. Interviews questions should be according to the interviewee's level. Gather information according to his/her requirement. Questions should be open-ended however; the interviewee can provide a clear answer
of your question. There are three types of questions. These are structured, unstructured, and semi-structured. Structured interviews are conducted where the domain is specified. In this type-specific questions are asked and get to the point answer. In this way, all the questions are covered up in this type. The other is an unstructured interview in this type interviewer asks questions and requires a detailed answer to these questions. The interviewer applies only partial control over the way of discussion. In this way, some topics may be neglected [6]. Semi-structured is a combination of both. The semi-structured interview, where the elementary usual of the question is organized and used.

Questionnaires

The questionnaire is the best technique for gathering information. In this technique, questions are listed on paper. Questions are filled by the stakeholders and get the answer of these questions. In this technique, stakeholders cannot express their idea. No new dimension can be defined. The questionnaire type focused on the limited information eliminated unnecessary information.

Brainstorming

Brainstorming technique is a group discussion in which members share their ideas and find out the solution to the specific problem. Brainstorming generates or gathers new ideas rather than their quality. This technique is more popular because it is a group activity in which all the members share their idea. It is more productive for the reason that groups. When members generate ideas it is more value able as of group product and members enjoy the group activity. There is a method for conducting brainstorming tasks. These are:

- Subjects and design
- Procedure
- Results

- Discussion

It is not compulsory to manage brainstorming sessions to resolve major issues. The purpose of this technique is the introductory mission statement for a specific problem. The advantage of this technique is encouraging open-minded and free ideas or innovation on a particular predefined problem.

Storyboard

In this technique users, customers and developers draw a picture of what they want to develop software. Draw a picture of all requirements like toolbar, main window, dialogue boxes, etc. After drawing a full picture of all requirements, all members agreed upon it. It is just like paper prototyping [1]. Storyboarding is a very common technique for designing about which you want to get information for their project. Storyboarding is much more realistic for understanding software's structure for unknown persons who do not know about technical terms. There are some attributes or elements of storyboarding, which explain basic points for drawing storyboarding. These are:

- Level of details
- Inclusion of text
- Inclusion of people and emotions
- Number of frames
- Portrayal of time

The explanation is given below.

- Level of details
  The level of details describes the existence of actors and objects. It depends upon the designer how to draw a scene or describe only the detail of the interface.

- Inclusion of text
  In storyboarding text could be included in the design with each section. It may be possible designer will not use text.

- Inclusion of people and emotions
  During designing end-users should be in mind. Design should be according to end-user that affects the user or stakeholders.

- Number of frames
  Some frames should be in mind mostly 1 to 20 frames are included in each story. Each storyboard contains a different number of frames according to its requirement. Several features are contained in storyboards.

- Portrayal of time
  Time passing is included in storyboarding or used transitions.

Prototyping

Prototyping is a more significant technique for gathering requirements. Through prototyping, detailed requirements can be gathered if preliminary requirements are already collected [9]. Prototyping is much effective for gathering relevant information from users; users provide relevant information and also provide feedback. This technique is useful when users or stakeholders are not aware of technical terms in this way they deliver the right information and react to their requirements which are developed by designers or developers. Sometimes this technique is expensive in tenure of cost and time [6]. The prototype can be flat diagrams. It helps us to prevent misperception.

Use cases

Use case analysis is a document that defines the relationship between actor and system. Arrangement of actions a user uses a system to complete a procedure. Define how the system will behave in a particular situation. A use case can be used to represent business functionality [10]. An actor is used as interaction with the system how to discuss with the system or its environment. The use case will be successful when its goal is satisfied. Use case description also includes in use case analysis. Use case steps are written in an easy and understandable format of a use case diagram. The system is preserved such as black box, in which actor presents as whom, what will interact as system and purpose or goal for interaction with the system without knowing about the internal system. Here is the format of the use case description [11].

Template of Use Case

Use case No.	Goal name in verb phase
Goal	Longer detail of goal statement
Scope and level	What system is going to present, summary
Preconditions	What we expected is before now
Success end condition	Successful completion
Failed end condition	What will objective if goal aborted
Primary, secondary actors	Name of primary and secondary actors and their role
Trigger	Activities upon the system that start use case
Description	Description of all over the use case

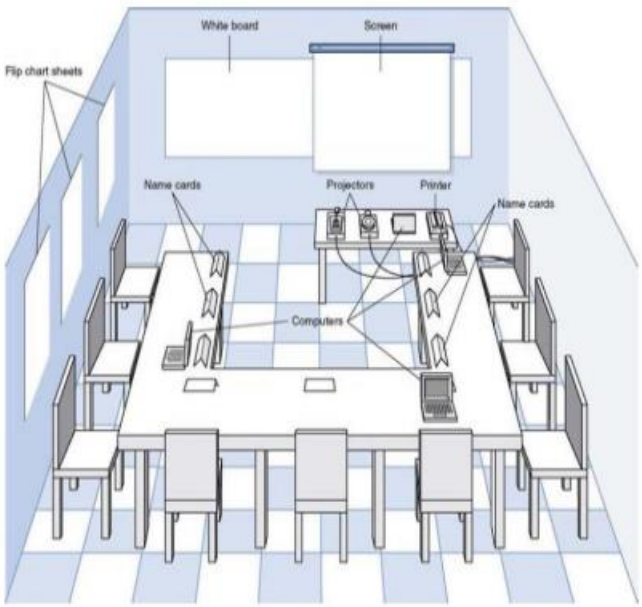
In table 1 there is given a template of use case description. Actors who will interact with the system, goal of use case description, precondition and if use case will not successful then failure condition all these points are mentioned in the use case description. Diagram of use case can present with an actor, use case, and system boundary through which actor interact with the use case. Diagram of use case presented as:

JAD (Joint Application Development)

In this technique, all stakeholders are included for the solution of the problem or for gathering information. With all parties, the decision can be made speedily. The main difference between JAD and brainstorming is that systems have previously been recognized before stakeholders take part. JAD session is well-maintained with defined phases and the role of actors. This type of technique is used to solve business issues rather than technical issues [6]. In the JAD session, a facilitator includes guideline of system requirement and help out users to resolve interview, provide information, and make decisions [12]. In the early JAD acronym was DESIGN, but now it become joint Application Development. JAD has five stages and their activities; a table of five stages is given below [13].

JAD Stages	Activities
Project Definition	✓ Define system goal, objective. ✓ Identify JAD team fellows. ✓ Establish project schedule.
Background Research	✓ Gather background knowledge of user requirement. ✓ Known about general issues that will discuss in JAD session.
Pre workshop Preparation	✓ Organize for session. ✓ Prepare all the documents and visual aids. ✓ Train the illuminator.
The Workshop	✓ Conclude solution within three to five day session. ✓ Finalize document meeting decision.
Final Documentation	✓ Prepare Closing document that contain final decision attained at through workshop.

JAD is useful due to some reasons like non-contributors are encouraged, dominance is reduced during the session, side discussions are included and true conflicts are under consideration in the JAD session. The meeting room of JAD is displayed below in figure 2 [4].



Issues in Gathering Requirements

When talking about requirement gathering or requirement elicitation then there may be many issues to gathering data from users or stakeholders. Here will be describing some issues and solutions of these issues [6].

- Scope
- Communication and understanding
- Quality of requirements
- Stakeholders
- Practice

The detail of issues and their solution are specified below.

Scope

The big issue of requirement elicitation is scope. Sometimes users or stakeholders are not familiar with or know the scope of the project. They can not specify the goal of their project. When a scope issue occurs, then it creates an issue to gather information from users. The scope should be limited and clearly define. However, requirements can be gathered correctly according to the user's needs or according to the nature of the project. The scope is very essential for good software project management. It has seemed that some projects are very worthy but due to limited scope, these are not successful.

Communication and understanding

In communication and understanding issues mostly faced end users. During communication with stakeholders, some issues may be a language issues. Stakeholders may be possible they do not know the language or it may possible they are not familiar with your condition and terms. Another communication issue could be that, however, their language is different so their rules may be different. So they are not able to understand your terms and they are not able to specify their requirement. Communication issues have four dimensions of the framework. These dimensions show the performance during the activity of requirement gathering. These are [14]:

- Stakeholder participant and selection
- Stakeholder interaction
- Communication activities
- Techniques

Explanations of these dimensions are given that can help to solve communication issues.

- **Stakeholder participant and selection**

First select stakeholders for gathering information. Selection should be on the right bases and the right users. The selection of stakeholders should be based on domain knowledge, about which domain have to gather requirements. Sometimes it might be happened to select stakeholders based on their position rather than their knowledge.

- **Stakeholder interaction**

Stakeholder interaction includes getting information from them through the meeting. In interaction, political or cultural issues may arise. Different cultures have different languages so there should be a common language to understand each other. The meeting schedule should be managed to agree to both parties.

- **Communication activities**

Communication will possible when both parties cooperate and negotiate. Communication activities categorize into three ways. One is knowledge acquisition, acquiring knowledge. Second is knowledge negotiation, negotiation with other stakeholders for the share of knowledge and requirement needs. Third is stakeholder acceptance, these requirements

should be accepted by stakeholders.

Techniques

Techniques used for the link to a developer with a customer. To maintain the relationship between the two techniques are used, Group and traditional. In group focus, brainstorming and workshops are included. In the traditional questionnaire, interviews and analysis of existing documents are included.

Quality of requirements

During gathering information some users did not provide the right information they are not serious. Sometimes their environment at that time is not according to their needs so they are not able to provide the right information. Requirement elicitation is the first phase for initialing any project so information should be correct and complete but stakeholders or other users where have to get requirements that do not provide correct and complete information that affects the overall project. In the end, project quality does not get the best due to the quality of requirements. The solution is that carefully gets information according to their need and put a limited question and get to the point answer in this way quality of the requirement can be improved.

