



University of Gondar
College of Informatics
Department of Information Systems
System Analysis and Design Module

Prepared by Nigus and Lomi

Table of Contents

Chapter One	1
System Development.....	1
SDLC MODEL.....	7
Waterfall Methodology	7
Spiral SDLC Model	9
The Best Prioritization Techniques	12
Eisenhower Matrix: 4 Quadrants time management	13
3. ABCDE method	14
Gantt chart used for	15
Build and manage a comprehensive project	15
Determine logistics and task dependencies	15
Monitor progress of a project	15
The benefits of using a Gantt chart	15
How to use Gantt charts?.....	16
Elements of basic Gantt chart	16
Program Evaluation Review Technique (PERT)	17
Chapter Two.....	18
Object Orientation the new software paradigm.....	18
Chapter Three.....	23
Understanding the Basics Object oriented concepts	23
Chapter Four	39
Gathering user requirements	39
Chapter Five	54
Determining What to Build: OO Analysis	54
5.1 Introduction.....	54
5.2 System Use Case Modeling	55
5.3 Use Case Documentation (Use Case Description)	56
5.4 Sequence Diagram	59
5.5 Conceptual Modeling: Class Diagram.....	61
5.6 Activity Diagram	64
5.7 User Interface Prototyping Evolving Sour Supplementary Specification	65
5.9 Organizing your Models with Packages	67
Chapter Six.....	68
Determining How to Build Your System: OO Design.....	68

6.1 Layering Your Models.....	68
6.2 Applying Design Patterns Effectively	71
Chapter Seven	73
Object Oriented Testing and Maintenance	73
8.1 Introduction.....	73
8.2 Maintenance of Systems	73
8.3 Managing Quality Assurance.....	74
8.4. Testing Plans.....	75

Chapter One

System Development

System development methodologies

Information systems development (ISD) is the process that, through process management, analysis, design, implementation, introduction, and continuous support, some collective work activity is made possible by new information-technological means. Information systems are an outgrowth of a process of organizational problem solving.

Information systems developed due to:

- Problems, real or anticipated undesirable situations that hinder the organization from achieving their goals and that require corrective actions.
- Opportunities a chance to improve an organization even in the absence of an identified problem.
- Directives a new requirement that is imposed by management, government, or some external influence.

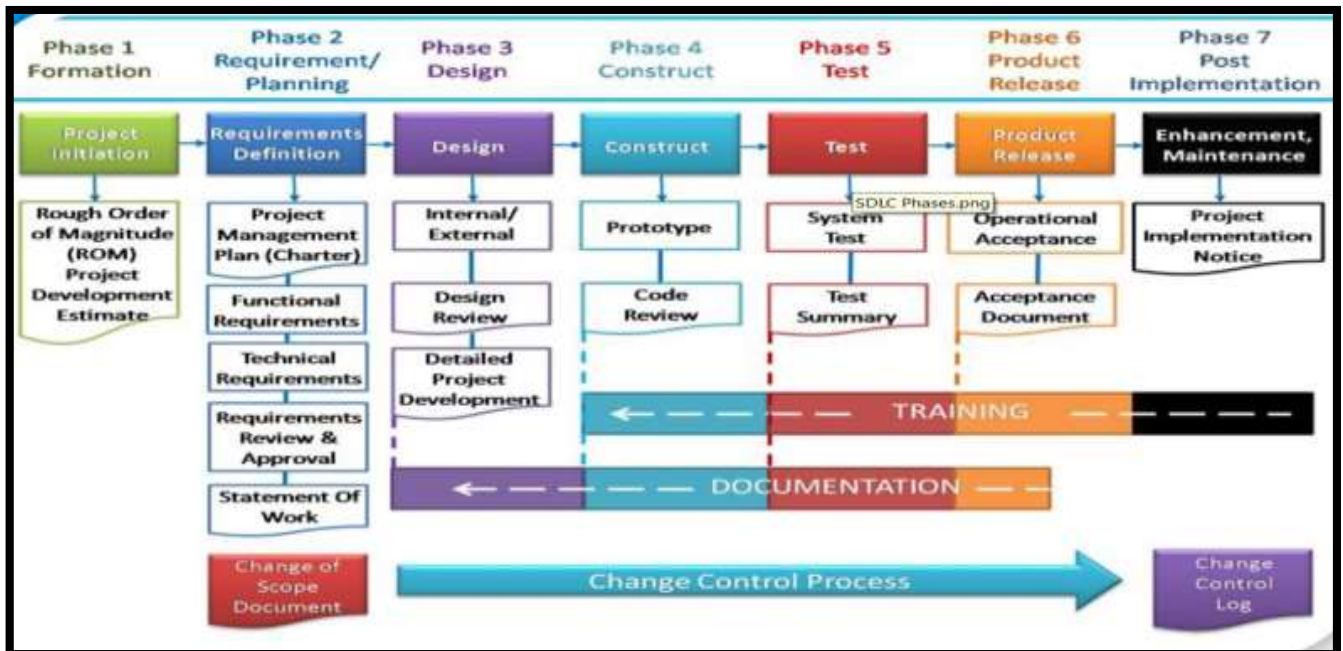
An information system is built as a solution to some type of problem or set of problems the organization perceives it is facing. The problem might be one in which managers and employees realize that the organization is not performing as well as expected, or it may come from the realization that the organization should take advantage of new opportunities to perform more successfully.

Systems development is the process of defining, designing, testing, and implementing a new software application or program. It could include the internal development of customized systems, the creation of database systems, or the acquisition of third party developed software. Written standards and procedures must guide all information systems processing functions. The organization's management must define and implement standards and adopt an appropriate system development life cycle methodology governing the process of developing, acquiring, implementing, and maintaining computerized information systems and related technology.

System development methodology is a methodology for systematically organizing the best ways to develop systems efficiently. It includes, for example, descriptions of work to be performed at each stage of the development process and drafted documents. Multiple methodologies—which differ according to viewpoint—are available. In terms of the development process, some example models are water-fall development, RAD, spiral development, iterative, agile-software development. And in terms of the design approach, some example methodologies are the process-oriented approach (POA), the data-oriented approach (DOA), the object-oriented approach (OOA), and the service-oriented approach (SOA).

Software/system development life cycle(SDLC)

This methodology was first developed in the 1960s to manage the large software projects associated with corporate systems running on mainframes. It is a very structured and risk-averse methodology designed to manage large projects that included multiple programmers and systems that would have a large impact on the organization.



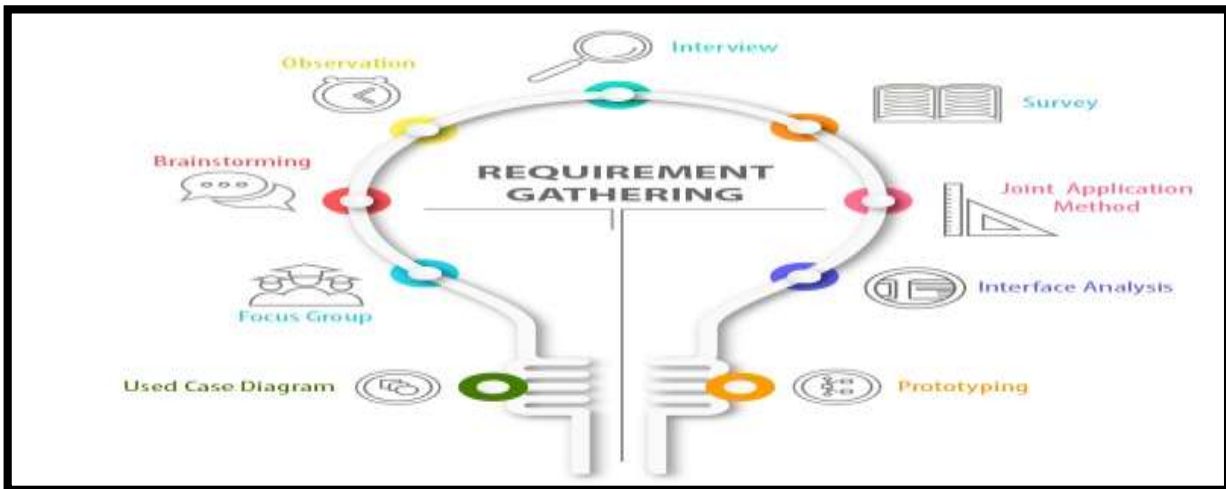
SDLC Process

Most system development methodologies can be grouped into one of four main which are listed above namely Water fall, RAD, Spiral and agile categories:

Project Initiation

This is the first stage in the Software Development Life Cycle where the project is initiated. The high-level scope, problems and solutions are determined, and planning is carried out accordingly for other stages. Other components that are to be considered in this stage are Resources, time/schedules, milestones, cost, business benefits and deadlines. In the case of enhancements to existing projects, the strengths and weaknesses of the current software are studied, and the improvements are set as a goal, along with the collected requirements. In the case of enhancements to existing projects, the strengths and weaknesses of the current software are studied, and the improvements are set as a goal, along with the collected requirements.

Requirements Gathering



Business requirements are gathered via meetings with project managers and stakeholders.

Identify.

- “Who will use the system”
- “How the system should work “
- “What should be the input & output of the system”

Analyze requirements for validity & incorporation of requirements

Finally, prepare Requirement specification document

The requirements are of the type:

- Functional Requirements
- Non-functional Requirements

The end-user requirements from the customer and other stakeholders (salespeople, domain/industry experts, etc.) are collected.

Requirements are gathered using the following techniques:

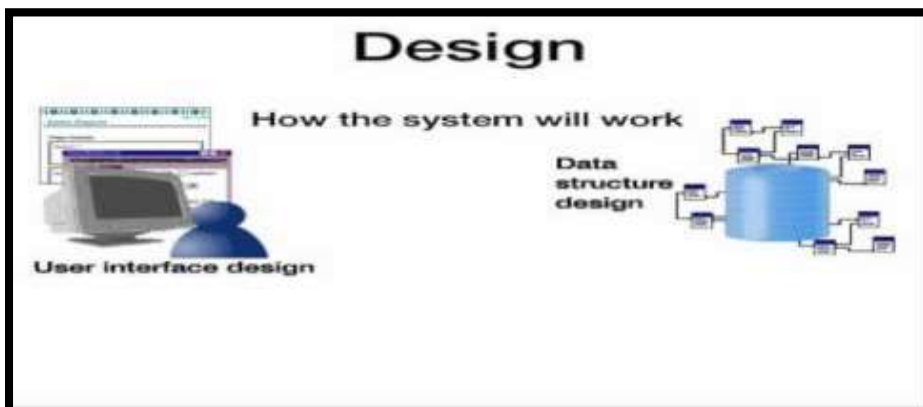
- By conducting Interviews
- By conducting Workshops
- By conducting Surveys and questionnaires
- Focus Groups
- Observations / time study
- By conducting Brainstorming Sessions
- Document Analysis (Ex: Regulatory requirements)
- Mind Mapping
- Benchmarks

Analysis

The Analysis Phase is where you break down the deliverables in the high-level Project Charter into the more detailed business requirements. The Analysis Phase is also the part of the project where you identify the overall direction that the project will take through the creation of the project strategy documents.

- Understand the business need and processing needs
- Gather, analyze, and validate the information.
- Define the requirements and prototypes for the new system.
- Evaluate the alternatives and prioritize the requirements.
- Examine the information needs of end-user and enhances the system goal.
- A Software Requirement Specification (SRS) document is used in the analysis phase, which specifies the software, hardware, functional, and network requirements of the system is prepared at the end of this phase.
- In this, every achievable requirement is analyzed and documented as Software Requirements Specifications (SRS) or Functional Requirements Specifications (FRS).
- This is effectively manageable for all the resources (developers, designers, testers, project managers and any other possible roles) to work on the chunks at all the stages in the Software Development Life Cycle.
- In many cases, a requirement gathering, and analysis can be carried out at the same time.

System Design



This is the stage which stated, “How to achieve what is needed?”

- Software Requirements Specifications (SRS) are now transformed to the system design plan, which contains a detailed and complete set of specifications, commonly known as “Design Specification”.
- Prepare design of network, databases, application, system interfaces, user interfaces, system and software design from software requirement specification.

- OV

t T

vil

- 440



rit.

- 5

- This stage is used to validate whether the application addresses all User Requirements, technical performance.
- This is performed by the testing team, and the focus is to find the defects.
- During test case execution, all the error found which are reported in the test management tool and the decision of considering the defect as Valid or Invalid depends on developers.
- Each defect that is found will have to go through the Defect Life Cycle in the defect management tool.
- Again, the testing approach that the project choose depends on various factors: the complexity of the project, the team's capability, time, etc.

Development or implementation



Write detailed user documentation and provide training for the system user.

- Once the testing is completed and, there are no open high priority issues, then comes the time to deploy the build to the Production environment. This is the environment which is accessible by real users. Real users can then use the software as per their needs.
- Deploying the build to production can be a complicated process. If the project is an existing application, technology migration is being carried out etc, it can be an extensive procedure.
- Depending on business criticality deployment teams may need to ensure that the application continues to function, while the deployment is in progress.
- Due to the high cut-over time, the Production deployment usually takes place during non-peak hours and / or weekends.

Maintenance



This stage is when the “fine tuning” of the software takes place. Once the build is deployed to Production environment, any issues that the real users face are considered as Post-Production issues.

- These Post-Production issues are addressed and resolved by the internal team usually termed as Maintenance team.
- This stage also addresses minor change requests, code fixes, etc. and deploys them in short intervals.
- Build a helpdesk to support the system user.
- One may change the application without impairing existing functionalities.
- You may add new functionalities to the existing application.
- You can fix any historical defects of the application in this phase

The SDLC methodology is sometimes referred to as the waterfall methodology to represent how each step is a separate part of the process; only when one step is completed can another step begin. After each step, an organization must decide whether to move to the next step or not. This methodology has been criticized for being quite rigid. For example, changes to the requirements are not allowed once the process has begun. No software is available until after the programming phase.

Again, SDLC was developed for large, structured projects. Projects using SDLC can sometimes take months or years to complete. Because of its inflexibility and the availability of new programming techniques and tools, many other software-development methodologies have been developed. Many of these retain some of the underlying concepts of SDLC but are not as rigid.

SDLC MODEL

Waterfall Methodology

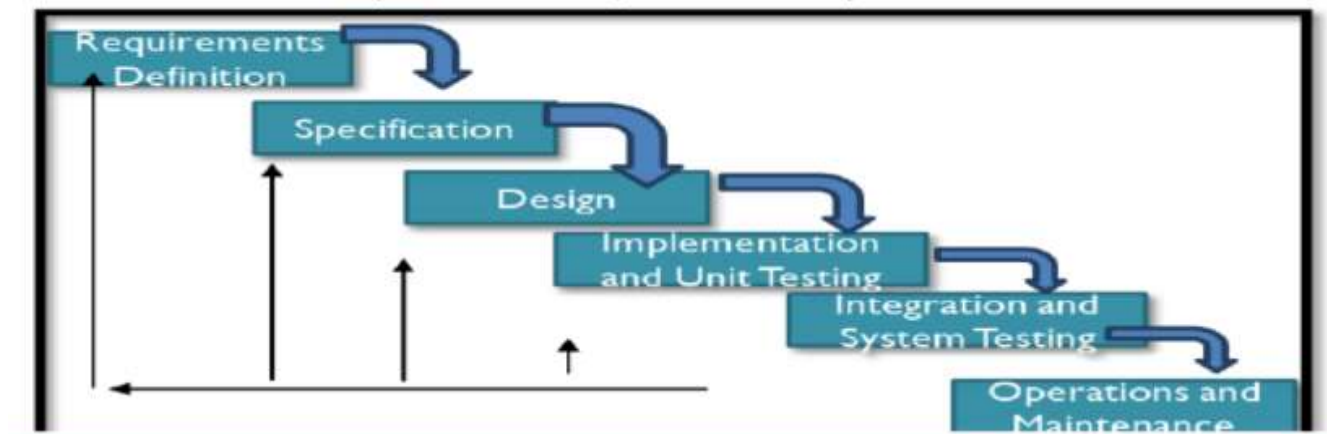
The Waterfall methodology is one of the oldest surviving SDLC methodologies. It follows a straightforward approach: the project development team completes one phase at a time, and each phase uses information from the last one to move forward.

While this methodology does make the needs and outcomes clear, and gives each stage of the model a well-defined starting and ending point, there are downsides in Waterfall’s rigidity. In fact, some experts believe the Waterfall model was never meant to be a working SDLC methodology for developing

software because of how fixed it is in nature. Because of this, SDLC Waterfall methods are best used for extremely predictable projects.

The waterfall is a cascade SDLC model that presents the development process like the flow, moving step by step through the phases of analysis, projecting, realization, testing, implementation, and support. This SDLC model includes gradual execution of every stage. Waterfall implies strict documentation. The features expected of each phase of this SDLC model are predefined in advance. The waterfall life cycle model is considered one of the best-established ways to handle complex projects.

Also called Linear Sequential Model/Classic Life Cycle Model.



Advantages:

- ✓ One of the main advantages of this model is its simplicity.
- ✓ It is conceptually straightforward and divides the large task of building a software system into a series of cleanly divided phases, each phase dealing with a separate logical concern.
- ✓ It is also easy to administer in a contractual setup as each phase is completed and its work product produced, some amount of money is given by the customer to the developing organization.

Limitations:

- ✓ It assumes that the requirements of a system can be frozen before the design begins. But for new systems, determining the requirements is difficult as the user does not even know the requirements.
- ✓ Freezing the requirements usually requires choosing the hardware, may become obsolete over a period of time.
- ✓ The entire software is delivered in one shot at the end. This entails heavy risks, as the user does not know until the very end what they are getting.

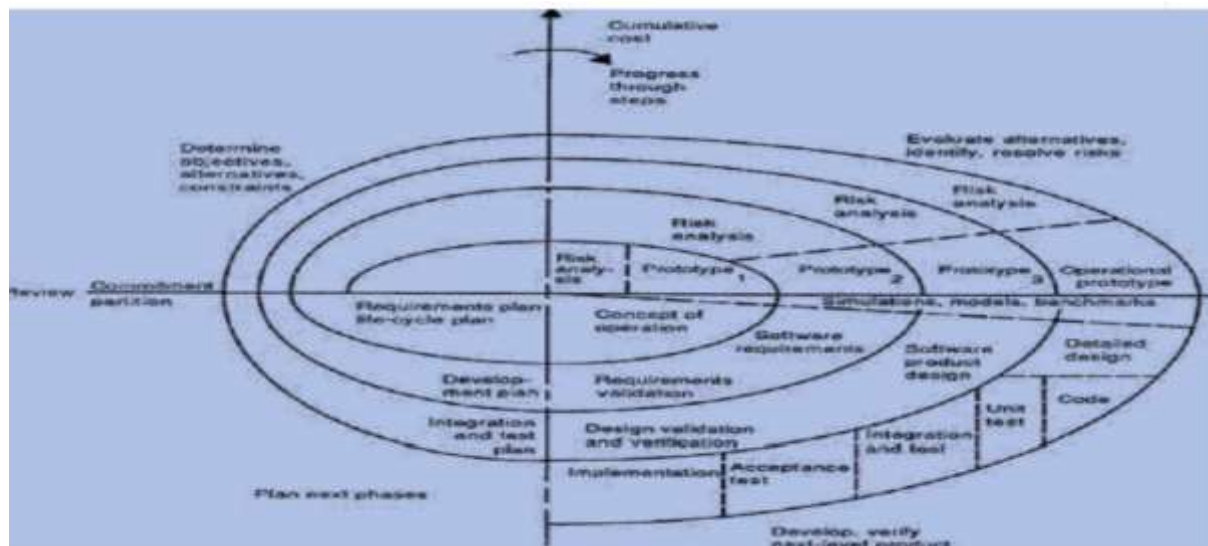
Agile

Agile is a combination of an incremental and iterative approach, where the product is released on an ongoing cycle then tested and improved at each iteration. Fast failure is encouraged in agile

methodology: the theory is that if you fail fast and early, you can solve minor issues before they grow into major issues. Agile is one of the most common methodologies out there today but it's technically more of a framework than a distinct model. Within Agile, there are sub-models in place such as extreme programming (XP), Rapid Application Development (RAD), Kanban and Scrum methodology.

Spiral SDLC Model

Spiral model is a combination of the Iterative and Waterfall SDLC models with a significant accent on the risk analysis. The main issue of the spiral model is defining the right moment to take a step into the next stage. The preliminary set timeframes are recommended as the solution to this issue. The shift to the next stage is done according to the plan, even if the work on the previous step isn't done yet. The plan is introduced based on the statistical data received in the last projects and even from the personal developer's experience



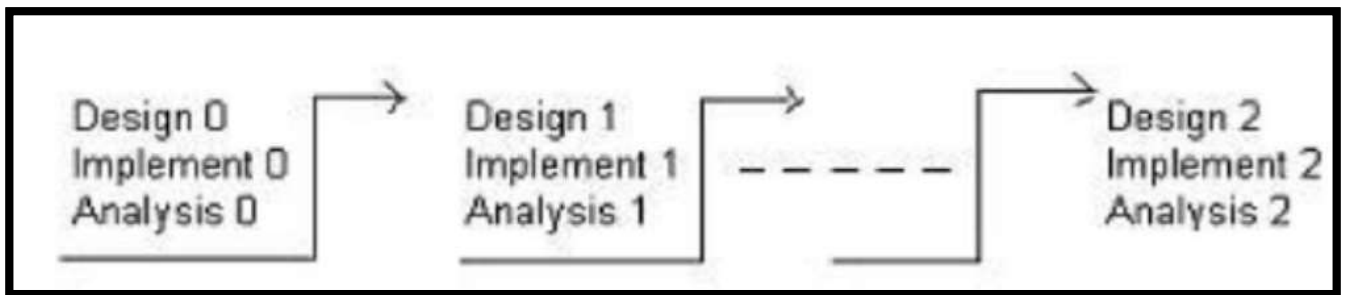
RAD

Rapid application development (RAD) is a software-development (or systems-development) methodology that focuses on quickly building a working model of the software, getting feedback from users, and then using that feedback to update the working model. After several iterations of development, a final version is developed and implemented.

Iterative Model

The iterative development process model tries to combine the benefits of both prototyping and the waterfall model. The basic idea is that the software should be developed in increments, each increment adding some functional capability to the system until the full system is implemented. At each step, extensions and design modifications can be made. An advantage of this approach is that it can result in better testing because testing each increment is likely to be easier than testing the entire system as in the

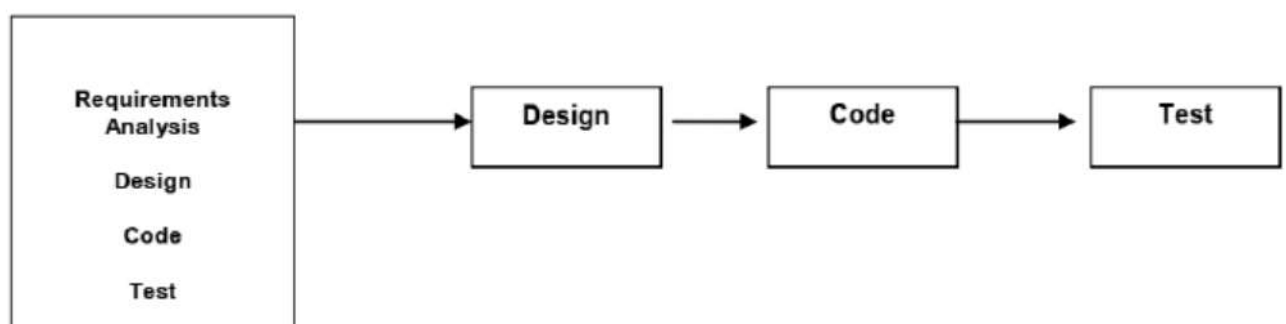
waterfall model. The prototyping, the increments provide feedback to the client that is useful for determining the final requirements of the system.



Prototyping Model

The goal of a prototyping-based development process is that instead of freezing the requirements before any design or coding can proceed. This prototype is developed based on the currently known requirements.

Development of the prototype obviously undergoes design, coding, and testing, but each of these phases is not done very formally or thoroughly. By using this prototype, the client can get an actual feel of the system; because the interactions with the prototype can enable the client to, better understand the requirements of the desired system. Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determine the requirements. In both situations, the risks associated with the projects are being reduced through the use of prototyping. After the prototype has been developed, the end users and clients are given an opportunity to use the prototype. Based on their experience, they provide feedback to the developers. Based on the feedback, the prototype is modified to incorporate some of the suggested changes that can be done easily, and then the users and the clients are again allowed to use the system. Based on the feedback, the initial requirements are modified to produce the final requirements specification, which is then used to develop the production quality system.



Problem Identification

One of the crucial tasks of in software development is continuously to scan operations for potential productivity improvements, anticipate problems, or, in the case of a poorly organized enterprise, sort out the most urgent problems or those to be solved with the least effort. A large number of checklists, questionnaires, audit procedures and guidelines are in use for screening possible problem areas. A detailed questionnaire on work organization and workplace layout was included in the third edition of the ILO publication Introduction to work study (ILO, 1979). Specific procedures have been elaborated for different sectors of industry, individual enterprises and functions such as product development, production and sales. Other weakness analyses concentrate on possible problem areas such as organization, training, technology, working conditions, etc.

Weakness analysis includes the following steps:

- define the level (enterprise, department, service) at which a weakness analysis should be applied;
- adapt existing procedures to your needs;
- train or advise collaborators in the use of weakness analysis;
- be sure to ask the right people to answer your questions while carrying out the analysis; and-evaluate the results.

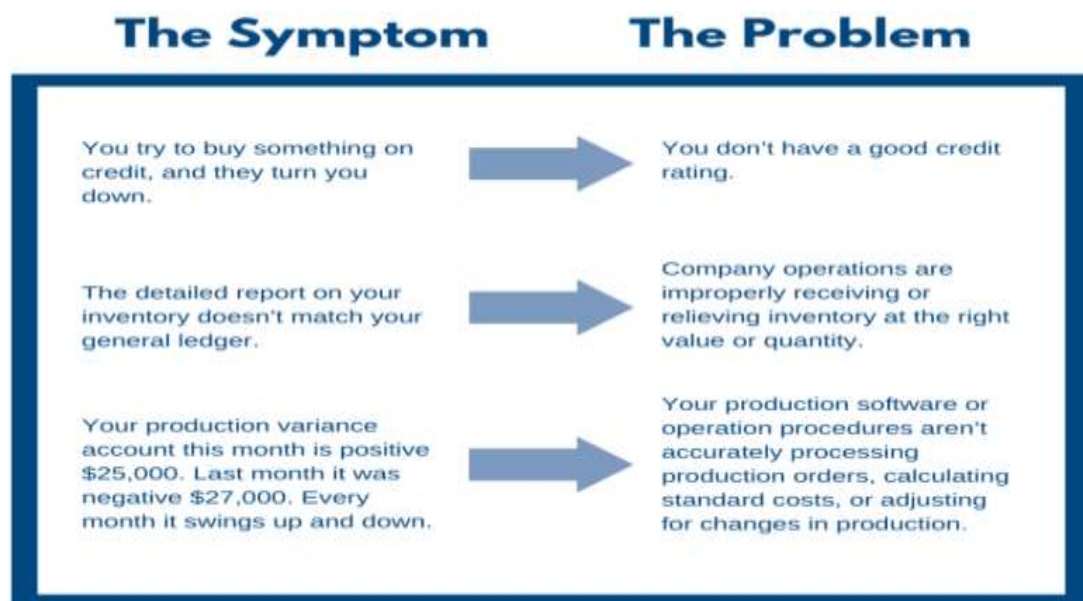
To identify the problem, use the following analysis type

Technique	Application	Main features
Weakness analysis	To screen possible problem areas	Different checklists/questionnaires available, brainstorming exercises, nominal grouping
SWOT analysis	To define Strengths, Weaknesses, Opportunities and Threats of an enterprise; to place productivity within strategic plans of an enterprise	SWOT factors are jointly (brainstorming) determined and weighted and alternative business (productivity) strategies are derived
Portfolio analysis	To concentrate productivity improvement on the most “promising” products	Groups products according to their life horizon and performance to achieve a balanced product structure
Pareto analysis	To define priorities of action based on one parameter	Groups objects (e.g. products) according to one parameter (e.g. sales volume); determines the 20% of products which account for 80% of sales, i.e. A-products)

Sensitivity analysis	To determine which action yields the best result (e.g. is energy saving better for overall profitability than cutting labour costs?)	In a set of parameters one is changed at a time, the others are kept constant and the influence on the result is calculated
Force-field analysis	To identify impelling and impeding forces in an organization	Impelling and impeding forces are determined, usually in a brainstorming exercise, and evaluated in their relative strength. The removal of restraining forces and strengthening of driving forces is analysed

Problem definition: symptoms vs problems

A problem's symptom is an indication that something didn't work out quite as expected. A problem cause, on the other hand, is the reason why the problem occurred in the first place.



Prioritizing problems

Problem Prioritization **helps us understand “what are the first set of problems that need solving?”**.

We do this by factoring in the intersection of all the things that are important to the business and the users. This exercise gives us the ability to focus our solution generation on a specific problem.

The Best Prioritization Techniques

1. RICE

it's a scoring system that sets your priorities and helps you consider each element of a project that you're working on. RICE is an acronym that stands for Reach, Impact, Confidence, and Effort. To assess the

priority level, you have to take into account the following factors: For each task, you and your team will agree on a score for each criterion.

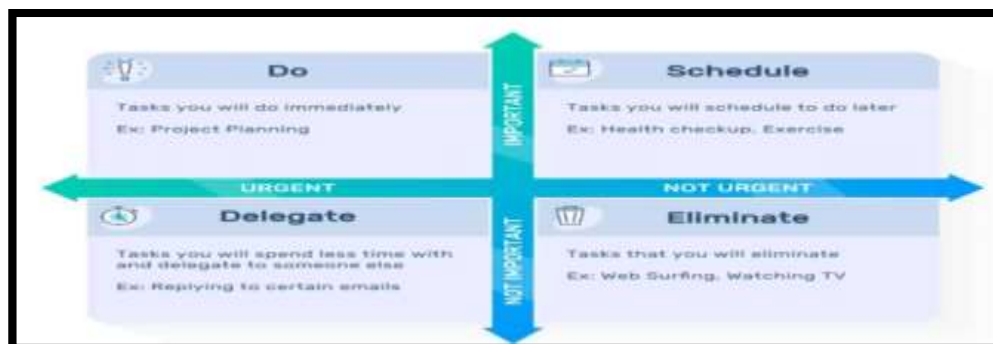
- **Reach** – An estimate how many people or events will be affected by the activity.
- **Impact** – Quantify the contribution of the activity to the end goal.
 - 3 for massive impact
 - 2 for high
 - 1 for medium
 - 0.5 for low
 - 0.25 for minimal
- **Confidence** – Estimate how confident you are about the impact
 - 100 for high confidence
 - 80 for medium
 - 50 for low
- **Effort** – An estimate of time and workforce (N people) needed to complete the task. The higher the effort, the higher the number we assign to it. For example:
 - Months: $5 \times N$ of people involved
 - Few weeks: $4 \times N$ of people
 - One week: $3 \times N$ of people
 - Few days: $2 \times N$ of people
 - One workday or less: $1 \times N$ of people

After assigning the scores to each of the categories, apply this formula:

$$RICE = (Reach * Impact * Confidence) Effort$$

The benefits of this model stand on being more informed in making decisions as a project manager and what's more advantageous to do first to have the most impact.

On the other hand, this method is very time-consuming, and there isn't always that much data for every product.



Eisenhower Matrix: 4 Quadrants time management

This method helps to be more productive and to focus on the important tasks ahead. It separates priorities into four quadrants to achieve a long-term success project. There's a chart divided into urgency and importance, which is identified as the following:

- **Urgent and important:** tasks such as deadlines and meetings;
- **Urgent but not important:** tasks such as long-term projects;
- **Not urgent but important:** tasks such as e-mails or calls;
- **Not urgent and not important:** tasks that are not directly related to your projects.

3. ABCDE method

Through this method, you can control and prioritize your tasks in your daily life. In order to use this technique, you must list all the personal and professional tasks you must follow up on. With that, you must assign each one to the following categories:

- **A stands for “Very important tasks”:** it's imperative that you take care of these, or you'll suffer consequences;
- **B stands for “Less important tasks”:** you need to take care of these tasks but not in an urgent manner;
- **C stands for “Nice tasks to do”:** these are tasks that should give you pleasure to do;
- **D stands for “Tasks to delegate”:** these tasks can be passed to someone else, so you don't get over whelmed;
- **E stands for “Tasks you can eliminate”:** these tasks should make part of your list, and you can eliminate them. etc. there are others

planning tools and techniques: there are six planning tools and techniques

Planning is a complex action that consists of a sequence of preconceived steps. They include the six major planning tools and techniques that managers in any sphere use, which are forecasting, contingency planning, scenario planning, benchmark analysis, participatory planning, and goal setting.