Stakeholders

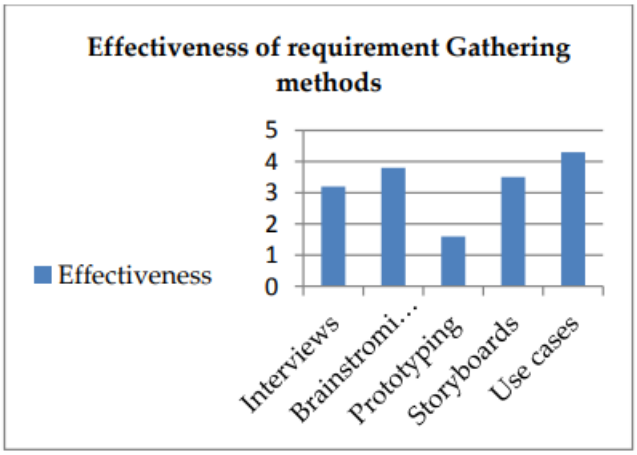
A stakeholder is one of the main issues during requirement gathering. Stakeholders do not know what they want and what their need is. They do not satisfy with one point because they do not know about technical terms. They have no idea about software working and system how it works. These are some conflicts that should be removed. This issue arose due to the unawareness of stakeholders They do not cooperate the whole time with project team members. On the other hand, they are not able to finalize the solution of the decision of any problem.

Practice

Generally, practice is a good factor in requirement elicitation. Sometimes not expert analysts are available for requirement gathering or there may be some gap between requirement theory and practice. Sometimes analysts repeat the mistake again and again. So, hire experience analysts for this purpose.

Effectiveness of Requirements Gathering Techniques

There is a graph that represents the overall effectiveness of different requirement techniques [15].



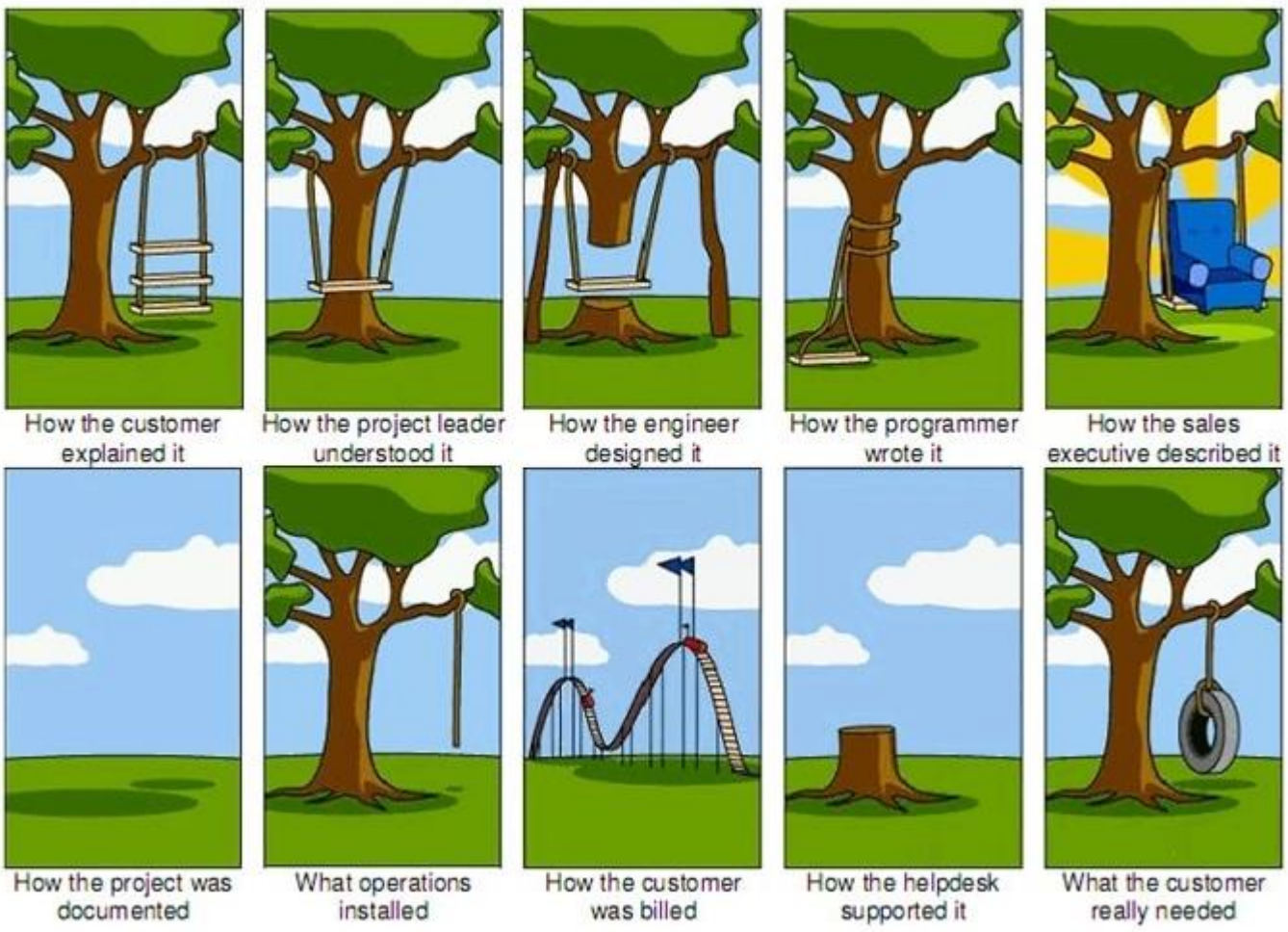
This graph shows the effectiveness of different techniques that are defined in this paper. Use cases are most effective because every use case presents a brief description of what will happen what are causes of that use case precondition, postcondition, other actors if are involved in it all these explanations is defined in use case scenario.

In any project main and the first phase is requirement elicitation. Correct information from stakeholders is significant for success and without any defective project. So, this paper is about different techniques of requirements elicitation. When talking about techniques then there are some issues. In requirement gathering, every technique is not used for every project. Some techniques have some issues that are cause techniques are used according to the project's nature. Every technique has some benefits and some drawbacks. Before using the technique check specifications of these techniques, which one is best or suitable for which type of project? At the end of this paper defined the result of techniques of requirement gathering, effectiveness of selected defined techniques.

3C STEPS TO SUCCESSFUL REQUIREMENTS GATHERING

What happens if you skip gathering requirements for your software project?

Depending on your project methodology, you may do this step at the beginning during a Discovery phase, you may do it during the project within each sprint or build cycle, or you may skip it altogether and hope for the best. That last option is a simple way to sabotage your project and guarantee a lot of late nights and awkward status meetings.



Tips for Successful Requirements Gathering

Successful requirements gathering is both an art and a science, but there are some general steps you can take to keep this all-important aspect of your project on the right path. Here are some guidelines that we try to follow at Phase2:

1. Establish Project Goals and Objectives Early

This step can feel redundant: of course, we know why we're doing this project…don't we? Even if you think you know, write it down, and get your stakeholders to sign off on it. Without clearly stated goals and objectives, you are lacking a framework to guide future decision-making. How do you know if a newly introduced requirement fits your project? Simple: does it help accomplish a goal, or does it satisfy an objective? If so, it's probably a good fit. If not, it's a good candidate for a future release.

2. Document Every Requirements Elicitation Activity

When you're amid stakeholder interviews and documentation reviews, you can often feel like you have a great grasp on things. But then a week goes by, and some details start to get a little fuzzy, and you realize you don't quite have a full grasp of your business requirements. It sounds obvious, but making sure that you are taking detailed notes during your stakeholder interviews is a powerful step in successful requirements gathering. And it's not enough to just write everything down, as you'll see in #3…

3. Be Transparent with Requirements Documentation

Sure, you understand the requirements. And your stakeholders understand the requirements. But do your stakeholders understand your understanding of the requirements?

After every meeting, go through your notes and clean them up – then share them with the project team, including the stakeholders. This transparency not only helps make sure everyone's on the same page but also fosters a sense of project buy-in through your project, beginning with the business requirements. And it circumvents the issue of someone saying "hey, you agreed to X but it's not here!" 6 weeks into the project. If it's not in the notes, it didn't happen.

4. Talk to The Right Stakeholders and Users

A project can often have "hidden" stakeholders. Ask probing questions in your kickoff and initial meetings to try and get to who the real users are. Often those people are not going to be the main decision-makers, but their buy-in is essential to a successful project. Disgruntled users who are forced to use a system every day that was designed without their input are a key ingredient for a failed project.

5. Don't Make Assumptions About Requirements

Don't assume that you understand everything, even if it seems obvious. A seemingly simple requirement such as "we want a blog" can mask all sorts of underlying assumptions, requirements, etc. What are the fields for a blog post? How are authors managed? What about tagging? Categories? How are the posts displayed? Are they aggregated into an archive? Is there an RSS feed? Who are the authors and what is their level of technical proficiency? Etc. etc. etc. The devil truly is in the details, but you can catch him by the tail if you ask a lot of questions and don't rely on assumptions.

6. Confirm, Confirm, Confirm

This ties into "be transparent" but is not entirely the same thing. Just sharing your notes with a stakeholder is great, but far more valuable is having a quick review with them and getting their official sign-off. This is true for meeting notes, user stories, diagrams, wireframes, really any kind of requirements artifact that you are creating. Radio silence is not an indicator of success – get actual confirmation from your stakeholders that you are representing the requirements correctly in whatever format you're using, then move on.

7. Practice Active Listening

Making someone feel heard is one of the greatest things you can do for them. But it goes beyond just listening to what they say – you also need to listen to what they don't say, and how they say things, and read their body language, etc. This is called active listening and it's a key component of successful requirements gathering. Don't assume that you're always getting the whole story – listen for little cues that reveal pain points, desires, unstated goals, and assumptions.

8. Focus On Business Requirements, Not Tools

Be careful when you are gathering requirements that you are focusing on and listening to what your stakeholder needs, not what your tool-of-choice happens to do best. Even if you know you are going to be using a certain product, you need to adapt the product to the user, not the other way around. Listen and gather first, then determine where the gaps are between your stakeholder's needs and any existing product you may have in mind. Remember: requirements are about the WHAT, not the HOW.