Forecasting helps predict what might happen in the future. Contingency planning and scenario planning are similar, the first one being short-term while the other is a long-term prognosis of things that might go wrong and preparing a particular plan of action to deal with them. Benchmarking is a planning technique that involves internal and external comparison to assess current performance and develop a plan of action to improve in the future. Participatory planning is supposed to involve all stakeholders in the process of planning. The last tool is goal setting, which is an essential element of any plan.

One of the most important though often neglected parts of planning is participatory planning. If people who are involved in the project do not understand what is going on, they can significantly impede the implementation down to its ruining. I could use this technique to organize a day-care/educational group for

children in which I am currently engaged. The team includes two organizers, a person responsible for the curriculum, and several other members in charge of miscellaneous issues. Not all of the team members seem to be aware of the strategic design. Growing uncertainty makes them show an initiative, that is out of place, and prevents them from following the intended plan. The failure to communicate openly causes misunderstanding, confusion, and interpersonal tensions. To explain to everyone how their input is going to be applied, and who in the team is involved in which part of the plan, is of utmost importance. The planning and organization process will benefit from using the participatory planning, should it be implemented.

Project management tool

Gantt chart

A Gantt chart, commonly used in project management, is one of the most popular and useful ways of showing activities (tasks or events) displayed against time. On the left of the chart is a list of the activities and along the top is a suitable time scale. Each activity is represented by a bar; the position and length of the bar reflects the start date, duration and end date of the activity. This allows you to see at a glance:

Gantt chart used for

Build and manage a comprehensive project

Gantt charts visualize the building blocks of a project and organize it into smaller, more manageable tasks. The resulting small tasks are scheduled on the Gantt chart's timeline, along with dependencies between tasks, assignees, and milestones.

Determine logistics and task dependencies

Gantt charts can be employed to keep an eye on the logistics of a project. Task dependencies ensure that a new task can only start once another task is completed. If a task is delayed (it happens to the best of us), then dependent issues are automatically rescheduled. This can be especially useful when planning in a multi-team environment.

Monitor progress of a project

As teams log time towards issues in your plan, you can monitor the health of your projects and make adjustments as necessary. Your Gantt chart can include release dates, milestones, or other important metrics to track your project's progress.

The benefits of using a Gantt chart

There are two main reasons Gantt charts are loved throughout the project management world. They make it easier to create complicated plans, especially those that involve multiple teams and changing deadlines. Gantt charts help teams to plan work around deadlines and properly allocate resources.

Project planners also use Gantt charts to maintain a bird's eye view of projects. They depict, among other things, the relationship between the start and end dates of tasks, milestones, and dependent tasks.

Modern Gantt chart programs such as Jira Software with Roadmaps and Advanced Roadmaps synthesize information and illustrate how choices impact deadlines.

How to use Gantt charts?

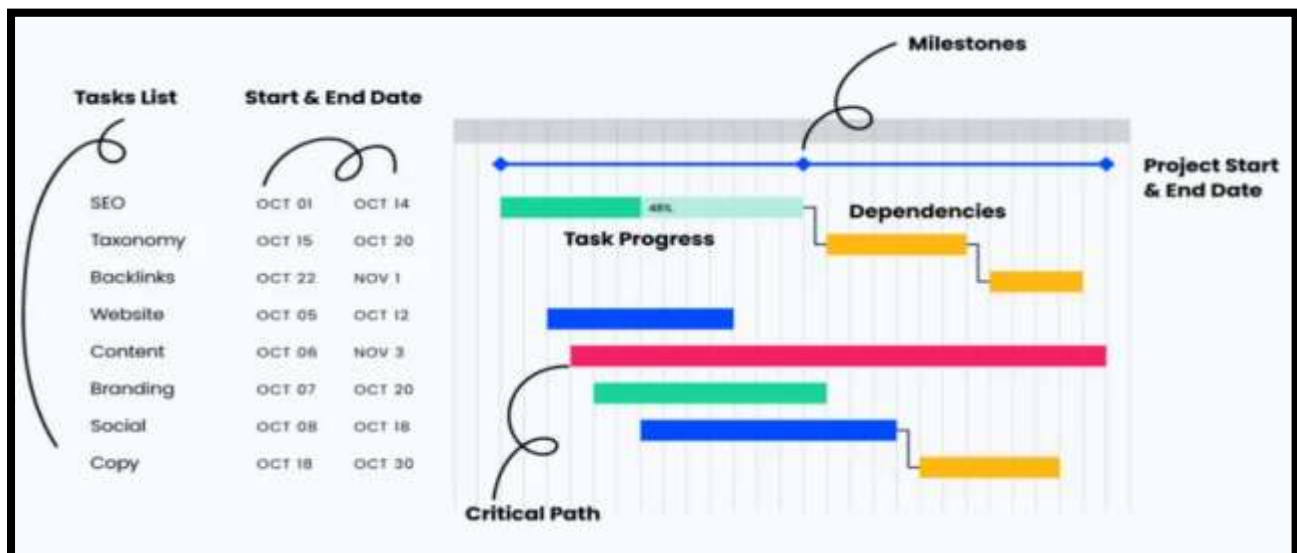
Jira Software comes with two different road mapping features, each with a slightly different focus. Jira Roadmaps is designed to track work assigned to a single team while Advanced Roadmaps is designed for larger, cross-organization project planning.

Elements of basic Gantt chart

By default, a Gantt chart is plain and rudimentary. That doesn't mean, however, that this type of project management tool lacks substance.

Gantt charts are typically configured with task names listed along the y axis, and the project timeline represented on the x axis. Even a simple Gantt chart should include the following 7 critical elements:

1. The **list of tasks** needed for this project: what are the activities required for project completion?
2. The **start date and end date of each task**
3. The **progress made toward the completion of each task**: is the task on track, at risk, or delayed?
4. The **dependencies across tasks**: how do the tasks relate to one another?
5. The **start date and end date for the entire project**
6. Important **milestone dates** within the project's timeline
7. The **project's critical path**: i.e., the set of tasks that take the longest time to complete in a project and so provide an estimate of project duration



Program Evaluation Review Technique (PERT)

What Is a Program Evaluation Review Technique (PERT) Chart?

A program evaluation review technique (PERT) chart is a graphical representation of a project's timeline that displays all of the individual tasks necessary to complete the project.

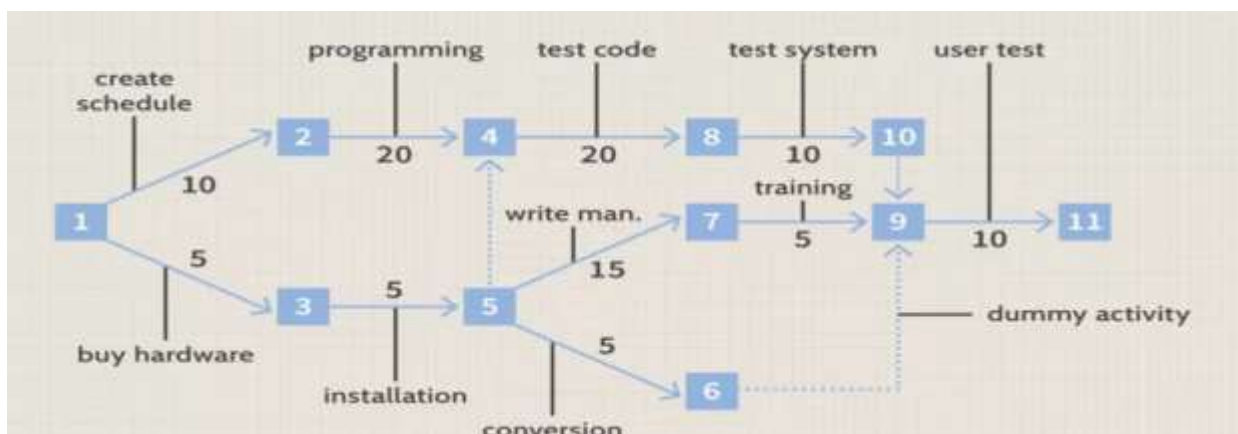
As a project management tool, the PERT chart is often preferred to the Gantt chart because it identifies task dependencies. However, a PERT chart can be more difficult to interpret. A PERT chart uses circles or rectangles called nodes to represent project events or milestones. These nodes are linked by vectors, or lines, that represent various tasks and their dependencies. A PERT chart allows managers to evaluate the time and resources necessary to manage a project.

a PERT Chart Work A project manager creates a PERT chart in order to analyze all of a project's tasks while estimating the amount of time required to complete each one. Using this information, the project manager can estimate the minimum amount of time required to complete the entire project.

This information also helps the manager develop a project budget and determine the resources needed to accomplish the project. A PERT chart uses circles or rectangles, called nodes, to represent project events or milestones. The nodes are linked by vectors or lines that represent various tasks.

Dependent tasks are items that must be performed in a specific manner. For example, if an arrow is drawn from Task No. 1 to Task No. 2 on a PERT chart, Task No. 1 must be completed before work on Task No. 2 begins. Items at the same stage of production but on different task lines within a project are referred to as parallel tasks. They're independent of each other, and occur at the same time.

A well-constructed PERT chart looks like this:



- Numbered rectangles are nodes and represent events or milestones.
- Directional arrows represent dependent tasks that must be completed sequentially.
- Diverging arrow directions (e.g. 1-2 & 1-3) indicate possibly concurrent tasks.
- Dotted lines indicate dependent tasks that do not require resources.

Chapter Two

Object Orientation the new software paradigm

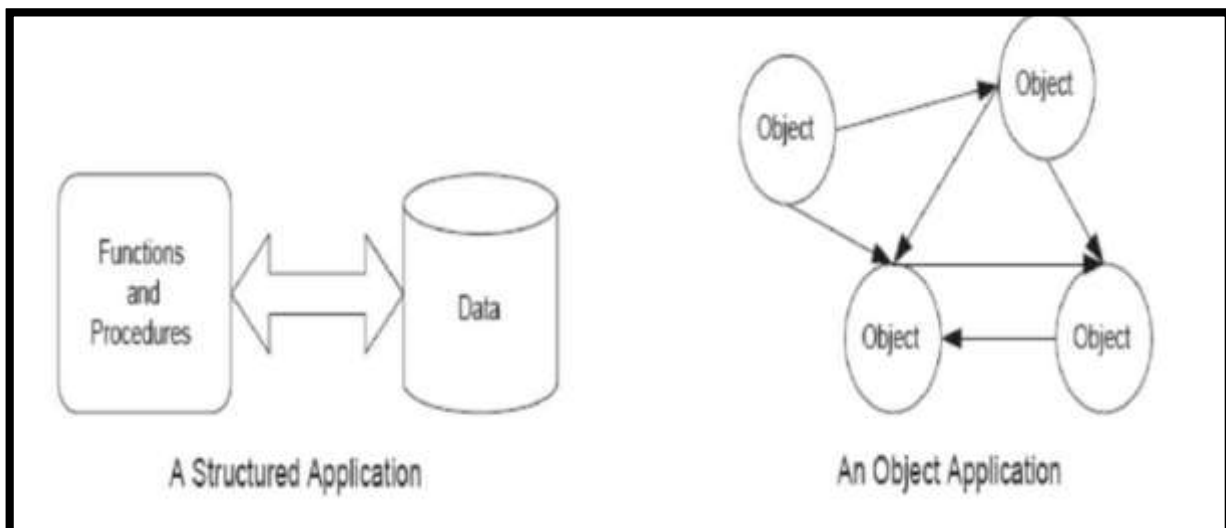
Information systems are an outgrowth of a process of organizational problem solving. Information systems developed due to; Problems, real or anticipated undesirable situations that hinder the organization from achieving their goals and that require corrective actions. Opportunities a chance to improve an organization even in the absence of an identified problem.

Structured Paradigm

The structured paradigm is a development strategy based on the concept that a system should be separated into two parts: Data and functions/procedures (modeled using a process model). Using the structured approach, you develop applications in which data is separated from behavior in both the design model and in the system implementation (that is, the program). Example: Consider the design of an information system for a university. Taking the structured approach, you would define the layout of a data initially as a separate system and the design of a program to access that data as another. The programs have the ability to change the data states.

Object oriented paradigm

The main concept behind the object oriented paradigm is that instead of defining systems as two separate parts (data and functions), system is defined as a collection of interacting objects. It describes and builds a system that consists of objects. An object oriented system comprises a number of software objects that interact to achieve the system objective.



The potential benefits of object orientation

Increased Reusability

The object oriented paradigm provides opportunities for reuse through the concepts of inheritance, polymorphism, encapsulation, coupling and cohesion. OO paradigm provides more opportunities for reuse than the structured paradigm.

Increased Extensibility

Because classes have both data and functionality, when you add new features to the system you need to make changes in one place – the class.

Improved Quality

Quality systems are on time, on budget and meet or exceed the expectations of their users. Improved quality comes from increased participation of users in systems development. OO systems development techniques provide greater opportunity for users to participate in the development process.

Financial Benefits

Reusability, extensibility, and improved quality are all technical benefits. Object orientation enables you to build systems better, faster and cheaper. The benefits OO are realized throughout the entire development life cycle, not just programming.

Increased chance of project success

A project is successful if it is on time, on budget and meets the needs of the users. Users are expert at business and they are the only ones who can tell you what they need. You need to know the right question to ask, know the business very well. You need models that communicate the required information and that users understand. You need to work closely with users; Time invested in defining requirements and modeling pays off in the long run.

Reduce maintenance Burdon

Software organizations currently spend significant resources (80%) maintaining and operating software, and because of the long waiting list of work to be done, it takes significant time to get new projects started. These two problems are respectively called the maintenance Burdon and the application backlog. These are problems that object orientation can help you to overcome.

The potential drawbacks of object orientation

Nothing is perfect including object orientation. While many benefits exist to OO, they come at a price:

- ✓ OO requires greater concentration on requirements analysis and design
 - You cannot build a system that meets users' needs unless you know what those needs are (you need to do requirements)
 - You cannot build a system unless you know how it all fit together (you need to do analysis and design)
 - But this fact is often ignored by many developers

- ✓ Developers must closely work with users
 - Users are the experts but they have their own jobs to do (busy)
- ✓ OO requires a complete change in the mindset on the part of individuals
 - They should understand the benefits of OO
- ✓ OO is just more than programming
- ✓ Many OO benefits are long term
 - OO truly pays off when you extend and enhance your system
- ✓ OO demands up front investments in training education and tools
 - Organizations must train and educate their development staff.
 - Development tools and magazines
- ✓ OO techniques do not guarantee you will build the right system
 - While OO increases the probability of project success, it still depends on the ability of individuals involved.
 - Developers, users, managers must be working together to have a good working atmosphere
- ✓ OO necessitates increased testing
 - OO is typically iterative in nature, and probably developing complex system using the objects, the end result is you need to spend more time in testing.
- ✓ OO is only part of the solution
 - You still need CASE tools
 - Need to perform quality assurance (QA)

Object standards

Object orientation today becomes the significant part of the software development. Objects are the primary enabling technology for components. It defines requirements for the information, geometry, behavior, and presentation of objects, to give reassurance of quality that will enable greater collaboration and efficient information exchange across construction industry for the system. By standardizing objects, you can consistently use, compare, analyze, and share information to make informed decisions quickly and confidently. The standard is paramount, not just for some objects as we can create objects to a common data set. Since the lack of structure and data consistency recognized across all objects, object standard is required to achieve data consistency and better structure with recognizable set of criteria, to which all objects can be created.