9. Prioritize Your Product Features

In an agile methodology, we work towards a Minimum Viable Product (MVP), which encapsulates the least amount of functionality that would count as a successful product at launch. Even when following a non-agile methodology, prioritizing is your friend when you are gathering requirements. It's easy for requirements gathering sessions to turn into wish list gathering sessions, where stakeholders tell Santa (i.e. you) everything they want. The point isn't to ignore that information (in fact it often reveals goals and assumptions if you're using Active Listening) but rather to clearly and transparently prioritize what you're hearing and delineate what is in scope for your initial launch and what is not. You want to track wish-list items, "nice-to-haves," etc. but prioritizing helps you focus your efforts and helps you make decisions if time gets short and something has to go.

10. Remember That You Didn't Get Everything

Even the best requirements gatherer is going to miss things. Why? Because you and your stakeholders are human beings, and human beings make mistakes. You will think of things later that you forgot to ask. Your stakeholder will think of things that they forgot to mention. Things will change. Priorities will shift. The good news is that if you plan for this, you can build in time during your project lifecycle for ongoing requirements management. This time is essential because requirements (being human-driven and human-created) are simply not static. Giving yourself time to actively manage requirements throughout the entire project can help you stop scope creep before it starts, and make sure that your team is always focusing on the right set of priorities that match actual requirements.

There's a lot more that can be said about the art and science of requirements gathering, but hopefully, this list has given you some helpful tools to manage this process successfully. Now that you know how to define what you're building, learn how to align your stakeholders around a common vision for your project.

4 REQUIREMENTS ANALYSIS AND TECHNIQUES

Requirement Analysis Techniques

Requirement Analysis, also known as Requirement Engineering, is the process of defining user expectations for a new software being built or modified. In software engineering, it is sometimes referred to loosely by names such as requirements gathering or requirements capturing. Requirements analysis encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product or project, taking account of the possibly conflicting requirements of the various stakeholders, analyzing, documenting, validating, and managing software or system requirements. Here are the objectives for performing requirement analysis in the early stage of a software project:
From What to How: Software engineering task bridging the gap between system requirements engineering and software design.

- 3 Orthogonal Views: Provides software designer with a model of:
- system information (static view)
- function (functional view)
- behavior (dynamic view)
- Software Architecture: Model can be translated to data, architectural, and component-level designs.
- Iterative and Incremental Process: Expect to do a little bit of design during analysis and a little bit of analysis during design.

What is the Requirement?

A software requirement is a capability needed by the user to solve a problem or to achieve an objective. In other words, a requirement is a software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documentation. Ultimately, what we want to achieve is to develop quality software that meets customers' real needs on time and within budget.
Perhaps the greatest challenge being faced by software developers is to share the vision of the final product with the customer. All stakeholders in a project - developers, end-users, software managers, customer managers - must achieve a common understanding of what the product will be and do, or someone will be surprised when it is delivered. Surprises in software are rarely good news.
Therefore, we need ways to accurately capture, interpret, and represent the voice of customers when specifying the requirements for a software product.

Activities for Requirement Analysis

Requirements analysis is critical to the success or failure of a systems or software project. The requirements should be documented, actionable, measurable, testable, traceable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design. Conceptually, requirements analysis includes four types of activity:

1. Eliciting requirements: the task of communicating with customers and users to determine what their requirements are. This is sometimes also called requirements gathering.
2. Analyzing requirements: determining whether the stated requirements are unclear, incomplete, ambiguous, or contradictory, and then resolving these issues.

3. Requirements modeling: Requirements might be documented in various forms, such as natural-language documents, use cases, user stories, or process specifications.
4. Review and retrospective: Team members reflect on what happened in the iteration and identify actions for improvement going forward.

Requirements analysis is a team effort that demands a combination of hardware, software, and human factors engineering expertise as well as skills in dealing with people. Here are the main activities involve in requirement analysis:

- Identify customers' needs.
- Evaluate the system for feasibility.
- Perform economic and technical analysis.
- Allocate functions to system elements.
- Establish a schedule and constraints.
- Create system definitions.

Requirement Analysis Techniques

A requirement analysis has a

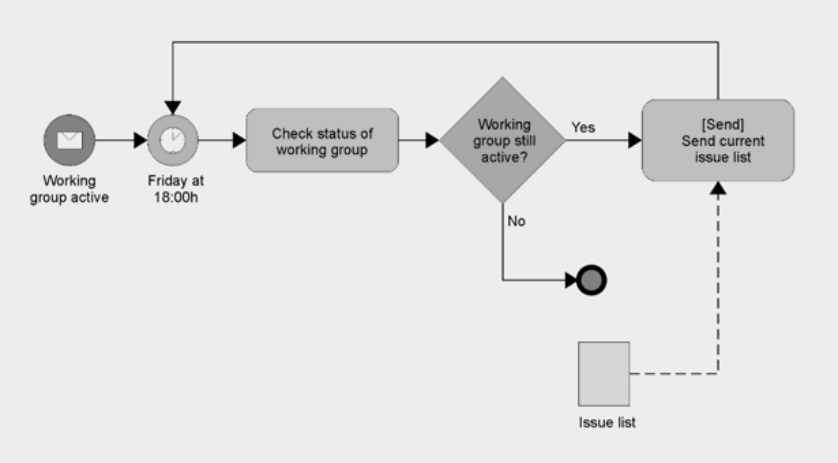
- Specific Goal
- Specific Input
- Specific Output
- Uses resources
- Has several activities to be performed in some order
- It May affect more than one organization unit
- Creates value of some kind for the customer

1. Business process modeling notation (BPMN) BPMN (Business Process Modeling & Notation) is a graphical representation of your business process using simple objects, which helps the organization to communicate in a standard manner. Various objects used in BPMN include
    - Flow objects
    - Connecting objects
    - Swim lanes
    - Artifacts

A good design BPMN model should be able to give the detail about the activities carried out during the process like,

- Who is performing these activities?
- What data elements are required for these activities?

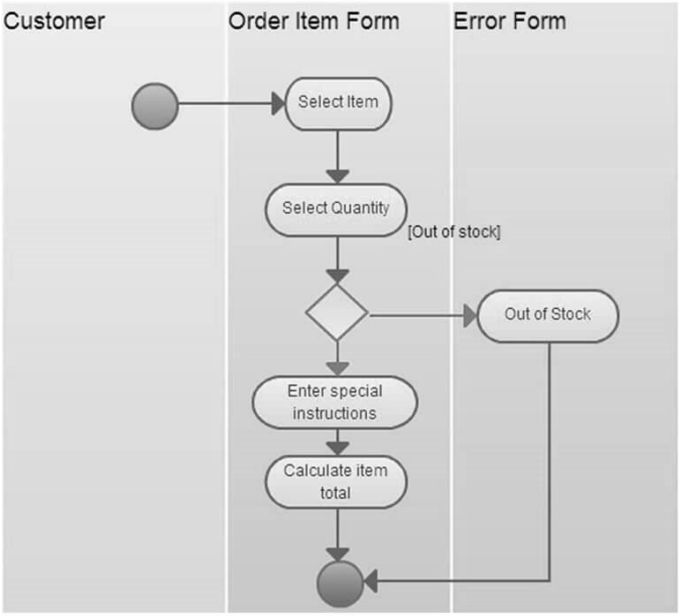
The biggest benefit of using BPMN is that it is easier to share, and most modeling tools support BPMN.



2. UML is a modeling standard primarily used for the specification, development, visualization, and documenting of a software system. To capture important business processes and artifacts UML provides objects like

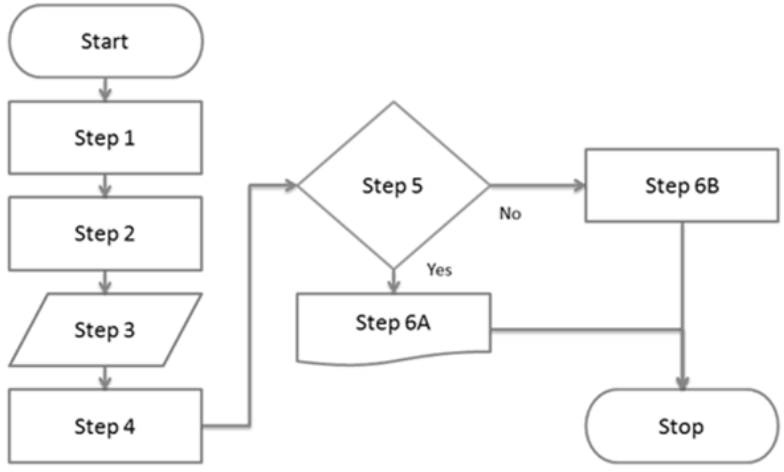
- State
- Object
- Activity
- Class diagram

14 UML diagrams help with modeling like the use case diagram, interaction diagram, class diagram, component diagram, sequence diagram, etc. UML models are important in the IT segment as it becomes the medium of communication between all stakeholders. A UML-based business model can be a direct input to a requirements tool. A UML diagram can be of two type's Behavioral model and the Structural model. A behavioral model tries to give information about what the system does while a structural model will give what the system consists of.



3. Flow chart technique

A flowchart is a visual representation of the sequential flow and control logic of a set of related activities or actions. There are different formats for flowcharts which include Linear, Top-down, and cross-functional (swim lanes). A flow chart can be used for different activities like representing data flows, system interactions, etc. The advantage of using a Flowchart is that it can be easy to read and write even for non-technical team members, and can show the parallel process by function, critical attributes of a process, etc.