Basic standards of object for analysis include:

- ✓ Identifying objects
- ✓ Identifying the object relationships

- ✓ Identifying the attributes
- ✓ Identifying services
- ✓ Defining object life cycles

The object orientation software process and models

Software process is a set of project phases, stages, methods, techniques, and practices that people employ to develop and maintain software and its associated artifacts. It enables organizations to increase productivity when developing software. Any modern object-oriented approach to develop software process must have objectives, a focus of activity over the workflows, and incremental deliverables. Each of the phases is described as follows:

Inception

In this phase, a business case is made for the proposed system. This includes feasibility analysis that should answer questions such as the following:

- ✓ Do we have the technical capability to build it?
- ✓ If we build it, will it provide business value?
- ✓ If we build it, will it be used by the organization
- ✓ To answer these questions, the development team performs work related primarily to the business modeling, requirements, and analysis workflows. This implies that the design, implementation, and test workflows also could be involved. The primary deliverables from the inception phase are:
 - ✓ A vision document that sets the scope of the project, identifies the primary requirements and constraints, sets up an initial project plan, and describes the feasibility and risks associated with the project
 - ✓ The adoption of the necessary environment to develop the system

Elaboration

The analysis and design workflows are the primary focus during this phase. This phase continues with developing the vision document, including finalizing the business case, revising the risk assessment, and completing a project plan in sufficient detail to allow the stakeholders to be able to agree with constructing the actual final system. It deals with gathering the requirements, building the UML structural and behavioral models of the problem domain, and detailing the how the problem domain models fit into the evolving system architecture. The primary deliverables of this phase include:

- ✓ The UML structure and behavior diagrams
- ✓ An executable of a baseline version of the evolving information system

Construction

The construction phase, as expected by its name, is heavily focused on programming the evolving information system. As such, it is primarily concerned with the implementation workflow. However, the requirements, analysis, and design workflows also are involved with this phase. It is during this phase that missing requirements are uncovered, and the analysis and design models are finally completed. At times, an iteration may have to be rolled back. The primary deliverable of this phase is an implementation of the system that can be released for beta and acceptance testing.

Transition

Like the construction phase, the transition phase addresses aspects typically associated with the implementation phase of SDLC approach. Its primary focus is on the testing and deployment workflows. Depending on the results from the testing workflow, it is possible that some redesign and programming activities on the design and implementation workflows could be necessary, but they should be minimal at this point in time. From a managerial perspective, the project management, configuration and change management, and environment are involved. The other deliverables include user manuals, a plan to support the users, and a plan for upgrading the information system in the future.

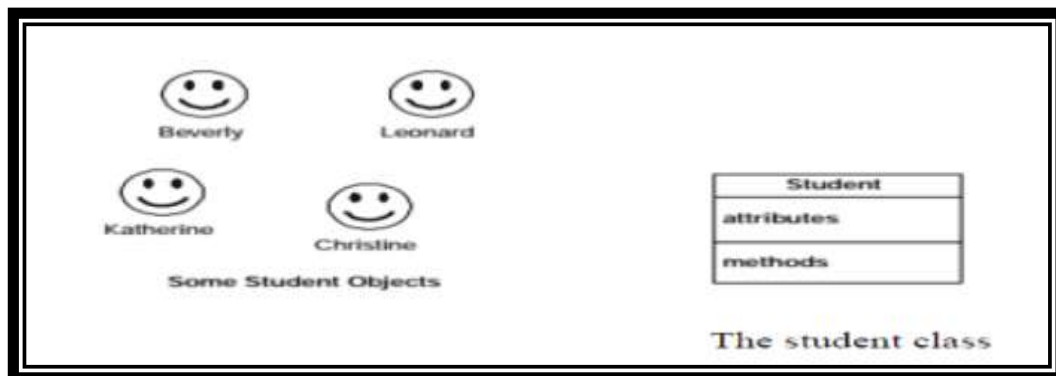
Chapter Three

Understanding the Basics Object oriented concepts

OO concepts from structured point of view

Objects and Classes

The OO paradigm is based on building systems from items called objects. An object is any person, place, thing, event, concept, screen, or report. A class generalizes/represents a collection of similar objects and is effectively a template from which to create objects. In a university system, Sarah is a student object, she attends several seminar objects, and she is working on a degree object. In a banking system, Sarah is a customer object. She has a checking account object from which she bounces rubber-check objects. In an inventory control system, every inventory item is an object, every delivery is an object, and every customer is an object. In the real world, you have objects; therefore, you need them as a concept to reflect your problem space accurately. However, in the real world, objects are often similar to other kinds of objects. Students share similar qualities (they do the same sort of things; they are described in the same sort of way), courses share similar qualities, inventory items share similar qualities, bank accounts share similar qualities, and so on. While you could model (and program) every object, that is a lot of work. I prefer to define what it is to be a student once, define course once, define inventory item once, and define bank account once, and so on. That is why you need the concept of a class.



The above figure depicts how we have student objects and how we model the class Student. It also shows the standard notations to model a class. Classes are typically modeled by using a rectangle that lists its attributes and methods. Listing the attributes and methods can be quite helpful. It enables readers of your class models to gain a better understanding of your design at a single glance. Class names are typically singular nouns. The name of a class should be one or two words, usually a noun, and should accurately describe the class using common business terminology. If you are having trouble naming a class, either you need to understand it better or it might be several classes you have mistakenly combined. You should model classes with names like Student, Professor, and Course, not Students, People Who Teach Seminars. Think of it like this: in the real world, you would say —I am a

student, I am not a student's. Class can also represent concepts that are not nouns, like the process of checking out a book from a library. When object-oriented software is running, objects are instantiated (created/defined) from classes. We say an object is an instance of a class and we instantiate those objects from classes.

Attributes and Methods

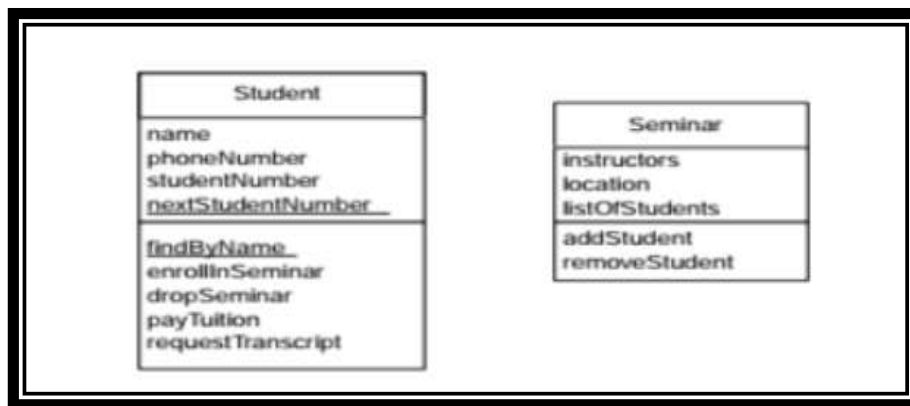
Classes have responsibilities, the things they know and do. Attributes are the things classes know; methods are the things classes do. The object-oriented paradigm is based on the concepts that systems should be built out of objects, and that objects have both data and functionality. Attributes define the data, while methods define the functionality. When you define a class, you must define the attributes it has, as well as its methods. The definition of an attribute is straightforward. You define its name, perhaps its type (whether it is a number, a string, or a date, and so forth). The definition of a method is simpler: you define the logic for it, just as you would code for a function or a procedure. An important implication is that methods do one of two things: either they return a value and/or they do something of value; that is, they have a side effect.

In the following figure, we are using the three-section class notation in this case: the top section for the name, the middle section to list the attributes, and the bottom section to list the methods. It depicts two types of attributes: instance attributes, which are applicable to a single object, and static attributes, which are applicable to all instances of a single class.

Attributes are underlined, instance attributes are not. For example, name is an instance attribute of the class Student. Each individual student has a name; for example, one student may have the name —Smith, John, I whereas another student may have the name —Jones, Sally. I It could even happen that two individual students may have the same name, such as —Smith, JohnI although they are, in fact, two different people. On the other hand, nextStudentNumber is a static attribute (also referred to as a class attribute) that is applicable to the class Student, not specifically to individual instances. This attribute is used to store the value of the next student number to be assigned to a student: when a new student joins the school, his or her student number is set to the current value of next StudentNumber, which is then incremented to ensure all students have unique student numbers.

Similarly, there is the concept of instance methods and static/class methods: instance methods operate on a single instance, whereas static methods operate potentially on all instances of a single class. In the following figure, you see that Student has instance methods called enrollInSeminar and dropSeminar, things an individual student would do. It also has the static method findByName, which supports the behavior of searching for students whose names meet specified search criteria, a method that operates on all instances of the class.

Example:



Abstraction, Encapsulation and information hiding

Instead of saying we determined what a class knows and does, we say we ‘abstracted’ the class. Instead of saying we designed how the class will accomplish these things; we say we ‘encapsulated’ them. Instead of saying we designed the class well by restricting access to its attributes, we say we have ‘hidden’ the information.

Abstraction

The world is a complicated place. To deal with that complexity we form abstractions of the things in it. For example, consider the abstraction of a person. From the point of view of a university, it needs to know the person ‘s name, address, telephone number, social security number, and educational background. From the point of view of the police, they need to know a person ‘s name, address, phone number, weight, height, hair color, eye color, and so on. It is still the same person, just a different abstraction, depending on the application at hand. Abstraction is an analysis issue that deals with what a class knows or does. Your abstraction should include the responsibilities, the attributes, and the methods of interest to your application—and ignore the rest. That is why the abstraction of a student would include the person ‘s name and address, but probably not his or her height and weight. OO systems abstract only what they need to solve the problem at hand. People often say abstraction is the act of painting a clear box around something: you are identifying what it does and does not do. Some people will also say that abstraction is the act of defining the interface to something. Either way, you are defining what the class knows and does.

Encapsulation

Although the act of abstraction tells us that we need to store a student’s name and address, as well as be able to enroll students in seminars, it does not tell us how we are going to do this. Encapsulation deals with the issue of how you intend to modularize the features of a system. In the object-oriented world, you modularize systems into classes, which, in turn, are modularized into methods and attributes. We say that we encapsulate behavior into a class or we encapsulate functionality into a method.

Encapsulation is a design issue that deals with how functionality is compartmentalized within a system. You should not have to know how something is implemented to be able to use it. The implication of encapsulation is that you can build anything anyway you want, and then you can later change the implementation and it will not affect other components within the system (as long as the interface to that component did not change). People often say encapsulation is the act of painting the **box black**: you are defining how something is going to be done, but you are not telling the rest of the world how you are going to do it. In other words you are hiding the details of the implementation of an item from the users of that item. For example, consider your bank. How does it keep track of your account information, on a mainframe, a mini, or a PC? What database does it use? What operating system? It does not matter, because it has encapsulated the way in which it performs account services. You just walk up to a teller and initiate whatever transactions you want. By hiding the details of the way it has implemented accounts, your bank is free to change that implementation at any time, and it should not affect the way services are provided to you.

Information Hiding

To make your applications maintainable, you want to restrict access to data attributes and some methods. The basic idea is this: if one class wants information about another class, it should have to ask for it, instead of taking it. When you think about it, this is exactly the way the real world works. If you want to learn somebody's name, what would you do? Would you ask the person for his name, or would you steal his wallet and look at his ID? By restricting access to attributes, you prevent other programmers from writing highly coupled code. When code is highly coupled, a change in one part of the code forces you to make a change in another, and then another, and so on. Coupling is described in detail in the next Section.

Inheritance

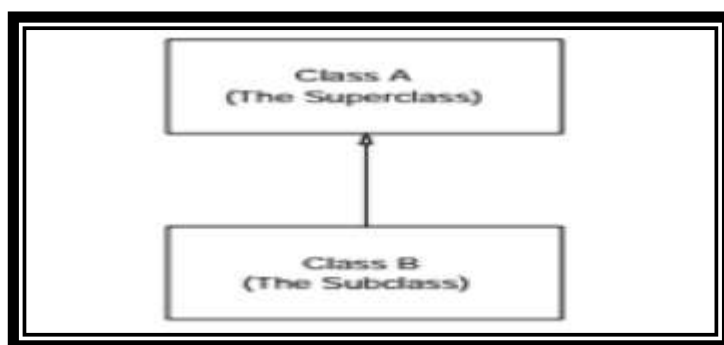
Similarities often exist between different classes. Two or more classes often share the same attributes and/or the same methods. Because you do not want to have to write the same code repeatedly, you want a mechanism that takes advantage of these similarities. Inheritance is that mechanism. Inheritance models —is a, —is kind of, and —is like relationships, enabling you to reuse existing data and code easily. For example, students have names, addresses, and telephone numbers, and they drive vehicles. At the same time, professors also have names, addresses, and telephone numbers, and they drive vehicles. Without a doubt, you could develop the classes for student and professor and get them both running. In fact, you could even develop the class Student first and, once it is running, make a copy of it, call it Professor, and make the necessary modifications. While this is straightforward to do, it is not perfect. What if there was an error in the original code for student? Now you must fix the error in two places, which is twice the work. What would happen if you needed to change the way you handled

names (say, you go from a length of 30 to a length of 40)? Now you would have to make the same change in two places again, which is a lot of dull, boring, tedious work (and costly). Would it not be nice if you had only one copy of the code to develop and maintain? This is exactly what inheritance is all about.

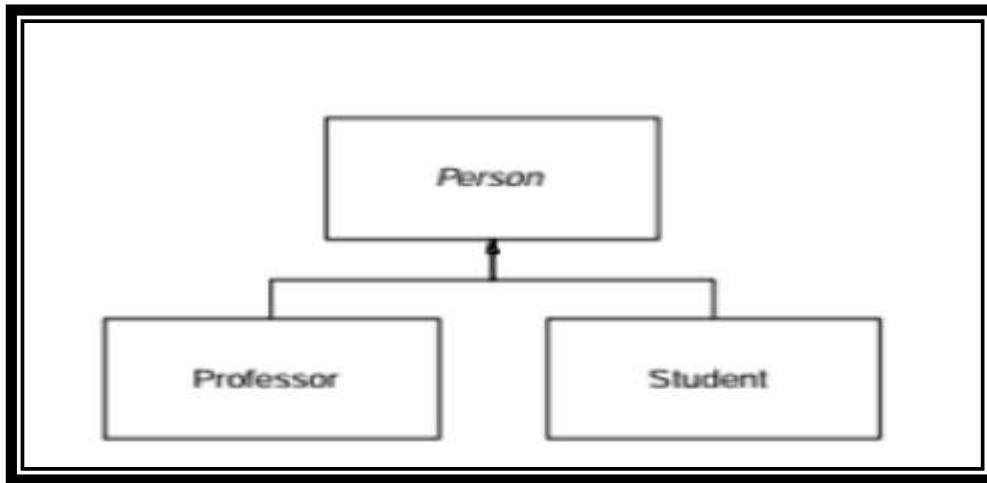
With inheritance, you define a new class that encapsulates the similarities between students and professors. This new class would have the attributes name, address, and phoneNumber, and the method driveVehicle. Because you need to name all our classes, you need to ask yourself what this collection of data and functionality describes. In this case, the name Person is fitting. Once you have the class Person defined, you then make Student and Professor inherit from it. You would say Person is the superclass of both Student and Professor, and Student and Professor are the subclasses of Person. Everything that a superclass knows or does, the subclass knows or does free without writing extra code. Actually, for this example, you would need to write two lines of code, one saying Student is a subclass of Person, and another saying Professor is a subclass of Person; therefore, it is almost free. Because Person has a name, address, and telephone number, both Student and Professor also have those attributes. Because Person has the ability to drive a vehicle, so do the classes Student and Professor.

Modeling Inheritance

The following figure depicts the modeling notation for inheritance, a line with a closed arrowhead. The way you would read the diagram is —B inherits from A. In other words, B is a direct subclass of A and A is the direct superclass of B.



The below figure presents how you would model the Person inheritance class hierarchy, often simply called a class hierarchy. Notice how the name of the Person class is in italics, indicating it is abstract, whereas Professor and Student are concrete classes. Abstract and concrete classes are discussed in the following Section.