4. Data flow diagram

Data flow diagrams show how data is processed by a system in terms of inputs and outputs. Components of the data flow diagram include

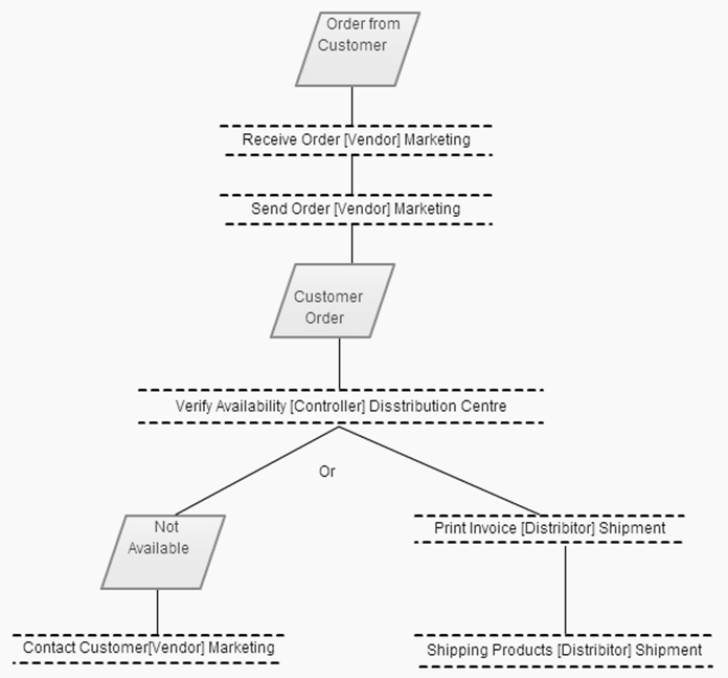
- Process
- Flow
- Store
- Terminator

A logical data flow diagram shows a system's activities while a physical data flow diagram shows a system's infrastructure. A data flow diagram can be designed early in the requirement elicitation process of the analysis phase within the SDLC (System Development Life Cycle) to define the project scope. For easy analysis, a data flow diagram can be drilled down into its sub-processes known as "leveled DFD".

5. Role Activity Diagrams- (RAD)

The role activity diagram is similar to flowchart type notation. In the Role Activity Diagram, role instances are process participants, which have start and end states. RAD requires a deep knowledge of processes or organizations to identify roles. The components of RAD include

- Activities
- External events
- States



Roles group activities together into units of responsibility, according to the set of responsibilities they are carrying out. An activity can be carried out in isolation from a role, or it may require coordination with activities in other roles.

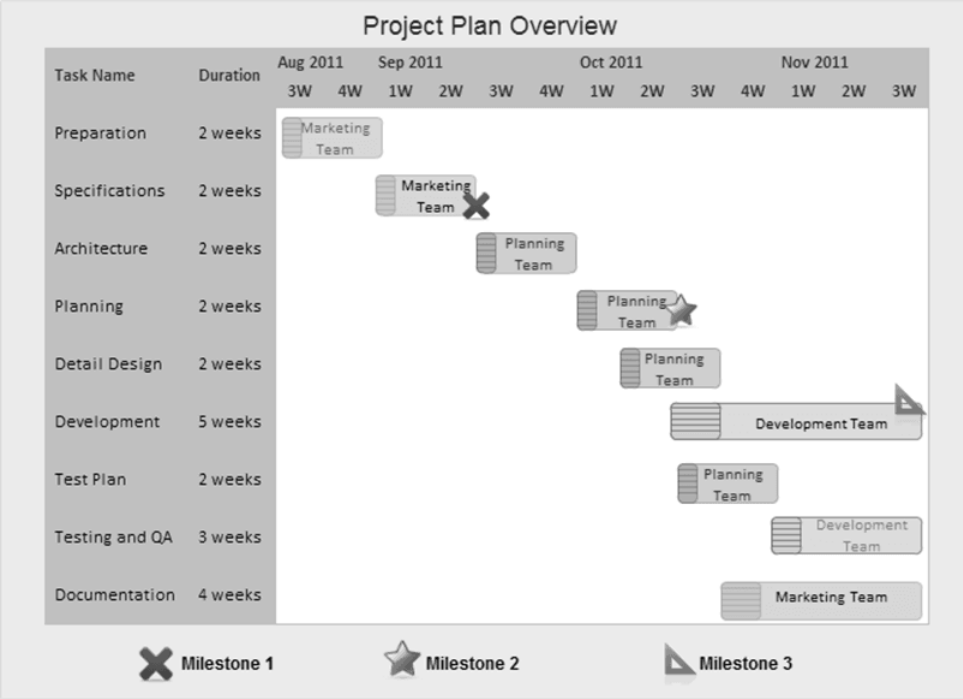
External events are the points at which state changes occur.

States are useful to map activities of a role as it progresses from state to state. When a particular state is reached, it indicates that a certain goal has been achieved.

RAD helps support communication as it is easy to read and presents a detailed view of the process and permitting activities in parallel.

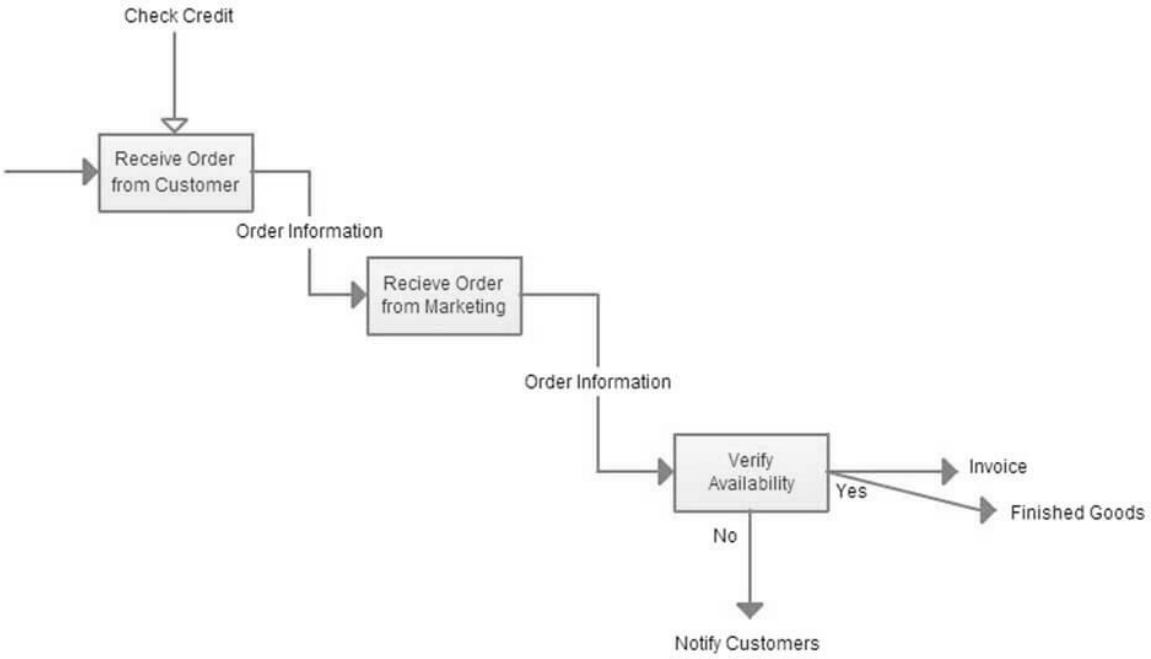
6. Gantt Charts

A Gantt chart is a graphical representation of a schedule that helps to coordinate, plan and track specific tasks in a project. It represents the total period of the object, broken down into increments. A Gantt chart represents the list of all tasks to be performed on the vertical axis while, on the horizontal axis, it lists the estimated activity duration or the name of the person allocated to the activity. One chart can demonstrate many activities.



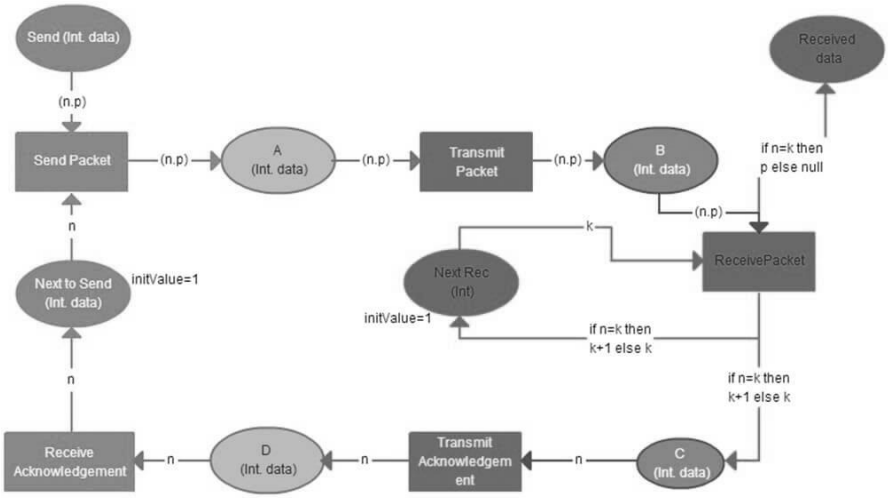
7. IDEF (Integrated Definition for Function Modeling)

IDEF or Integrated Definition for Function Modeling is a common name referred to classes of enterprise modeling languages. It is used for modeling activities necessary to support system analysis, design, or integration. There are about 16 methods for IDEF, the most useful versions of IDEF are IDEF3 and IDEF0.



8. Colored Petri Nets (CPN)

CPN or colored Petri nets are graphically oriented language for specification, verification, design, and simulation of systems. Colored Petri Nets is a combination of graphics and text. Its main components are Places, Transitions, and Arcs.

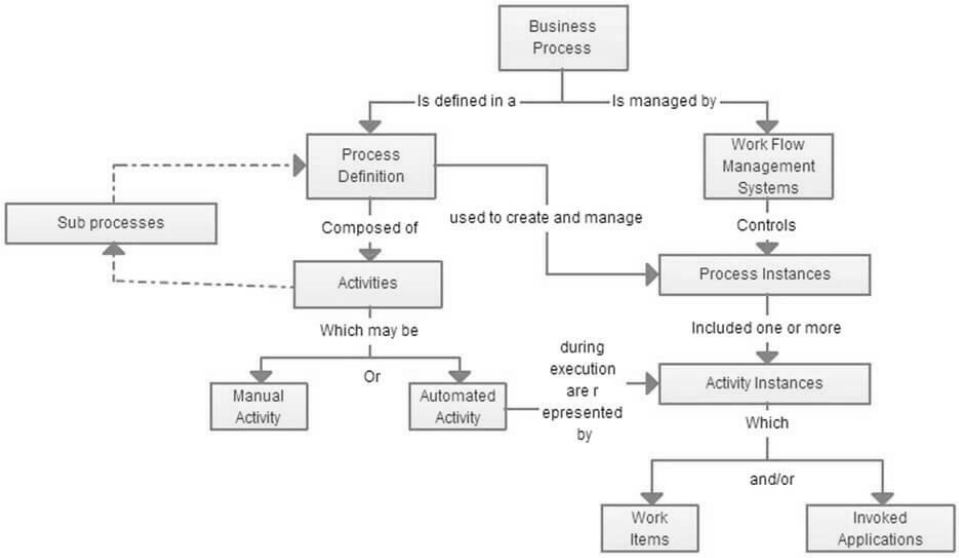


Petri nets objects have specific inscriptions like for

- Places: It has an inscription. Name, Color Set, Initial marking, etc. While
- Transition: lt has inscription like.Name (for identification) and.Guard (Boolean expression consists of some of the variables)
- Arcs: It has an inscription. Arc. When the arc expression is assessed, it yields a multi-set of token colors.