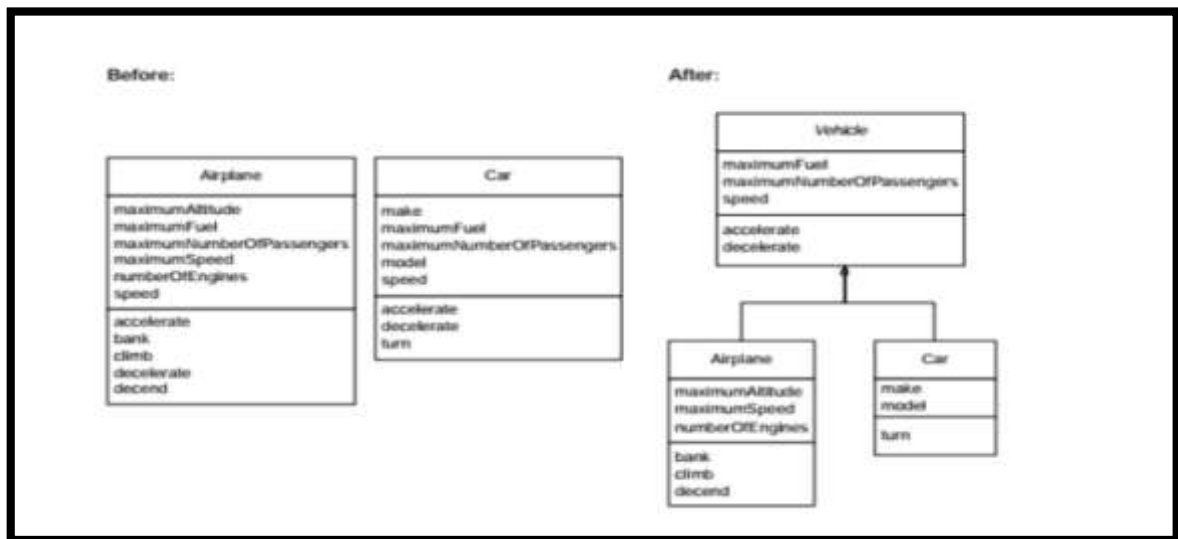
Single and Multiple Inheritance

When a class inherits from only one other class, we call this single inheritance. When a class inherits from two or more other classes, we call this multiple inheritance. Remember this: the subclass inherits all the attributes and methods of its superclass(es). Not all languages support multiple inheritance. C++ is one of the few languages that does, whereas languages such as Java, Smalltalk, and C# do not. The point to be made is if your target implementation language does not support multiple inheritance, then you should not use it when you are modeling.

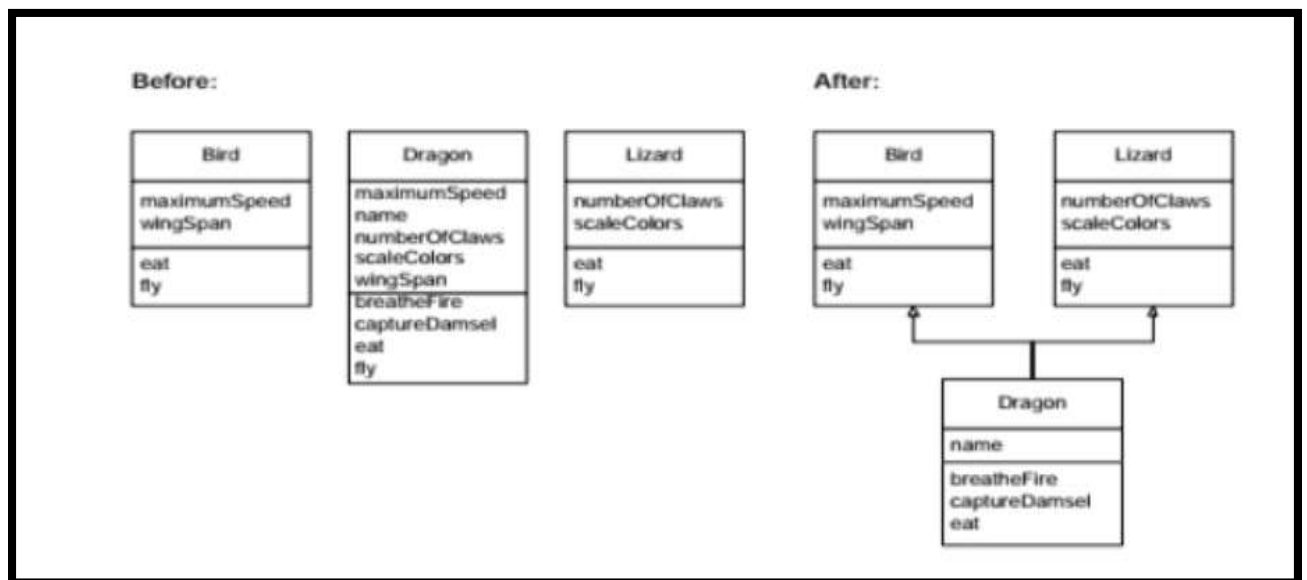
In the below figure, you see several similarities between airplanes and cars. They both have a number of passengers, a maximum fuel level, and they can either increase or decrease their speed. To take advantage of these similarities, you could create a new class called Vehicle and have Airplane and Car inherit from it. We say the classes Vehicle, Airplane, and Car form an inheritance hierarchy, also called a class hierarchy. The topmost class in a class hierarchy (in this case Vehicle) is called the root or root class. Notice how there is the method Turn for Car and Bank for Airplane. Turning and banking are exactly the same thing. You could have defined a Turn method in Vehicle, and had Airplane and Car inherit it (then you would remove Bank and Turn from the subclasses). This would imply that you would require users of airplanes (probably pilots) to change the terminology they use to work with airplanes. Realistically, this would not work. A better solution would be to define Turn in Vehicle and have the method Bank invokes it as needed.

In the following diagram, the class Vehicle is marked as abstract (the name is in italics; in previous versions of the UML you could also indicate with the constraint {abstract}), whereas Airplane and Car are not. We say Vehicle is an abstract class, whereas Airplane and Car are both concrete classes. The main difference between abstract classes and concrete classes is that objects are instantiated (created) from concrete classes, but not from abstract classes. For example, in your problem domain, you have airplanes and cars, but you do not have anything that is just a vehicle (if something is not an airplane or a car, you are not interested in it). This means your software will instantiate airplane and car objects, but

will never create vehicle objects. Abstract classes are modeled when you need to create a class that implements common features from two or more classes.



In the before picture of the following diagram, you might want to create a new class called Dragon. You already have the classes Bird and Lizard. A dragon is like a bird because they both fly. A dragon is also like a lizard because they both have claws and scales. Because dragons have the features of both birds and lizards, in the after picture we have the class Dragon inheriting from both Bird and Lizard. This is an example of an ‘is like’ relationship: a dragon is like a bird and a dragon is (also) like a lizard.

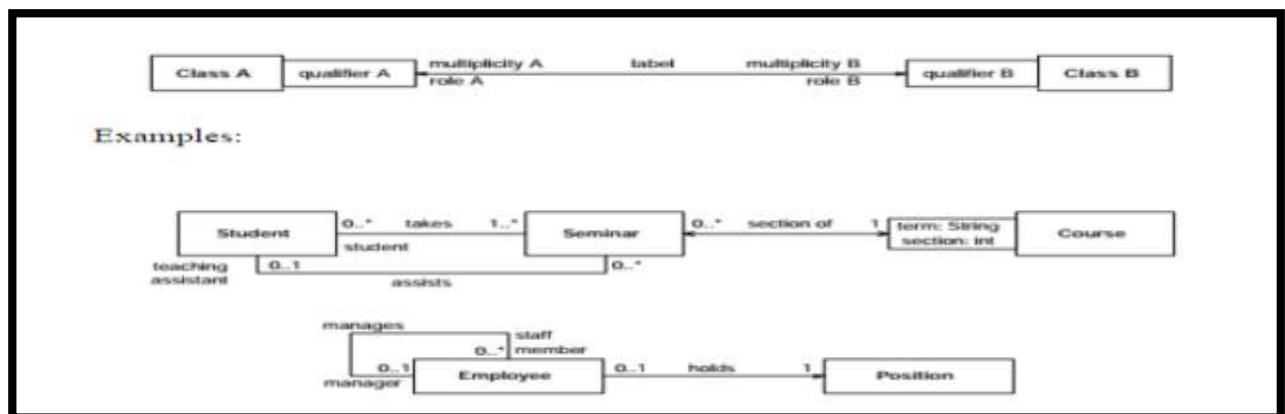


Notice how we listed the method `eat` for Dragon. Although all three types of creatures eat, they all eat in different ways. Birds eat bird seed, lizards eat bugs, and dragons eat knights in shining armor. Because the way dragons eat is different than the way either birds or lizards eat, I needed to redefine, or override, the definition of the `eat` method. The general idea is a subclass will need to override the definition of

either an attribute (occasionally) or a method whenever it uses that data, or performs that method in a manner different than that of its superclass.

Association

An association is a persistent relationship between two or more classes or objects. When you model associations in class diagrams, you show them as a thin line connecting two classes, as you see in the below figure.



Associations can become quite complex; consequently, you can depict several things about them on your diagrams. The above Figures show the common items to model for an association:

- ✓ **Directionality:** The open arrowheads indicate the directionality of the association. When there is one arrowhead the association is unidirectional: it can be traversed in one direction only (in the direction of the arrow). When there is either zero or two arrowheads the association is bi-directional: it can be traversed in both directions. Although you should indicate both arrowheads it is common practice to drop the arrowheads from bi-directional associations.
- ✓ **Label:** The label, which is optional, is typically one or two words describing the association. Reading one class, the label, and then the other class should produce a sentence fragment that describes the relationship, e.g., Professor teaches
- ✓ **Seminars.** Avoid generic labels like —has|| or —communicates with|| as much as possible.
- ⊗ **Multiplicity:** The multiplicity of the association is labeled on either end of the line, one multiplicity indicator for each direction (below table summarizes the potential multiplicity indicators you can

use).

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only n (where $n > 1$)
0..n	Zero to n (where $n > 1$)
1..n	One to n (where $n > 1$)

Table 28: Multiplicity indicators

Here is how you would read the associations:

- ✓ A student takes one or more seminars
- ✓ A seminar is taken by zero or more students

A student, as a teaching assistant, may assist in zero or more seminars

- ✓ A seminar may have zero or one student who acts as a teaching assistant
- ✓ A seminar is a section of one course
- ✓ A course has zero or more sections
- ✓ An employee holds one position
- ✓ A position may be held by one employee
- ✓ An employee may be managed by one other employee, their manager (the company president is the only one without a manager)
- ✓ An employee manages zero or more employees (some employees do not have any staff members)
- ✓ **Role:** The role is the context that an object takes within the association— may also be indicated at each end of the association.
- ✓ **Qualifier:** A qualifier is a value that selects an object from the set of related objects across an association. Qualifiers are optional and in practice are rarely modeled.

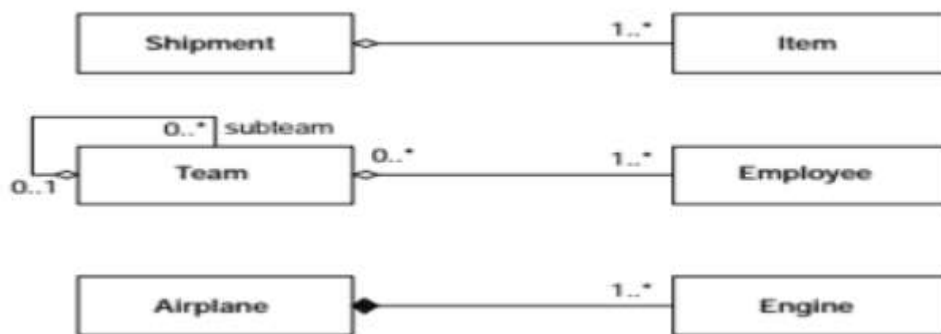
Several important lessons are contained in the earlier above figure. First, you see it is possible to have more than one association between two classes: the classes Student and Seminar have the takes and assists associations between them.

You are interested in two relationships between these two classes for your university information system; therefore, you need to model both associations. second, it is valid that the same class may be involved with both ends of an association; something called a recursive association or a self-association. A perfect example of this is the manages association that the Employee class has with itself. The way you read this association is that any given employee may have several other employees he or she manages, and that one other employee may, in turn, manage them.

Third, sometimes the direction of an association is one way, as you have seen in the figure, with the holds association between Employee and Position. The implication is employee objects know the position object they hold, but the position object does not need to know what employee holds it. This is called a unidirectional association, an association that is traversed in only one direction. If you do not have a requirement to traverse an association in both directions, for example, position objects do not have a need to collaborate with employee objects, and then you should use a unidirectional association.

Aggregation

Sometimes an object is made up of other objects. For example, an airplane is made up of a fuselage, wings, engines, landing gear, flaps, and so on. A delivery shipment contains one or more packages. A project team consists of two or more employees. These are all examples of the concept of aggregation, which represents —is part of relationships. An engine is part of a plane, a package is part of a shipment, and an employee is part of a team. Aggregation is simply a type of association; therefore, you still need to model the multiplicity and roles, just as you would with associations. Not indicating the multiplicity of the whole end of an aggregation is permissible as shown below:



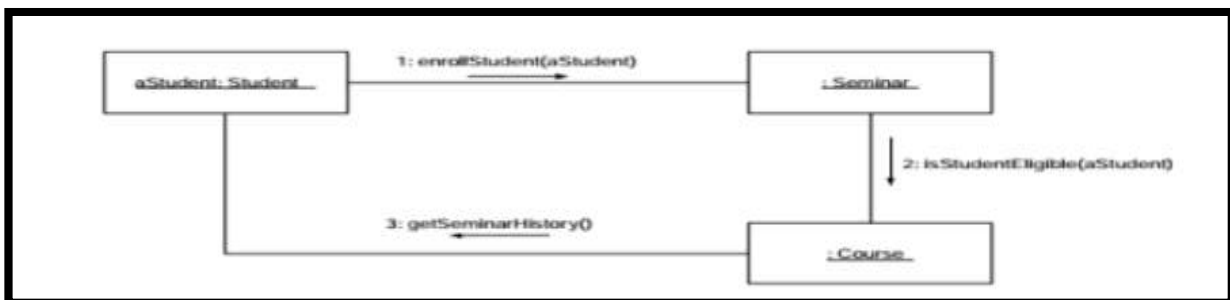
For example, in the figure above, multiplicity is not indicated for either shipments or airplanes; in these cases, it is assumed the multiplicity is 1. Just as associations are two-way streets, so is aggregation.

Furthermore, the aggregation is read in a similar manner:

- ✓ An item is part of one and only one shipment
- ✓ A shipment is composed of one or more item
- ✓ An engine is part of one and only one airplane
- ✓ An airplane has one or more engines
- ✓ An employee may be part of one or more teams
- ✓ A team is made up of one or more employees
- ✓ Any given team may be part of a larger team
- ✓ A team may be made up of smaller sub teams

Collaboration

Classes often need to work together to fulfill their responsibilities. Actually, it is typically the objects and the instances of the classes that are working together. Collaboration occurs between objects when one object asks another for information or to do something. For example, an airplane collaborates with its engines to fly. For the plane to go faster, the engines must go faster. When the plane needs to slow down, the engines must slow down. If the airplane did not collaborate with its engines, it would be unable to fly. Objects collaborate with one another by sending each other messages. A message is either a request to do something or a request for information. Messages are modeled in sequence and collaboration diagrams.



Persistence

Persistence focuses on the issue of how to make objects available for future use of your software. In other words, how to save objects to permanent storage. To make an object persistent, you must save the values of its attributes to permanent storage (such as a relational database or a file) as well as any information needed to maintain the relationships (aggregation, inheritance, and association) with which it is involved. In other words, you need to save the appropriate properties to permanent storage. In addition to saving objects, persistence is also concerned with their retrieval and deletion. From a development point of view, there are two types of objects: persistent objects that stick around and transient objects that do not. For example, a Customer is a persistent class. You want to save customer objects into some sort of permanent storage so you can work with them again in the future. A customer editing screen, however, is a transient object. Your application creates the customer-editing screen object, displays it, and then gets rid of it once the user is done editing the data for the customer with whom he or she is currently dealing.

Cohesion

Cohesion is a measure of how much an item, such as a class or method, makes sense. A good measure of the cohesiveness of something is how long it takes to describe it in one sentence: the longer it takes, the less cohesive it likely is. You want to design methods and classes that are highly cohesive. In other words, it should be very clear what a method or class is all about. A method is highly cohesive if it does

one thing and one thing only. For example, in the class `Student` you would have methods to enroll a student in a seminar and to drop a student from a seminar. Both of these methods do one thing and one thing only. You could write one method to do both these functions, perhaps called `changeSeminarStatus`. The problem with this solution is the code for this method would be more complex than the code for the separate `enrollInSeminar` or `dropSeminar` methods. This means your software would be harder to understand and, hence, harder to maintain. Remember that you want to reduce the maintenance burden, not increase it.