9. Workflow Technique

The workflow technique is a visual diagram that represents one or more business processes to clarify understanding of the processor to make process improvement recommendations. Just like other diagrams like flowcharting, UML activity, and process map, the workflow technique is the oldest and most popular technique. It is even used by BA for taking notes during requirements elicitation. The process comprises four stages



- Information Gathering
- Workflow Modeling
- Business process Modeling
- Implementation, Verification & Execution

10. Object-oriented methods

The object-oriented modeling method uses an object-oriented paradigm and modeling language for designing a system. Its emphasis is on finding and describing the object in the problem domain. The purpose of the object-oriented method is

- To help characterize the system
- To know what are the different relevant objects
- How do they relate to each other
- How to specify or model a problem to create an effective design
- To analyze requirements and their implications

This method applies to the system which has dynamic requirements (changes frequently). It is a process of deriving use cases, activity flow, and events flow for the system. Object-oriented analysis can be done through textual needs, communication with system stakeholders, and vision documents.

The object has a state, and state changes are represented by behavior. So, when the object receives a message, the state changes through behavior.

11. Gap Analysis

Gap Analysis is the technique used to determine the difference between the proposed state and the current state of any business and its functionalities. Does it answer questions like what is the current state of the project? Where do we want to be? etc. Various stages of Gap Analysis include

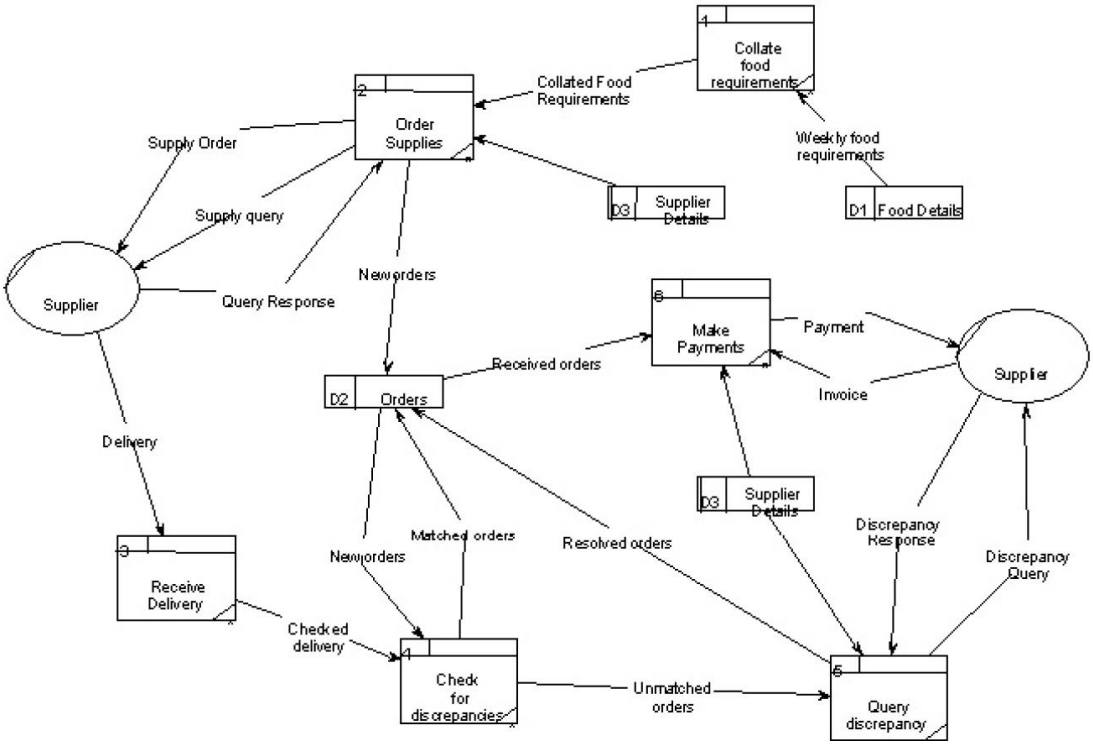
5 DATA-FLOW DIAGRAMS

What are data-flow diagrams?

Data-flow diagrams (DFDs) model a perspective of the system that is most readily understood by users – the flow of information through the system and the activities that process this information.
Data-flow diagrams provide a graphical representation of the system that aims to be accessible to computer specialists and non-specialist users alike. The models enable software engineers, customers, and users to work together effectively during the analysis and specification of requirements. Although this means that our customers are required to understand the modeling techniques and constructs, in data-flow modeling only a limited set of constructs are used, and the rules applied are designed to be simple and easy to follow. These same rules and constructs apply to all data-flow diagrams (i.e., for each of the different software process activities in which DFDs can be used).

An example data-flow diagram

An example of part of a data-flow diagram is given below. Do not worry about which parts of what system this diagram is describing – look at the diagram to get a feel for the symbols and notation of a data-flow diagram.



As can be seen, the DFD notation consists of only four main symbols:

1. Processes — the activities carried out by the system which uses and transform information. Processes are notated as rectangles with three parts, such as “Order Supplies” and “Make Payments” in the above example.

2. Data-flows — the data inputs to and outputs from these activities. Data-flows are notated as named arrows, such as “Delivery” and “Supply Order” in the example above.

3. External entities — the sources from which information flows into the system and the recipients of information leaving the system. External entities are notated as ovals, such as “Supplier” in the example above.

4. Data stores — where information is stored within the system. Datastores are notated as rectangles with two parts, such as “Supplier Details” and “Orders” in the example above. The diagrams are supplemented by supporting documentation including a data dictionary, describing the contents of data-flows and data stores; and process definitions, which provide detailed descriptions of the processes identified in the data-flow diagram.

The benefits of the data-flow diagram

Data-flow diagrams provide a very important tool for software engineering, for several reasons:

- The system scope and boundaries are indicated on the diagrams (more will be described the boundaries of systems and each DFD later in this chapter).
- The technique of decomposition of high-level data-flow diagrams to a set of more detailed diagrams provides an overall view of the complete system, as well as a more detailed breakdown and description of individual activities, where which is appropriate, for clarification and understanding.

Note
Use-case diagrams also provide a partition of a software system into those things which are inside the system and those things which are outside of the system.

Case study
We shall be using the following case study to explore different aspects of data-flow modeling and diagrams.

Video-Rental LTD case study

Video-Rental LTD is a small video rental store. The store lends videos to customers for a fee and purchases its videos from a local supplier.

A customer wishing to borrow a video provides the empty box of the video they desire, their membership card, and payment – payment is always with the credit card used to open the customer account. The customer then returns the video to the store after watching it.

If a loaned video is overdue by a day the customer's credit card is charged, and a reminder letter is sent to them. Each day after that a further card is made, and each week a reminder letter is sent. This continues until either the customer returns the video, or the charges are equal to the cost of replacing the video.

New customers fill out a form with their details and credit card details, and the counter staff give the new customer a membership card. Each new customer's form is added to the customer file.

The local video supplier sends a list of available titles to Video-Rental LTD, who decides whether to send them an order and payment. If an order is sent then the supplier sends the requested videos to the store. For each new video a new stock form is completed and placed in the stock file.

The different kinds (and levels) of data-flow diagrams

Although all data-flow diagrams are composed of the same types of symbols, and the validation rules are the same for all DFDs, there are three main types of data-flow diagrams:

- Context diagrams — context diagrams DFDs are diagrams that present an overview of the system and its interaction with the rest of the "world".

- Level 1 data-flow diagrams — Level 1 DFDs present a more detailed view of the system than context diagrams, by showing the main sub-processes and stores of data that make up the system as a whole.

- Level 2 (and lower) data-flow diagrams — a major advantage of the data-flow modeling technique is that through a technique called "leveling", the detailed complexity of real-world systems can be managed and modeled in a hierarchy of abstractions. Certain elements of any dataflow diagram can be decomposed ("exploded") into a more detailed model a level lower in the hierarchy.

During this unit, we shall investigate each of the three types of diagram in the sequence they are described above. This is both a sequence of increasing complexity and sophistication and also the sequence of DFDs that is usually followed when modeling systems. For each type of diagram we shall first investigate what the features of the diagram are, then we shall investigate how to create that type of diagram. However, before looking at particular kinds of dataflow diagrams, we shall briefly examine each of the symbols from which DFDs are composed.

Elements of data-flow diagrams

Four basic elements are used to construct data-flow diagrams:

- processes
- data-flows
- data stores
- external entities

The rest of this section describes each of the four elements of DFDs, in terms of their purpose, how the element is notated, and the rules associated with how the element relates to others in a diagram.

Notation and software

A number of different notations exist for depicting these elements, although it is only the shape of the symbols which vary in each case, not the underlying logic. This unit uses the Select SSADM notation in the description and construction of data-flow diagrams.

As data-flow diagrams are not a part of the UML specification, ArgoUML and Umbrello do not support their creation. However, Dia is free software available for both Windows and Ubuntu which does support data-flow diagrams.

Processes

Purpose

Processes are the essential activities, carried out within the system boundary, that use information. A process is represented in the model only where the information which provides the input into the activity is manipulated or transformed in some way so that the data flowing out of the process is changed compared to that which flowed in.

The activity may involve capturing information about something that the organisation is interested in, such as a customer or a customer's maintenance call. It may be concerned with recording changes to this information, a change in a customer's address for example. It may require calculations to be carried out, such as the quantity left in stock following the allocation of stock items to a customer's job; or it may involve validating information, such as checking that faulty equipment is covered by a maintenance contract.