A highly cohesive class represents one type of object and only one type of object. For example, for the university information system we model professors, not employees. While a professor is, indeed, an employee, they are very different from other kinds of employees. For example, professors do different things than do janitors, who do different things than do secretaries, who do different things than do registrars, and so on. We could easily write a generic `Employee` class that is able to handle all the functionality performed by every type of employee working for the university. However, this class would quickly become cumbersome and difficult to maintain. A better solution would be to define an inheritance hierarchy made up of the classes `Professor`, `Janitor`, `Secretary`, `Registrar`, and so on. Because many similarities exist between these classes, you would create a new abstract class called `Employee`, which would inherit from `Person`. The other classes, including `Professor`, would now inherit from `Employee`. The advantage of this is each class represents one type of object. If there are ever any changes that need to be made with respect to janitors, you can go right to the class `Janitor` and make them. You do not need to worry about affecting the code for professors. In fact, you do not even need to know anything about professors at all.

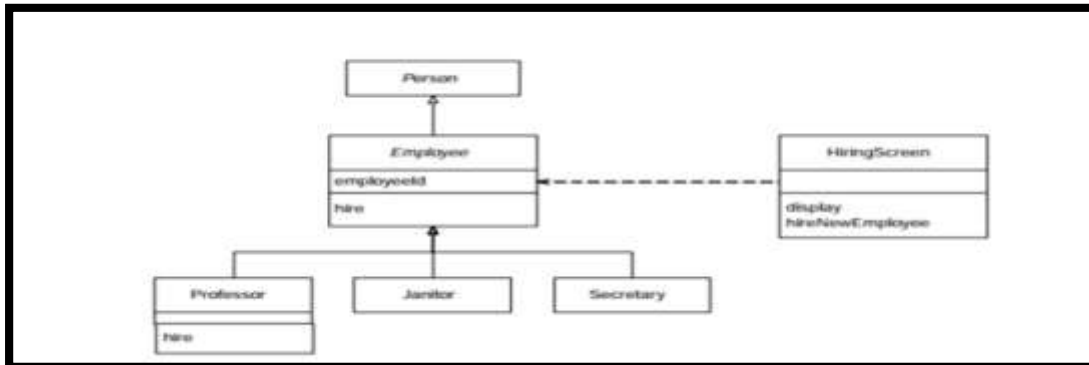
Polymorphism

An individual object may be one of several types. For example, a John Smith object may be a student, a registrar, or even a professor. Should it matter to other objects in the system what type of person John is? It would significantly reduce the development effort if other objects in the system could treat people objects the same way and not need to have separate sections of code for each type. The concept of polymorphism says you can treat instances of various classes the same way within your system. The implication is you can send a message to an object without first knowing what type it is and the object will still do —the right thing, I at least from its point of view.

Polymorphism at University

Consider a slightly more realistic example of polymorphism by exploring the design of how the university handles the hiring of new staff, depicted in the following figure. There is a standard process for hiring staff at the university: once a person is hired, she is added to the university pension plan and an employee card is created for her. When a professor is hired at the university, the same process is

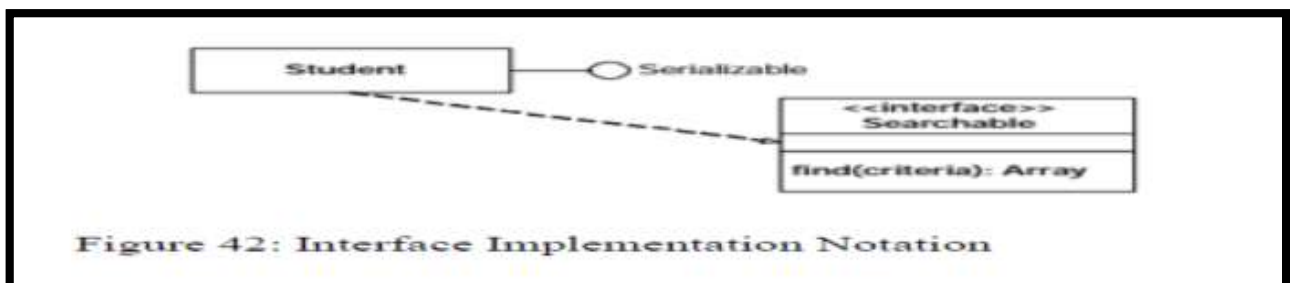
followed. If the hire method has been implemented in the Employee class, it would implement the behavior needed to add the person into the university pension plan and print an employee card for them. The hire method has been overridden in the Professor class. Presumably, it would invoke the hire method in the Employee class because that behavior is still applicable for professors.



By being able to send the message `hire` to any kind of employee, there is not the need for a complicated set of IF or CASE statements in the `hireNewEmployee` method of the screen object. This method does not need to send a `hire Professor` message to professor objects, `hireJanitor` to janitor objects, and so on. It just sends `hire` to any type of employee and the object will do the right thing. As a result, you can add new types of employees (perhaps Registrar) and you do not need to change the screen object at all. In other words, the class is loosely coupled to the employee class hierarchy, enabling you to extend your system easily.

Interfaces

An interface is the definition of a collection of one or more methods, and zero or more attributes. Interfaces ideally define a cohesive set of behaviors. Interfaces are implemented by classes and components. To implement an interface, a class or component must include the methods defined by the interface. For example, the below figure indicates the class **Student** implements the **Serializable** interface and the **Searchable** interface. To implement the **Searchable** interface, **Student** must include a method called `find`, which takes `criteria` as a parameter.



Interfaces are a powerful method for ensuring loose coupling. They allow a class to participate in a common set of functionality without another class having to know anything about it except that it

supports that interface. A GUI object, for example, could present a list of students that meet a set of criteria without knowing anything about the Student class except that it implements the Searchable interface and the name of a single method to get a presentation name, which could perhaps be part of a Description interface. That same GUI could perform the exact same action with professors if the Professor class also implemented those two interfaces. If the Seminar class implemented those interfaces, the GUI could display them, too, and anything else that implemented those interfaces. The GUI knows nothing about these classes except that they implement two interfaces; this promotes loose coupling between the GUI and those classes.

Components

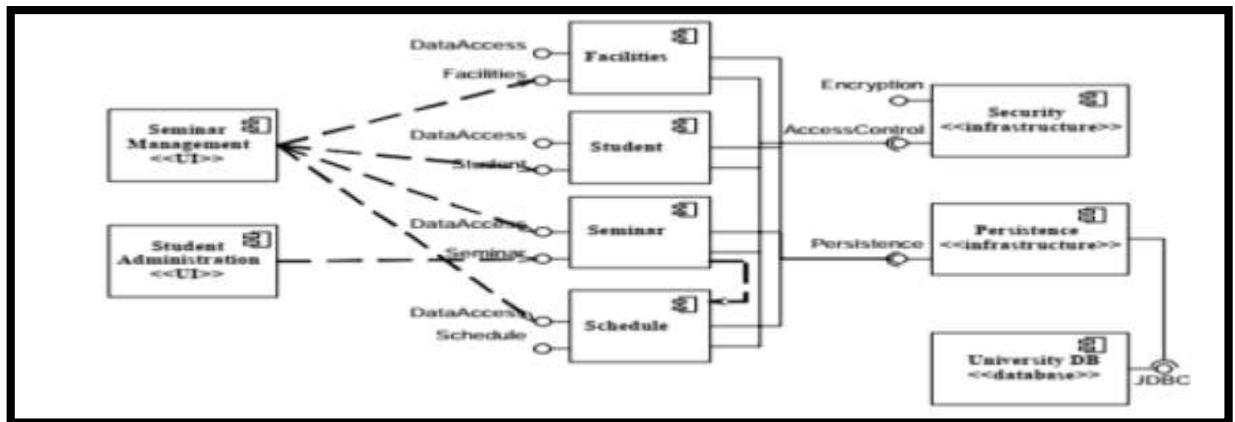
A component is a modular, extensible unit of independent deployment that has contractually specified interface(s) and explicitly defined dependencies, if any. Ideally, components should be modular, extensible, and open. Modularity implies a component contains everything it needs to fulfill its responsibilities; extensibility implies a component can be enhanced to fulfill more responsibilities than it was originally intended to, and open implies it can operate on several platforms and interact with other components through a single programming interface.

Components, like classes, implement interfaces. A component 's interfaces define its access points. Components are typically implemented as collections of classes, ideally classes that form a cohesive subset of your overall systems. Components are typically heavyweights and could even be thought of as large classes or even subsystems. For example, a database could be a component or the collection of business/domain classes that implement the behaviors required to implement people within your application could be a component.

Component diagrams (Object Management Group 2003) show the software components that make up a larger piece of software, their interfaces, and their interrelationships. The following figure shows an example of a component diagram being used to model the business architecture for the university. The boxes represent components—in this case either the user interfaces that people use to interact with the

university systems, business components such as Facility, or technical components such as the Security component or the database.

In the figure you see that the UI



components have dependencies on the interfaces of the business components. By making them dependent on the interfaces and not the components themselves you make it possible to replace the components with different implementations as long as the new version implements the given interfaces. This reduces coupling. Similarly, it is possible for business components to be dependent on each other; for example, the Seminar component is coupled to the Schedule component.

Patterns

Doesn't it always seem as if you are solving the same problems repeatedly? If you personally have not solved a given problem before, then chances are pretty good you could find somebody who had tackled the same or, at least, a similar problem in the past.

Sometimes the problem you are working on is simple, sometimes it is complex, but usually it has been worked on before. Wouldn't it be nice to be able to find a solution easily, or at least a partial solution, to your problem? Think how much time and effort could be saved if you had access to a library of solutions to common system development problems. This is what patterns are all about. A pattern is a solution to a common problem taking relevant forces into account, effectively supporting the reuse of proven techniques and approaches of other developers. Several flavors of patterns exist, including analysis patterns, design patterns, and process patterns. Analysis patterns describe a solution to common problems found in the analysis/business domain of an application, design patterns describe a solution to common problems found in the design of systems, and process patterns address software process related issues. Analysis patterns, design patterns and process patterns are discussed in the coming chapters.

UML Diagram

Although there is far more to modeling than just the UML, as you will see throughout this module, the reality is that the UML defines the standard modeling artifacts when it comes to object technology. There are three classifications of UML diagrams:

Behavior diagrams. This is a type of diagram that depicts behavioral features of a system or business process. This includes activity, state machine, and use case diagrams as well as the interaction diagrams.

Interaction diagrams. This is a subset of behavior diagrams that emphasize object interactions. This includes collaboration, sequence diagrams.

- ✓ **Structure diagrams.** This is a type of diagram that depicts the static elements of a specification that are irrespective of time. This includes class, component, deployment, and package diagrams.

Chapter Four

Gathering user requirements

Requirements identify the objective of a product. It is features of system or system function used to fulfill system purpose. It focuses on customer 's needs and problem, not on solutions. Before anything is done to develop a product, we must identify what the system must do. Therefore, understanding what requirements are, what they are not, and how best to identify them is crucial to the success of the system. Development of a system must follow a set of practices, procedures, rules, and techniques called methodology.

A requirement gathering is also called requirements elicitation. It is a process of collecting the user needs to solve a problem or issues and to achieve an objective. If the requirement gathering is not done properly/ completely, all the subsequent phase is incomplete, no matter how best the design, until and unless requirements are complete. So, we should carefully plan and carry out the requirements gathering with a systematic approach. Gathering user requirements is the first step in software development. Types of requirements can be categorized as follows:

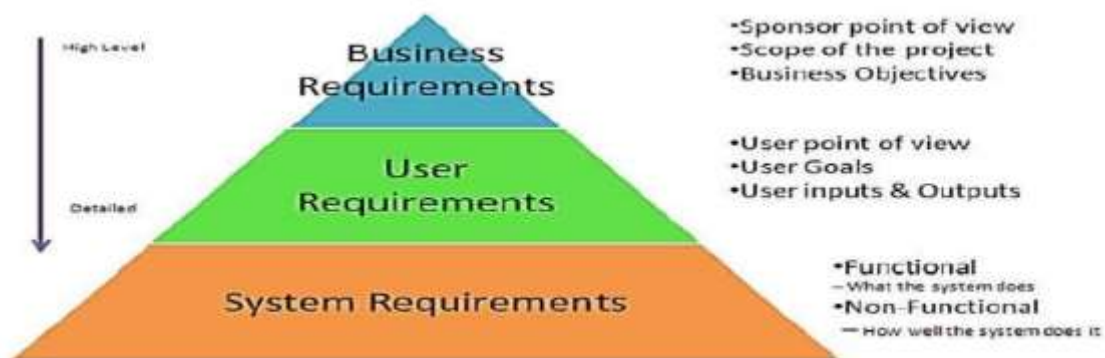


Figure 49: Types of Requirements

Putting together requirements gathering team

Choosing Good Subject-Matter Experts – is a person who has special skills or knowledge on a particular job or topic.

- ✓ Choosing Good Facilitators – is to make easy or ease a process.
- ✓ Choosing Good Scribes – a scribe is a person who copies manuscripts or a pointed instrument used for marking where something should be cut.

Functional Requirements

It specifies the software functionality that the developers must build into the product to enable users to accomplish their tasks, thereby satisfying the business requirements. It is a state what the system must do. In fact, it is usually stated by using the “shall” statement.

Example: The website shall notify the administrator via email when a user register with it

Non – Functional Requirements

Non – functional requirements are Constraints or standards that the system must have or comply with. It defines the system 's quality characteristics. Non – functional requirements describe aspects of the system that are not directly related to the functional behavior of the system.

Nonfunctional requirements include a broad variety of requirements that apply to many different aspects of the system, from usability to performance. These requirements are discussed below:

Usability: is the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component. Usability requirements include, for example, conventions adopted by the user interface, the scope of online help, and the level of user documentation. Often, clients address usability issues by requiring the developer to follow user interface guidelines on color schemes, logos, and fonts.

- ✓ **Reliability:** is the ability of a system or component to perform its required functions under stated conditions for a specified period of time. Reliability requirements include, for example, an acceptable mean time to failure and the ability to detect specified faults or to withstand specified security attacks.
- ✓ **Robustness:** the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions.
- ✓ **Safety:** a measure of the absence of catastrophic consequences to the system.
- ✓ **Performance:** requirements are concerned with quantifiable attributes of the system, such as response time (how quickly the system reacts to a user input).
- ✓ **Supportability:** requirements are concerned with the ease of changes to the system after deployment, including for example, adaptability (the ability to change the system to deal with additional application domain concepts), maintainability (the ability to change the system to deal with new technology or to fix defects), and internationalization (the ability to change the system to deal with additional international conventions, such as languages, units, and number formats).
- ✓ **Portability:** (the ease with which a system or component can be transferred from one hardware or software environment to another)

Fundamental requirements gathering techniques

Essential Use Case Modeling

It is a diagram of all the use-cases, actors and use case-actor association used to describe a particular system. It is semantically closed abstraction of a subject system. A use case model comprises zero or more use case diagrams, although most have at least one diagram, and one or more use-case

specifications (often simply called use cases). Use case diagrams are one of the standard Unified Modeling Language (UML) artifacts. There are two basic types of use case models:

- ✓ **Essential use case models**
- ✓ **System use case model**

Essential use case model often referred to as a task case model or an abstract use case model models a technology independent view of your behavioral requirements, whereas system use case models also known as concrete use case models or detailed use case models, model your analysis of your behavioral requirements, describing in detail how users will work with your system, including references to its user-interface aspects.

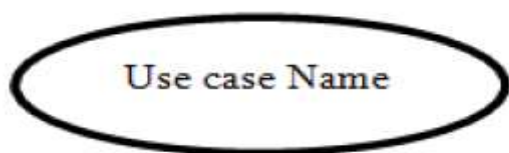
Essential use-case modeling is:

A fundamental aspect of usage-centered designs, an approach to software development.

- ✓ Intended to capture the essence of problems through technology-free, idealized, and abstract descriptions.
- ✓ **More flexible**, leaving open more options and more readily accommodating changes in technology.
- ✓ More robust than concrete representations, simply because they are more likely to remain valid in the face of both changing requirements and changes in the technology of implementation.
- ✓ **Ideal artifacts** to capture the requirements for your system.