Notation

Processes are depicted with a box, divided into three parts.

The notation for a process

The top left-hand box contains the process number. This is simply for identification and reference purposes and does not in any way imply priority and sequence.

The main part of the box is used to describe the process itself, giving the processing performed on the data it receives.

The smaller rectangular box at the bottom of the process is used in the Current Physical Data-Flow Diagram to indicate the location where the processing takes place. This may be the physical location of the Customer Services Department or the Stock Room, for example. However, it is more often used to denote the staff role responsible for performing the process. For example, Customer Services, Purchasing, Sales Support, and so on.

Rules

The rules for processes are:

- Process names should be an imperative verb specific to the activity in question, followed by a pithy and meaningful description of the object of the activity. Create Contract, or Schedule Jobs, as opposed to using very general or non-specific verbs, such as Update Customer Details or Process Customer Call.
- Processes may not act as data sources or sinks. Data flowing into a process must have some corresponding output, which is directly related to it. Similarly, data flowing out of a process must have some corresponding input to which it is directly related.
- Normally only processes that transform system data are shown on data-flow diagrams. Only where an enquiry is central to the system is it included.
- Where a process is changing data from a data store, only the changed information flow to the data store (and not the initial retrieval from the data store) is shown on the diagram.
- Where a process is passing information from a data store to an external entity or another process, only the flow from the data store to the process is shown on the diagram.

Data-flows

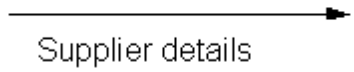
Purpose

A data flow represents a package of information flowing between two objects in the data-flow diagram. Data-flows are used to model the flow of information into the system, out of the system, and between elements within the system.

Occasionally, a data flow is used to illustrate information flows between two external entities, which is, strictly speaking, outside of the system boundaries. However, knowledge of the transfer of information between external entities can sometimes aid understanding of the system under investigation, in which case it should be depicted on the diagram.

Notation

A data flow is depicted on the diagram as a directed line drawn between the source and recipient of the data flow, with the arrow depicting the direction of flow.



The directed line is labeled with the data-flow name, which briefly describes the information contained in the flow. This could be a Maintenance Contract, Service Call Details, Purchase Order, and so on.
Data-flows between external entities are depicted by dashed, rather than unbroken, lines.

Rules

The rules for drawing data flows are:

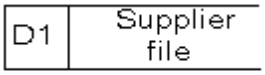
- Information always flows to or from a process; the other end of the flow may be an external entity, a data store or another process. An occasional exception to this rule is a data-flow between two external entities.
- Datastores may not be directly linked by data flows; information is transformed from one stored state to another via a process.
- Information may not flow directly from a data store to an external entity, nor may it flow from an external entity directly to a data store. This communication and receipt of information stored in the system always take place via a process.
- The sources (where data of interest to the system is generated without any corresponding input) and sinks (where data is swallowed up without any corresponding output) of data-flows are always represented by external entities.
- When something significant happens to a data flow, as a result of a process acting on it, the label of the resulting data flow should reflect its transformed status. For example, “Telephoned Service Call” becomes “Service Call Form” once it has been logged.

Data stores

Purpose

A datastore is a place where data is stored and retrieved within the system. This may be a file, Customer Contracts file for example, a catalog or reference list, Options Lists for example, a log book such as the Job Book, and so on.

Notation
A datastore is represented in the data-flow diagram by a long rectangle, containing two locations.



The small left-hand box is used for the identifier, which comprises a numerical reference prefixed by a letter.

The main area of the rectangle is labeled with the name of the data store. Brief names are chosen to reflect the content of the data store.

Rules

The rules for representing data stores are:

- One convention that could be used is to determine the letter identifying a data store by the store's nature.
- “M” is used where a manual data store is being depicted.
- “D” is used where it is a computer-based data store.
- “T” is used where a temporary datastore is being represented.
- Datastores may not act as data sources or sinks. Data flowing into a data store must have some corresponding output and vice versa.
- Because of their function in the storage and retrieval of data, data stores often provide input dataflows to receive output flows from several processes. For the sake of clarity and to avoid crisscrossing of data flows in the data-flow diagram, a single data store may be included in the diagram at more than one point. Where the depiction of a data store is repeated in this way, this is signified by drawing a second vertical line along the left-hand edge of the rectangle for each occurrence of the data store.

External entities

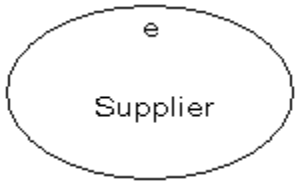
Purpose

External entities are entities outside of the system boundary which interact with the system, in that they send information into the system or receive information from it. External entities may be external to the whole organization — as in Customer and Supplier in our running example; or just external to the application area where users' activities are not directly supported by the system under investigation. Accounts and Engineering are shown as external entities as they are recipients of information from the system. Sales also provide input to the system.

External entities are often referred to as sources and sinks. All information represented within the system is sourced initially from an external entity. Data can leave the system only via an external entity

Notation

External entities are represented on the diagram as ovals drawn outside of the system boundary, containing the entity name and an identifier.



Names consist of a singular noun describing the role of the entity. Above the label, a lower case letter is used as the identifier for reference purposes.

Rules

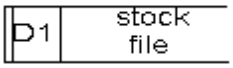
The rules associated with external entities are:

- Each external entity must communicate with the system in some way, thus there is always a data flow between an external entity and a process within the system.
- External entities may provide and receive data from several processes. It may be appropriate, for the sake of clarity and to avoid crisscrossing data flows, to depict the same external entity at a number of points on the diagram. Where this is the case, a line is drawn across the left corner of the ellipse, for each occurrence of the external entity on the diagram. Customer is duplicated in this way in our example.

Multiple copies of entities and data stores on the same diagram

At times a diagram can be made much clearer by placing more than one copy of an external entity or datastore in different places — this can avoid a tangle of crossing data-flows.
Where more than one copy of an external entity appears on a diagram it has a cut off corner in the top left, such as below:

When more than one copy of a data store appears on a diagram it has a cut-off left side, such as below:

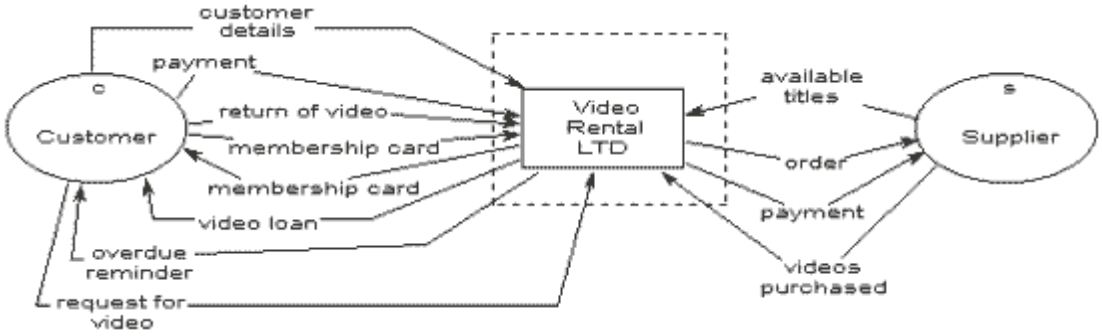


Context diagrams

What is a context diagram?
The context diagram is used to establish the context and boundaries of the system to be modelled: which things are inside and outside of the system being modeled, and what is the relationship of the system with these external entities.

A context diagram sometimes called a level 0 data-flow diagram, is drawn in order to define and clarify the boundaries of the software system. It identifies the flows of information between the system and external entities. The entire software system is shown as a single process.

A possible context diagram for the Video-Rental LTD case study is shown below.



The process of establishing the analysis framework by drawing and reviewing the context diagram inevitably involves some initial discussions with users regarding problems with the existing system and the specific requirements for the new system. These are formally documented along with any specific system requirements identified in previous studies.

Having agreed on the framework, a detailed investigation of the current system must be planned. This involves identifying how each of the areas included within the scope will be investigated. This could be by interviewing users, providing questionnaires to users or clients, studying existing system documentation and procedures, observation and so on. Key users are identified and their specific roles in the investigation are agreed upon.

Constructing a context diagram

To produce the context diagram and agree on the system scope, the following must be identified:

- external entities
- data-flows

You may find the following steps useful:

1. Identify data flows by listing the major documents and information flows associated with the system, including forms, documents, reference material, and other structured and unstructured information (emails, telephone conversations, information from external systems, etc.).
2. Identify external entities by identifying sources and recipients of the data-flows, which lie outside of the system under investigation. The actors and any use case models you have created may often be external entities.
3. Draw and label a process box representing the entire system.
4. Draw and label the external entities around the outside of the process box.
5. Add the data flow between the external entities and the system box. Where documents and other packets of information flow entirely within the system, these should be ignored from the point of view of the context diagram – at this stage, they are hidden within the process box.

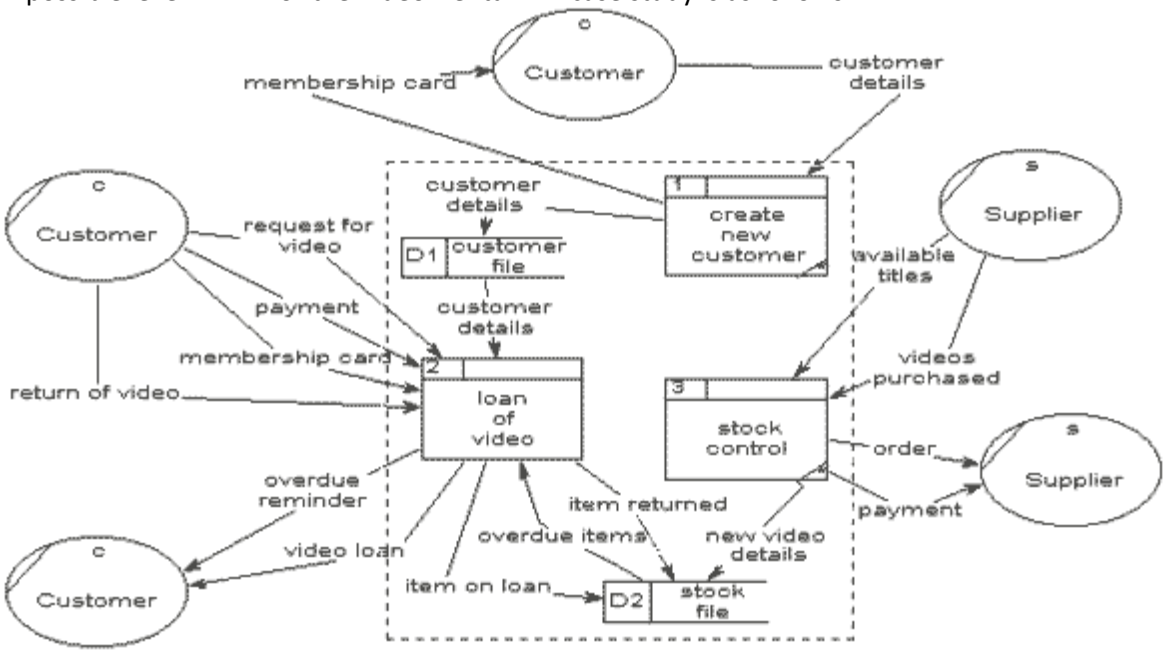
This system boundary and details depicted in the context diagram should then be discussed (and updated if necessary) in consultation with your customers until an agreement is reached.

Having defined the system boundary and scope, the areas for investigation will be determined, and appropriate techniques for investigating each area will need to be decided.

Level 1 data-flow diagrams

What is a level 1 DFD?
As described previously, context diagrams (level 0 DFDs) are diagrams where the whole system is represented as a single process. A level 1 DFD notates each of the main sub-processes that together form the complete system. We can think of a level 1 DFD as an "exploded view" of the context diagram.

A possible level 1 DFD for the Video-Rental LTD case study is as follows:



Notice that the external entities have been included in this diagram, but outside of the rectangle represents the boundary of this diagram (i.e., the system boundary). It is not necessary to always show the external entities on level 1 (or lower) DFDs, however, you may wish to do so to aid clarity and understanding.

We can see that on this level 1 DFD there are several data stores and data-flows between processes and the data stores.

It is important to notice that the same data flows to and from the external entities appear on this level 1 diagram and the level 0 context diagram. Each time a process is expanded to a lower level, the lower level diagram must show all the same data flows into, and out of the higher level process it expands.

Constructing level 1 DFDs

If no context diagram exists, first create one before attempting to construct the level 1 DFD (or construct the context diagram and level 1 DFD simultaneously). The following steps are suggested to aid the construction of Level 1 DFD:

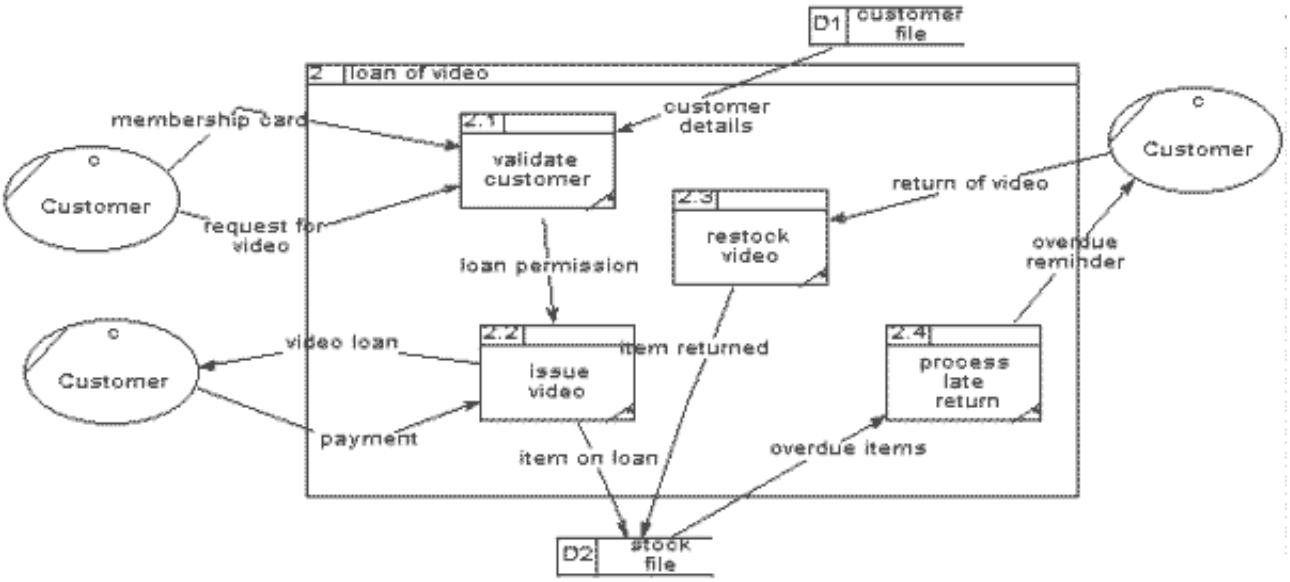
1. Identify processes. Each data flow into the system must be received by a process. For each data flow into the system examine the documentation about the system and talk to the users to establish a plausible process of the system that receives the data flow. Each process must have at least one output data flow. Each output data-flow of the system must have been sent by a process; identify the processes that send each system output.
2. Draw the data flows between the external entities and processes.
3. Identify data stores by establishing where documents/data need to be held within the system. Add the data stores to the diagram, labeling them with their local name or description.
4. Add data-flows flowing between processes and data stores within the system. Each data store must have at least one input data flow and one output data flow (otherwise data may be stored, and never used, or a store of data must have come from nowhere). Ensure every datastore has input and output data-flows to system processes. Most processes are normally associated with at least one data store.
5. Check diagram. Each process should have an input and an output. Each data store should have an input and an output. Check the system details to see if any process appears to be happening for no reason (i.e., some "trigger" data-flow is missing, which would make the process happen).

Decomposing diagrams into level 2 and lower hierarchical levels

What is a level 2 (or lower) DFD?
We have already seen how a level 0 context diagram can be decomposed (exploded) into a level 1 DFD. In DFD modeling terms we talk of the context diagram as the "parent" and the level 1 diagram as the "child".

This same process can be applied to each process appearing within a level 1 DFD. A DFD that represents a decomposed level 1 DFD process is called a level 2 DFD. There can be a level 2 DFD for each process that appears in the level 1 DFD.

A possible level 2 DFD for process “2: Loan of video” of the level 1 DFD is as follows:



Note, that every data flow into and out of the parent process must appear as part of the child DFD. The numbering of processes in the child DFD is derived from the number of the parent process – so all processes in the child DFD of process 2, will be called 2.X (where X is the arbitrary number of the process on the level 2 DFD). Also there are no new data-flows into or out of this diagram – this kind of data-flow validation is called balancing.

Look at the rectangular boundary for this level 2 DFD. Outside the boundary is the external entity “Customer”. Also outside the boundary are the two data stores – although these data stores are inside the system (see the level 1 DFD), they are outside the scope of this level 2 DFD.

Constructing level 2 (and lower) DFDs — functional decomposition

The level 1 data-flow diagram provides an overview of the system. As the software engineers' understanding of the system increases it becomes necessary to expand most of the level 1 processes to a second or even third level in order to depict the detail within it. Decomposition is applied to each process on the level 1 diagram for which there is enough detail hidden within the process. Each process on the level 2 diagrams also needs to be checked for possible decomposition, and so on.

A process box that cannot be decomposed further is marked with an asterisk in the bottom right-hand corner. A brief narrative description of each bottom-level process should be provided with the data flow diagrams to complete the documentation of the data-flow model. These make up part of the process definitions which should be supplied with the DFD.

Each process on the level 1 diagram is investigated in more detail, to give a greater understanding of the activities and data flows. Normally processes are decomposed where:

- There are more than six data flows around the process
- The process name is complex or very general which indicates that it incorporates a number of activities.

The following steps are suggested to aid the decomposition of a process from one DFD to a lower level DFD. As you can see they are very similar to the steps for creating a level 1 DFD from a context diagram:

1. Make the process box on the level 1 diagram the system boundary on the level 2 diagram that decomposes it. This level 2 diagram must balance with its “parent” process box — the dataflows to and from the process on the level 1 diagram will all become data flows across the system boundary on the level 2 diagram. The sources and

recipients of data-flows across the level 2 system boundary are drawn outside the boundary and labeled exactly as they are on the level 1 diagram. Note that these sources and recipients may be data stores as well external entities or other processes.This is because a data store in a level 1 diagram will be outside the boundary of a level 2 process that sends or receives data flows to/from the data store.

2. Identify the processes inside the level 2 system boundary and draw these processes and their data-flows. Remember, each data flow into and out of the level 2 system boundary should be to/ from a process. Using the results of the more detailed investigation, filter out and draw the processes at the lower level that send and receive information both across and within the level 2 system boundary. Use the level numbering system to number sub-processes so that, for example, process 4 on the level 1 diagram is decomposed to sub-processes 4.1, 4.2, 4.3 … on the level 2 diagram.
3. Identify any data stores that exist entirely within the level 2 boundary, and draw these data stores.
4. Identify data flows between the processes and data stores that are entirely within the level 2 system boundary. Remember, every data store inside this boundary should have at least one input and one output data flow.
5. Check the diagram. Ensure that the level 2 data-flow diagram does not violate the rules for dataflow diagram constructs.

6 SOFTWARE DESIGN BASICS

Software Design Basics

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas, for coding and implementation, there is a need for more specific and detailed requirements in software terms. The output of this process can directly be used in implementation in programming languages. Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from the problem domain to the solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

Software Design Levels

Software design yields three levels of results:

- **Architectural Design** - The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get an idea of the proposed solution domain.
- **High-level Design**- The high-level design breaks the 'single entity-multiple component' concept of architectural design into a less-abstracted view of sub-systems and modules and depicts their interaction with each other. The high-level design focuses on how the system along with all of its components can be implemented in the form of modules. It recognizes the modular structure of each sub-system and their relation and interaction with each other.
- **Detailed Design**- Detailed design deals with the implementation part of what is seen as a system and its subsystems in the previous two designs. It is more detailed about modules and their implementations. It defines the logical structure of each module and its interfaces to communicate with other modules.