Therefore, when you are using an essential use case modeling, you develop a use case diagram, identify essential use cases, and identify potential actors that interact with your system. The use case diagrams depict the core elements:

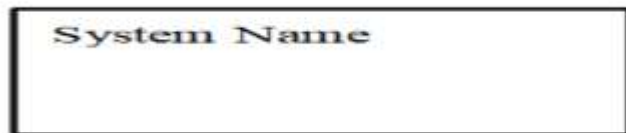
- ✓ **Use Case(s)** – describes a sequence of actions that provide something of measurable value to an actor. It describes the typical ways (or cases) of using the system. Each use case expresses the goal of the actors involved and describes the task that the system, with the assistance of appropriate actors, will perform. You can get the idea of use case 's goal simply by observing its name and associations. It is drawn as a horizontal ellipse as follows:



Actor(s) – is a person, organization, or external system that plays a role in one or more interactions with your system. A list of actors would be a list of all the different roles that people or other systems could play while interacting with the system. Actors in other term we can say users. Few systems operate without interacting with other systems, and many systems interact only with other systems. Actors are outside the system and usually outside the control of the system. Actors are drawn as stick figures as shown below:

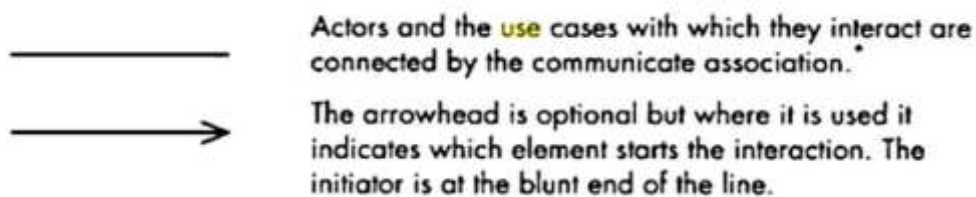


System Boundary (optional) – is used to indicate the scope of your system. It is drawn by a rectangle around the use cases. Anything within the box represents functionality that is in scope, and anything outside the box is not. The symbol looks the following:

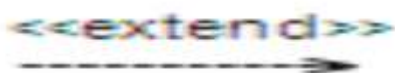


Use case diagram also depicts the core relationships:


Association(s) – it is a relationship between actors and use cases. It is indicated in use case diagrams by solid lines. It exists whenever an actor is involved with an interaction described by a use case. It can also exist between use cases and even between actors, although this is typically an issue for system use case models. A use case has at most one communicates association to a specific actor and an actor has at most one communicates association to a specific use case, no matter how many interactions there are. It is modeled as lines connecting use cases and actors to one another, with an optional arrowhead on one end of the line. The arrowhead is often used to indicate the direction of the initial invocation of the relationship and/or to indicate the primary actor within the use case. It is shown below:



- ✓ **Extend:** a relationship from the extension use case to a base use case specifying how the behavior of extension use case can be inserted into the behavior defined for the base use case. It is represented as follows:



- ✓ **Include:** a relationship from a base use case to inclusion use case specifying how the behavior of the inclusion use case can be inserted into the behavior defined for the base use case. It is represented as follows:

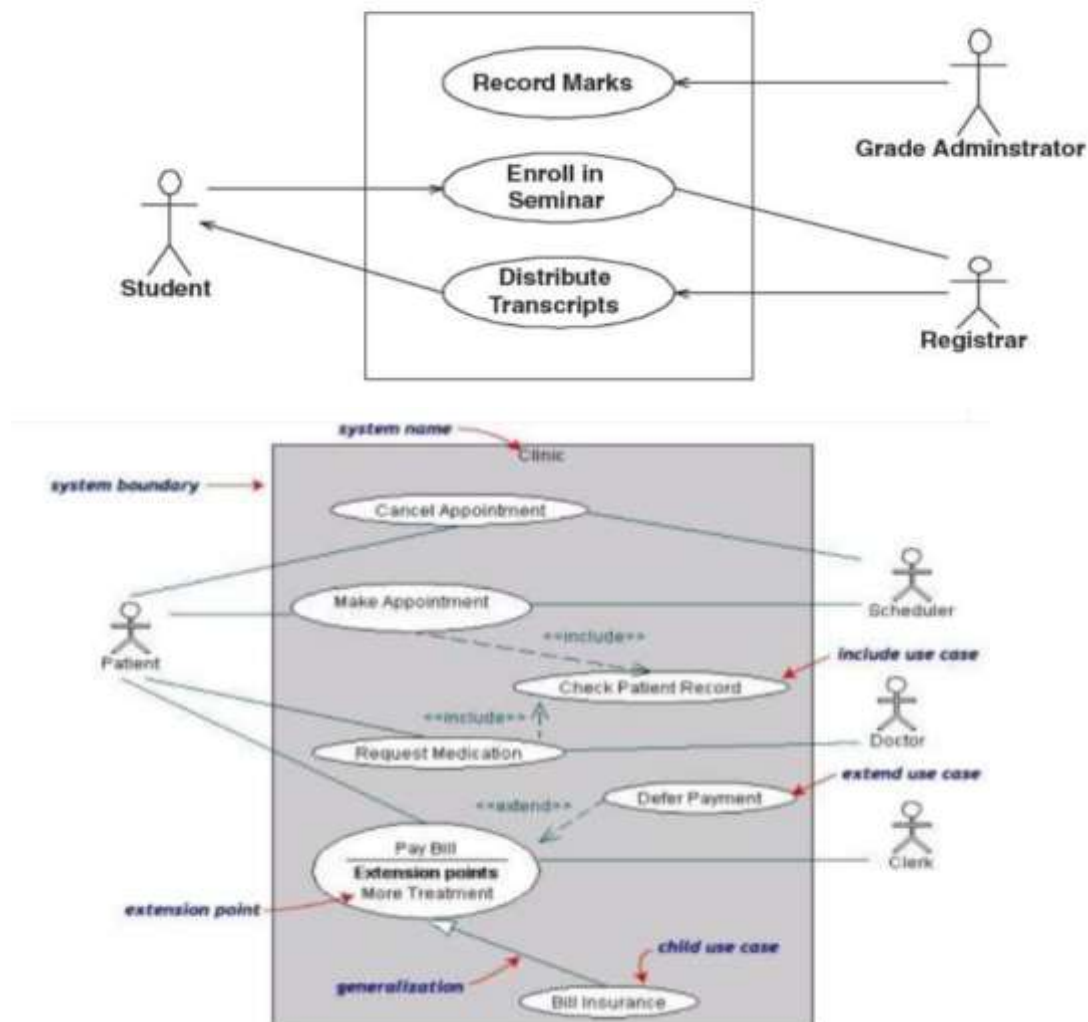
<<include>>


- ✓ **Generalization:** a taxonomic relationship between a more general use case and a more specific use case. It is represented as follows:



Example 1:

Students are enrolling in courses with the potential help of registrars. Professors input the marks students earn on assignments and registrars authorize the distribution of transcripts (report cards) to students. Note how for some use cases there is more than one actor involved. Moreover, note how some associations have arrowheads any given use case association will have a zero or one arrowhead.



Actors are always involved with at least one-use case and are always drawn on the outside edges of a use case diagram.

You should ask how the actors interact with the system to identify an initial set of use cases. Then, on

the diagram, you connect the actors with the use cases with which they are involved.

There are several interesting things to note about the use cases:

- ✓ No time ordering is indicated between use cases.
- ✓ Actors are usually involved in many use cases.
- ✓ Use cases are not functions.
- ✓ Use cases should be functionally cohesive.
- ✓ Each use case should be temporally cohesive.
- ✓ Use cases should describe something of business value.
- ✓ Repetitive actions need not be expressed within a single use case.

Identifying Actors

An actor represents anything or anyone that interfaces with the system. This may include people (not just the end user), external systems, and other organizations. Actors are always external to the system being modeled; they are never part of the system. To help find actors in your system, you should ask yourself the following questions:

- ✓ Who is the main customer of your system?
- ✓ Who obtains, supply, use, remove and provides information from this or to the system?
- ✓ Who installs, operates and shut-down the system?
- ✓ What other systems interact with this system?
- ✓ Does anything happen automatically at a preset time?
- ✓ Where does the system get information?

Example:

If the person employed as the registrar at a university is also taking courses, he or she may play the roles of both Student and Registrar. This is perfectly valid from a use case point of view. To describe an actor, you want to give it a name that accurately reflects its role within your model. Actor names are usually singular nouns, such as Grade Administrator, Customer, and Payment Processor.

Identifying Use Cases

To identify use cases, is to identify potential services by asking your stakeholders the following questions from the point of view of the actors:

- ✓ What are users in this role trying to accomplish?

- ✓ To fulfill this role, what do users need to be able to do?
- ✓ What are the main tasks of users in this role?
- ✓ What information do users in this role need to examine, create, or change?
- ✓ What do users in this role need to be informed of by the system?
- ✓ What do users in this role need to inform the system about?

Example:

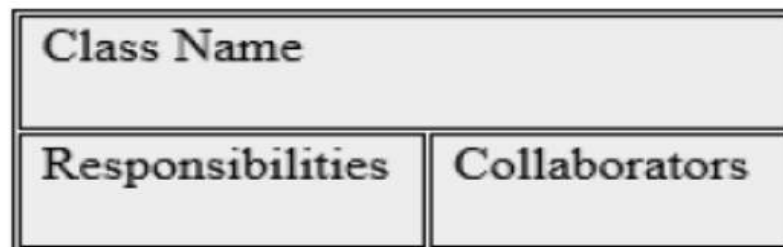
From the point of view of the Student actor, you may discover that students

- ✓ Enroll in, attend, drop, fail, and pass seminars.
- ✓ Need a list of available seminars.
- ✓ Need to determine basic information about a seminar, such as its description and its prerequisites.
- ✓ Obtain a copy of their transcript, their course schedules, and the fees due.
- ✓ Pay fees, pay late charges, receive reimbursements for dropped and cancelled courses, receive grants, and receive student loans.

Essential User Interface Prototyping

Domain modeling with class responsibility collaborator (CRC) cards

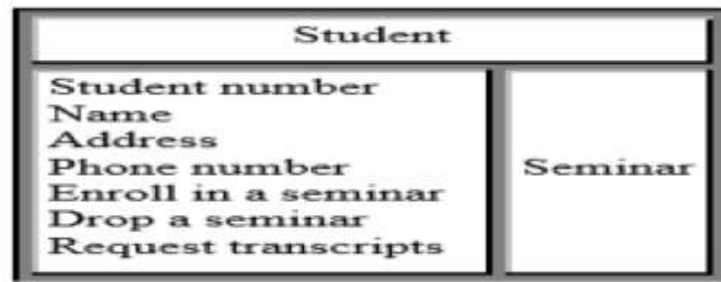
A class responsibility collaborator (CRC) model is a collection of standard index cards that have been divided into three sections, as depicted in figure below:



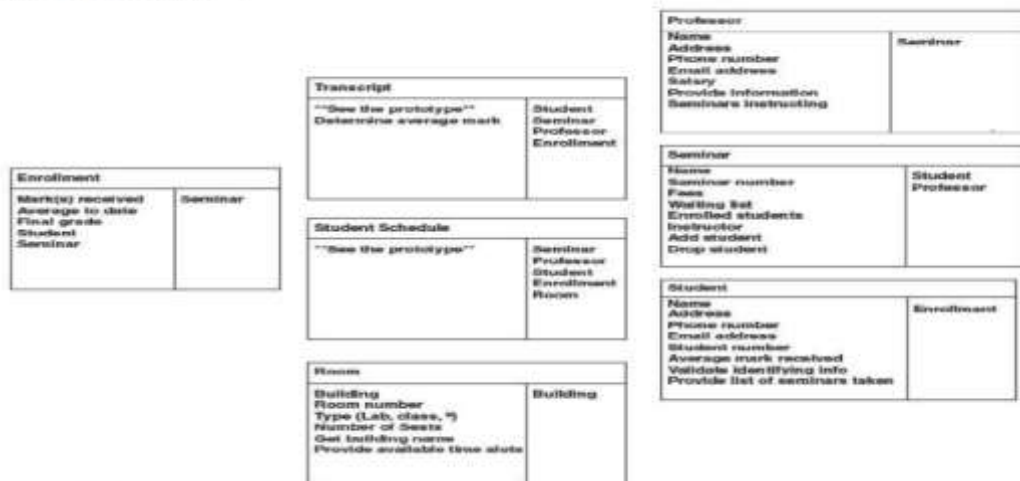
A class represents a collection of similar objects, a responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities.

Example:

In a university system, classes would represent students, tenured professors, and seminars. The CRC models are an incredibly effective tool for conceptual modeling as well as for detailed design. The name of the class appears across the top of a CRC card and is typically a singular noun or singular noun phrase, such as Student, Professor, or Seminar. You use singular names because each class represents a generalized version of a singular object. The information about a student describes a single person, not a group of people. Therefore, it makes sense to use the name Student and not Students. Class names should also be simple.



More CRC Examples:



responsibility is anything that a class knows or does. For instance, Students have names, addresses, and phone numbers. A student knows these things. Students also enroll in seminars, drop seminars, and request transcripts. A student does these things. The things a class knows and does constitute its responsibilities. Important: A class is able to change the values of the things it knows, but it is unable to change the values of what other classes know. Sometimes a class has a responsibility to fulfill, but not have enough information to do it. As you see in figure above, students enroll in seminars. To do this, a student needs to know whether a spot is available in the seminar and, if so, he then needs to be added to the seminar. However, students only have information about themselves (their names and so forth), and not about seminars. What the student needs to do is collaborate/interact with the card labeled Seminar to sign up for a seminar. Therefore, Seminar is included in the list of collaborators of Student. Collaboration takes one of two forms: A request for information or a request to do something. For **example**, the card Student requests an indication from the card Seminar whether a space is available, a request for information.

Student then requests to be added to the Seminar, a request to do something. Another way to perform this logic, however, would have been to have Student simply request Seminar to enroll himself into itself. Then have Seminar do the work of determining whether a seat is available. If the seat is available, then enroll the student; if not, and then inform the student that he was not enrolled. So how do you create CRC models? Iteratively perform the following steps

- ✓ Find Classes

Find Responsibility – you should ask yourself what a class does. Example: Transcripts must be able to calculate their average marks and rooms must be able to indicate when they are available

Define Collaborators – collaboration must occur when a class needs information it does not have. Similarly, collaboration occurs when a class needs to modify information it does not have because any given class can update only the information it knows. This implies that if it needs to have information updated in another class, then it must ask that class to update it.

Move the cards around – CRC modeling is typically performed by a group of people around a large desk. As the CRC cards are created, they are placed on the desk, so everyone can see them. Two cards that collaborate with one another should be placed close together on the table, whereas two cards that do not collaborate should be placed far apart. Furthermore, the more two cards collaborate; the closer they should be on the desk. By having cards that collaborate with one another close together, it is easier to understand the relationships between classes.

Advantages of CRC Cards

- ✓ The experts do the analysis
- ✓ User participation increased
- ✓ Breaks down communication barriers
- ✓ Simple and straightforward
- ✓ Portable
- ✓ Transition – ease of transition from process orientation to object-orientation.

Disadvantages of CRC Cards

- ✓ Inefficient for large systems
- ✓ Limited details
- ✓ Hard to get users together.
- ✓ CRC cards are limited.

Developing a supplementary Specification

- ✓ It is a RUP (Rational Unified Process) document that contains requirements not contained directly in your use cases. This often includes business rules, technical requirements, and constraints. In short, it is a container into which you place other requirements. It is not model in their own right but instead is the documentation of models.

It consists of:

Business Rule (BR): It defines or constrains one aspect of your business that is intended to assert business structure or influence the behavior of your business. It often focuses on access control issues. Some BR focus on the policies of your organization; perhaps the university policy is to

expel for one year anyone who fails more than two courses in the same semester.

Example:

–Professors are allowed to input and modify the marks of the students taking the seminars they instruct, but not the marks of students in other seminars.

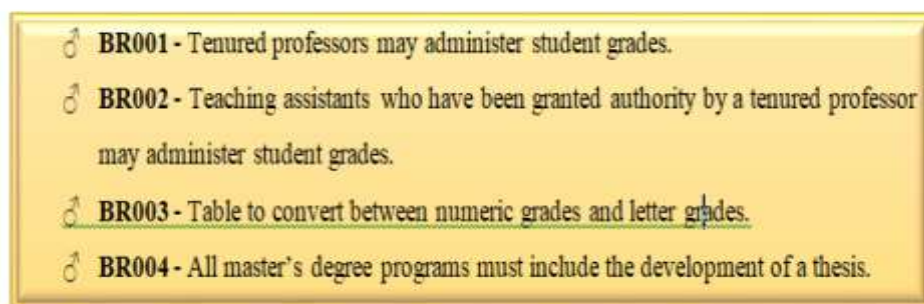
–How to convert a percentage mark (for example, 91 percent) that a student receives in a seminar into a letter grade (for example, A-).