Modularization

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out the task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.
Modular design unintentionally follows the rules of the 'divide and conquer' problem-solving strategy this is because there are many other benefits attached to the modular design of the software.

Advantages of modularization:

- Smaller components are easier to maintain
- Programs can be divided based on functional aspects
- The desired level of abstraction can be brought into the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from a security aspect

Concurrency

Back in time, all software are meant to be executed sequentially. By sequential execution, we mean that the coded instruction will be executed one after another implying only one portion of the program being activated at any given time. Say, a software has multiple modules, then only one of all the modules can be found active at any time of execution.
In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules, and executing them in parallel. In other words, concurrency provides the capability of the software to execute more than one part of code in parallel to each other. The programmers and designers must recognize those modules, which can be made parallel execution.

Example

The spell check feature in a word processor is a module of the software, which runs alongside the word processor itself.

Coupling and Cohesion

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together to achieve some tasks. They are though, considered as a single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

Cohesion

Cohesion is a measure that defines the degree of intradependability within elements of a module. The greater the cohesion, the better the program design. There are seven types of cohesion, namely –

- **Co-incidental cohesion** - It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not accepted.
- **Logical cohesion** - When logically categorized elements are put together into a module, it is called logical cohesion.
- **Temporal Cohesion** - When elements of the module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- **Procedural cohesion** - When elements of the module are grouped, which are executed sequentially to perform a task, it is called procedural cohesion.
- **Communicational cohesion** - When elements of the module are grouped, which are executed sequentially and work on the same data (information), it is called communicational cohesion.
- **Sequential cohesion** - When elements of a module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- **Functional cohesion** - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of a module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

Coupling

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program. There are five levels of coupling, namely -

- **Content coupling** - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
- **Common coupling**- When multiple modules have read and write access to some global data, it is called common or global coupling.

- **Control coupling**- Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
- **Stamp coupling**- When multiple modules share a common data structure and work on a different part of it, it is called stamp coupling.
- **Data coupling**- Data coupling is when two modules interact with each other utilizing passing data (as a parameter). If a module passes data structure as a parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

Design Verification

The output of the software design process is design documentation, pseudo-codes, detailed logic diagrams, process diagrams, and a detailed description of all functional or non-functional requirements.
The next phase, which is the implementation of software, depends on all outputs mentioned above.
It then becomes necessary to verify the output before proceeding to the next phase. The early any mistake is detected, the better it is or it might not be detected until testing of the product. If the outputs of the design phase are informal notation forms, then their associated tools for verification should be used otherwise a thorough design review can be used for verification and validation.
With a structured verification approach, reviewers can detect defects that might be caused by overlooking some conditions. A good design review is important for good software design, accuracy, and quality.

Software Design Strategies

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find the optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.
There are multiple variants of software design. Let us study them briefly:

Structured Design

Structured design is a conceptualization of a problem into several well-organized elements of a solution. It is concerned with the solution design. The benefit of the structured design is, that it gives a better understanding of how the problem is being solved. The structured design also makes it simpler for a designer to concentrate on the problem more accurately.
Structured design is mostly based on the 'divide and conquer strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.
The small pieces of the problem are solved through solution modules. The structured design emphasizes that these modules be well organized to achieve a precise solution.
These modules are arranged in a hierarchy. They communicate with each other. A well-structured design always follows some rules for communication among multiple modules, namely -

Cohesion - a grouping of all functionally related elements.
Coupling - communication between different modules.

A well-structured design has high cohesion and low coupling arrangements.

Function Oriented Design

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing a significant task in the system. The system is considered as a top view of all functions. Function-oriented design inherits some properties of structured design where divide and conquer methodology is used. This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and its operation. These functional modules can share information among themselves using information passing and using information available globally.
Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function-oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process

- The whole system is seen as how data flows in the system employing a data flow diagram.
- DFD depicts how functions change data and the state of the entire system.
- The entire system is logically broken down into smaller units known as functions based on their operation in the system.
- Each function is then described at large.

Object-Oriented Design

Object-oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategy focuses on entities and their characteristics. The whole concept of software solutions revolves around the engaged entities.

Let us see the important concepts of Object-Oriented Design:

- Objects - All entities involved in the solution design are known as objects. For example, people, banks, companies, and customers are treated as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.
- Classes - A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which define the functionality of the object. In the solution design, attributes are stored as variables, and functionalities are defined through methods or procedures.
- Encapsulation - In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together but also restricts access to the data and methods from the outside world. This is called information hiding.
- Inheritance - OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and reuse allowed variables and
- methods from their immediate superclasses. This property of OOD is known as inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.
- Polymorphism - OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, a respective portion of the code gets executed.

Design Process

The software design process can be perceived as a series of well-defined steps. Though it varies according to the design approach (function-oriented or object-oriented, It may have the following steps involved:

- A solution design is created from a requirement or previously used system and/or system sequence diagram.
- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
- Class hierarchy and relation among them is defined.
- The application framework is defined.

Software Design Approaches

Here are two general approaches for software design:

Top-Down Design

We know that a system is composed of more than one subsystem and it contains several components. Further, these sub-systems and components may have their own set of sub-systems and components and create a hierarchical structure in the system.

The top-down design takes the whole software system as one entity and then decomposes it to achieve more than one subsystem or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of the system in the top-down hierarchy is achieved. Top-down design starts with a generalized model of a system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

Bottom-up Design

The bottom-up design model starts with the most specific and basic components. It proceeds with composing higher levels of components by using basic or lower-level components. It keeps creating higher-level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased. A bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.
Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

7 SOFTWARE IMPLEMENTATIONS

Software Implementation

Structured Programming

In the process of coding, the lines of code keep multiplying, thus, the size of the software increases. Gradually, it becomes next to impossible to remember the flow of a program. If one forgets how software and its underlying programs, files, and procedures are constructed it then becomes very difficult to share, debug and modify the program. The solution to this is structured programming. It encourages the developer to use subroutines and loops instead of using simple jumps in the code, thereby bringing clarity to the code and improving its efficiency Structured programming also helps programmers to reduce coding time and organize code properly.

Structured programming states how the program shall be coded. Structured programming uses three main concepts:

- **Top-down analysis** - Software is always made to perform some rational work. This rational work is known as a problem in the software parlance. Thus we must understand how to solve the problem. Under top-down analysis, the problem is broken down into small pieces where each one has some significance. Each problem is individually solved and steps are clearly stated about how to solve the problem.
- **Modular Programming** - While programming, the code is broken down into a smaller group of instructions. These groups are known as modules, subprograms, or subroutines. Modular programming is based on the understanding of top-down analysis. It discourages jumps using 'goto' statements in the program, which often makes the program flow nontraceable. Jumps are prohibited and modular format is encouraged in structured programming.
- **Structured Coding** - About top-down analysis, structured coding sub-divides the modules into further smaller units of code in the order of their execution. Structured programming uses control structure, which controls the flow of the program, whereas structured coding uses control structure to organize its instructions in definable patterns.

Functional Programming

Functional programming is a style of programming language, which uses the concepts of mathematical functions. A function in mathematics should always produce the same result on receiving the same argument. In procedural languages, the flow of the program runs through procedures, i.e. the control of the program is transferred to the called procedure. While control flow is transferring from one procedure to another, the program changes its state.

In procedural programming, a procedure can produce different results when it is called with the same argument, as the program itself can be in a different state while calling it. This is a property as well as a drawback of procedural programming, in which the sequence or timing of the procedure execution becomes important.

Functional programming provides means of computation as mathematical functions, which produce results irrespective of program state. This makes it possible to predict the behavior of the program.

Functional programming uses the following concepts:

- First-class and High-order functions - These functions can accept another function as an argument or return other functions as results.
- Pure functions - These functions do not include destructive updates, that is, they do not affect any I/O or memory and if they are not in use, they can easily be removed without hampering the rest of the program.
- Recursion - Recursion is a programming technique where a function calls itself and repeats the program code in it unless some pre-defined condition matches. Recursion is the way of creating loops in functional programming.
- Strict evaluation - It is a method of evaluating the expression passed to a function as an argument. Functional programming has two types of evaluation methods, strict (eager) and non-strict (lazy). Strict evaluation always evaluates the expression before invoking the function. Nonstrictevaluation does not evaluate the expression unless it is needed.
- λ-calculus - Most functional programming languages use λ- calculus as their type systems. λ-expressions are executed by evaluating them as they occur. Common Lisp, Scala, Haskell, Erlang, and F# are some examples of functional programming languages.

Programming style

Programming style is a set of coding rules followed by all the programmers to write the code. When multiple programmers work on the same software project, they frequently need to work with the program code written by some other developer. This becomes tedious or at times impossible, if all developers do not follow some standard programming style to code the program.

An appropriate programming style includes using function and variable names relevant to the intended task, using well-placed indentation, commenting code for the convenience of the reader, and overall presentation of code. This makes the program code readable and understandable by all, which in turn makes debugging and error solving easier. Also, proper coding style helps ease documentation and updation.

Coding Guidelines
The practice of coding style varies with organizations, operating systems, and the language of coding itself.

The following coding elements may be defined under the coding guidelines of an organization:

- Naming conventions - This section defines how to name functions, variables, constants, and global variables.
- Indenting - This is the space left at the beginning of a line, usually 2-8 whitespace or a single tab.
- Whitespace - It is generally omitted at the end of the line.
- Operators - Defines the rules of writing mathematical, assignment, and logical operators. For example, the assignment operator '=' should have space before and after it, as in "x = 2".
- Control Structures - The rules of writing if-then-else, case switch, while-until, and for control flow statements solely and in nested fashion.
- Line length and wrapping - Defines how many characters should be there in one line, mostly a line is 80 characters long. Wrapping defines how a line should be wrapped if is too long.
- Functions - This defines how functions should be declared and invoked, with and without parameters.
- Variables - This mentions how variables of different data types are declared and defined.
- Comments - This is one of the important coding components, as the comments included in the code describe what the code does and all other associated descriptions. This section also helps create help documentation for other developers.