It has a unique identifier. By convention use the format of BR#, but you are free to set your own numbering approach.

The unique identifier enables you to refer easily to business rules in other development artifacts, such as class models and use cases. There are at least three sections of a business rules:

- a. Name – the name should give you a good idea about the topic of the business rule.
- b. Description – the description defines the rule exactly. Although I used text to describe this rule it is quite common to see diagrams such as flow charts or UML activity diagrams used to describe an algorithm.
- c. Example (optional) – an example of the rule is presented to help clarify it.
- d. Source (optional) – the source of the rule is indicated so it may be verified (it is quite common that the source of a rule is a person, often one of your project stakeholders, or a team of people).
- e. Related Rules (optional) – a list of related business rules, if any, is provided to support traceability between rules.
- f. Revision History (optional) – an indication of the date a change was made, the person who made the change, and a description of the change.

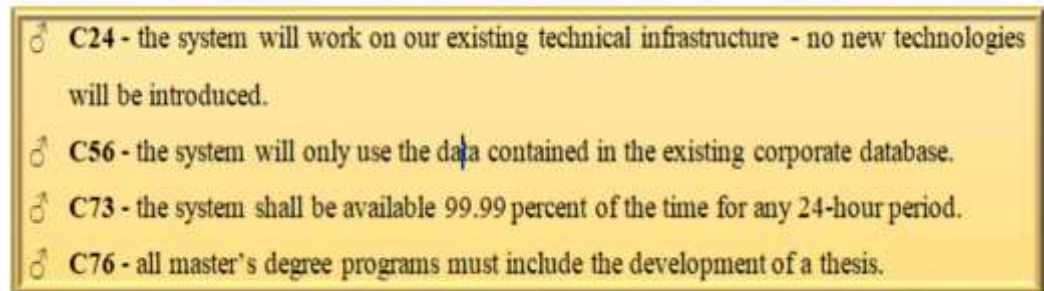
✓ Example:



Constraints: It is a restriction on the degree of freedom you have in providing a solution. It is effectively global requirements, such as limited development resources or a decision by senior management that restricts the way you develop a system. It can be economic, political, technical, or environmental and pertain to your project resources, schedule, target environment, or to the system itself.

Example:

The potential constraints for the university system

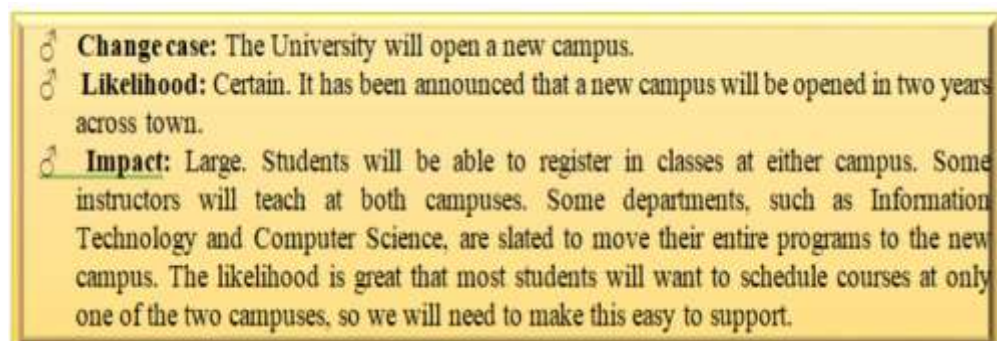


Identifying Change Cases

Change cases are used to describe new potential requirements for a system or modifications to existing requirements. You describe the potential change to your existing requirements, indicate the likeliness of that change occurring, and indicate the potential impact of that change.

Example:

Consider an existing University



- ✓ The name of a change case should describe the potential change itself. Change cases are often the result of brainstorming with your project stakeholders. Good questions to consider include:
- ✓ How can the business change?
- ✓ What is the long-term vision for our organization?
- ✓ What technology can change?
- ✓ What legislation can change?
- ✓ What is your competition doing?
- ✓ What systems will we need to interact with?
- ✓ Who else might use the system and how?

Ensuring Your Requirements Are correct: Requirement validation Techniques

Requirements Validation

- ✓ Checking a work product against higher-level work products or authorities that frame this particular product.
- ✓ It is a heterogeneous process based on application of a great variety of independent techniques.
- ✓ It is nothing more than checking whether the analysts have understood the stakeholders' intention correctly and have not introduced any errors when writing the specification.
- ✓ Requirements are validated by stakeholders.

Requirements Verification

- ✓ Checking a work product against some standards and conditions (specification) imposed on this type of product and the process of its development to allow them to be used effectively to guide further work
- ✓ Requirements are verified by the analysts mainly
- ✓ It is shown graphically as follows:

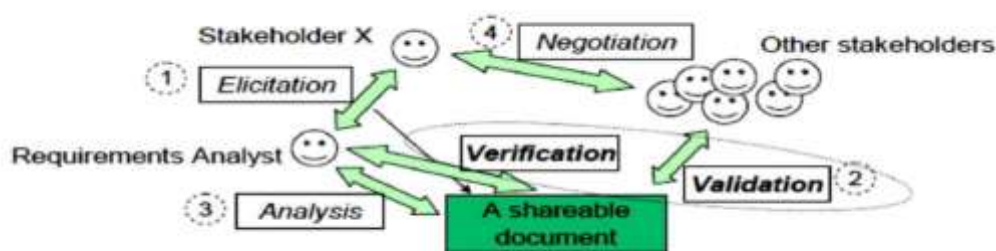
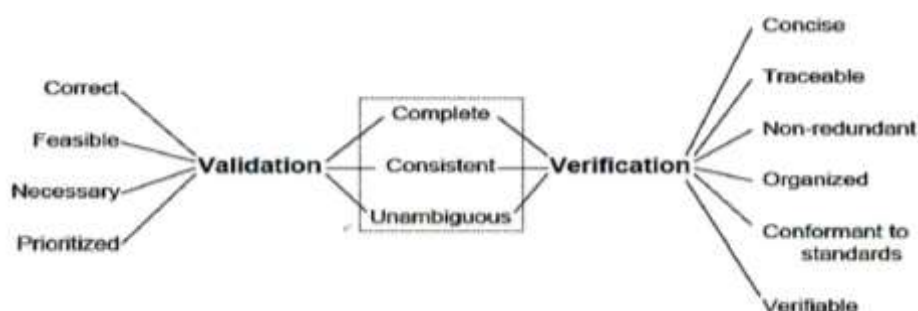


Figure 56: Requirements Verification and validation

- ✓ A stakeholder is a party that has an interest in a company and can either affect or be affected by the business.
- ✓ Requirements elicitation is the practice of researching and discovering the requirements of a system from users, customers, and other stakeholders.
- ✓ Negotiation is an open process for two parties to find an acceptable solution to a complicated conflict.

Characteristics of Requirements Validation and Verification



Requirements Validation Techniques

The most common validation technique that can be performed manually is called review. Three major types of reviews can be differentiated:

- ✓ **Prototypes** – A prototype allows the stakeholders to try out the requirements for the system and experience them thereby: develop the prototype (tool support), Training of the stakeholders, Observation of prototype usage, and Collect issues.
- ✓ **Inspections** – A static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems.
- ✓ **Walk-throughs** – is a meeting where you gather all of your stakeholders together and walk-through the requirements documentation, page-by-page, line-by-line, to ensure that the document represents everyone's complete understanding of what is to be accomplished in this particular project.

Testing Early and Often

Testing is the process of evaluating a system or its components with the intent to find that whether it satisfies the specified requirements or not. It is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual desire or requirements.

Who Tests?



Figure 58: Who Tests?

There are mainly three reasons why we should start testing early and often:

- ✓ With early and regularly tests we have a much higher chance of catching up with our due dates. If we start early, we have more time to resolve discovered defects, and if we test more often, we have a better chance of catching more defects faster, and we are going to have more time to fix them.
- ✓ Testing early and often saves us much effort. Testing early and often can help us to catch errors in very first stages of development. If we cannot or don't find defects soon as we move on in the software development process, it is going to get more complicated and harder for us to discover the defects and we have to put a lot of effort and time to dive into the code and figure it out.

- ✓ It is easier to get back on track. If we start testing soon and often, we can catch the defects fast and in the early stages so if we have to (want to) change our design and approaches to address the defect we have more time and fewer complications ahead of us compare to our options in the later stages.

Methods of Testing

- ✓ **Black Box Testing:** Testing without having any knowledge of the interior workings of the applications. It is also called blind testing or closed box testing. The tester does not have access to the source code.
- ✓ **White Box Testing:** Detailed investigation of internal logic and structure of the code. It is also called glass testing or open box testing. The tester needs to possess knowledge of the internal working of the code.

It is simply called test documentation. A use case represents the actions that are required to enable or abandon a goal. A use case has multiple —paths‖ that can be taken by any user at any one time. Scenario describes of how one or more people or organizations interact with the system. Scenario describes the steps, events, and/or actions which occur during the interaction. A use-case scenario is a single path through the use-case.

A Test Scenario is defined as any functionality that can be tested. It is also called test condition or test possibility. As a tester, you may put yourself in the end user’s shoes and figure out the real- world scenarios and use cases of the application under test.

Use case testing is defined as a software testing technique, that helps identify test cases that cover the entire system, on a transaction by transaction basis from start to the finishing point.

Example 1:

Suppose in a use case for —Login‖ functionality, to access a Gmail of a Web Application. An actor is represented by “A” and system by “S”. The UI is shown as follows:



The image shows a login form with two input fields. The first field is labeled 'Email' and has an envelope icon to its right. The second field is labeled 'Password' and has a padlock icon to its right. Below these fields is a green button with the text 'Log in' in white.

Use Case Scenario Testing

The basic main test cases are:

Main Success Scenario	Step	Description
A:Actor S:System	1	A: Enter Agent Name & Password
	2	S: Validate Password
	3	S: Allow Account Access
Extensions	2a	<u>Password not valid</u> S: Display Message and ask for re-try 4 times
	2b	<u>Password not valid 4 times</u> S: Close Application

Explanation of the test cases:

- ✓ In the first step, the Actor enters email and password for end-to-end scenario of login functionality.
- ✓ In the next step, the system will validate the password.
- ✓ Next, if the password is correct, the access will be granted.
- ✓ There can be an extension of this use case. In case password is not valid system will display a message and ask for re-try four times.
- ✓ If Password, not valid four times system will ban the IP address.

Example 2:

Suppose in a use-case for —Login functionality of an ATM, an actor is represented by “A” and system by “S”. Here below, the test cases and description:

Main Success Scenario A: Actor S: System	Step	Description
	1	A: Inserts card
	2	S: Validates card and asks for PIN
	3	A: Enters PIN
	4	S: Validates PIN
	5	S: Allows access to account
Extensions	2a	Card not valid S: Display message and reject card
	4a	PIN not valid S: Display message and ask for re-try (twice)
	4b	PIN invalid 3 times S: Eat card and exit

Advantages of Use Case Scenario Testing

- ✓ It helps to capture the functional requirements of a system

- ✓ It is traceable.
- ✓ It can serve as the basis for the estimating, scheduling, and validating effort.
- ✓ It can evolve at each iteration from a method of capturing requirements, to development guidelines to programmers, to a test case and finally into user documentation.

Disadvantages of Use Case Scenario Testing

- ✓ Poor identification of structure and flow.
- ✓ Time-consuming to generate.
- ✓ Limited software tool support.
- ✓ Still poor integration with established methods.

Chapter Five

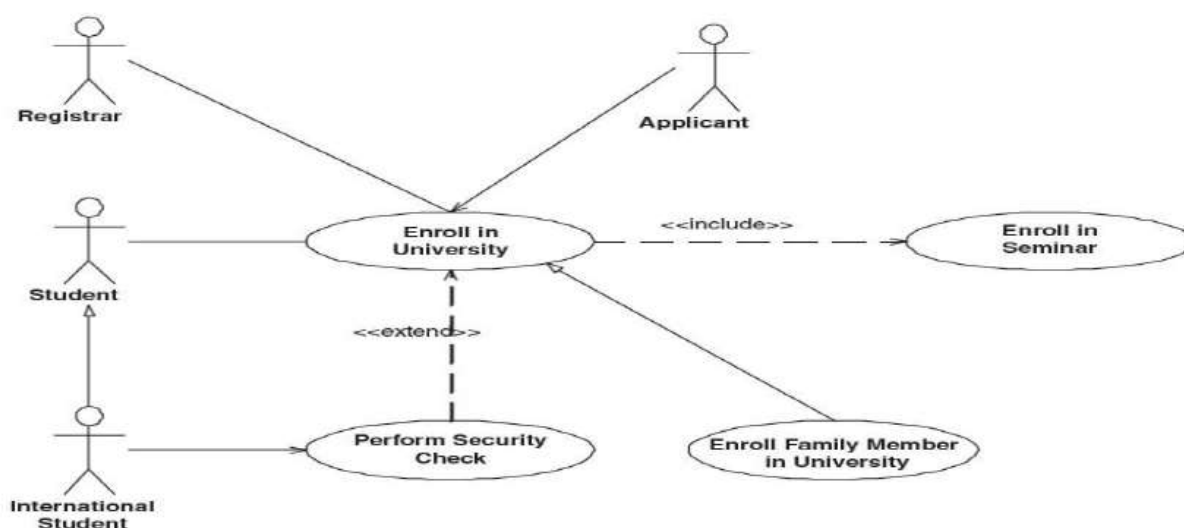
Determining What to Build: OO Analysis

5.1 Introduction

Your requirements model, although effective for understanding what your users want to have built, is not as effective at understanding what will be built. Object-oriented analysis techniques, such as system use-case modeling, sequence diagramming, class modeling, activity diagramming, and user-interface prototyping are used to bridge the gap between requirements and system design. Requirements engineering focuses on understanding users and their usage, whereas analysis focuses on understanding what needs to be built. Therefore, the purpose of analysis is to understand what will be built which is an iterative process. OO Analysis is the process that groups the items that interact with one another, typically by class, attributes, or operations, to create a model that accurately represent the intended purpose of the system as a whole.

5.2 System Use Case Modeling

During analysis, your main goal is to evolve your essential use case into system use cases. The main difference between an essential use case and a system use case is, in the system use case, you include high-level implementation decisions. Example:



A system use case model is composed of a use case diagram and the accompanying documentation describing the use cases, actors, and associations.

5.3 Use Case Documentation (Use Case Description)

The use case has a basic course of action, which is the main start-to-finish path the user will follow.

When you document your use case, the following sections are included:

- ☐ Name- the name of the use case. The name should implicitly express the user's intent of the use case
- ☐ Description- several sentences summarizing the use case
- ☐ Actors [optional]-list of actors associated with the use case
- ☐ Status [optional] – an indication of the status of the use case like work in progress, ready for review, passed review, or failed review
- ☐ Preconditions- a list of conditions, if any, that must be met before a use case may be invoked
- ☐ Post conditions- a list of conditions, if any, that will be true once the use case finishes successfully
- ☐ Extension points [optional]- list of the points in a use case from which other use cases extend
- ☐ Include use cases [optional] a list of use cases this one includes
- ☐ Basic Course of Action- the main path of logic an actor follows through a use case. Often referred to as —happy path or the —main path, because it describes how the use case works when everything works as it normally should
- ☐ Alternative Course of Action- the infrequently used paths of logic in a use case a result of exceptions.

There are two common styles exist for writing use cases:

a. Narrative Style – it is used to write the basic and alternative courses of action are written one step at a time. Example:

Name: Enroll in Seminar

Identifier: UC 17

Description:

Enroll an existing student in a seminar for which she is eligible.

Preconditions:

The Student is registered at the University.

Post conditions:

The Student will be enrolled in the course she wants if she is eligible and room is available.

Basic Course of Action:

1. The use case begins when a student wants to enroll in a seminar.
2. The student inputs her name and student number into the system via *UI23 Security Login Screen*.
3. The system verifies the student is eligible to enroll in seminars at the university according to business rule *BR129 Determine Eligibility to Enroll*. [Alt Course A]
4. The system displays *UI32 Seminar Selection Screen*, which indicates the list of available seminars.
5. The student indicates the seminar in which she wants to enroll. [Alt Course B: The Student Decides Not to Enroll]
6. The system validates the student is eligible to enroll in the seminar according to the business rule *BR130 Determine Student Eligibility to Enroll in a Seminar*. [Alt Course C]

7. The system validates the seminar fits into the existing schedule of the student according to the business rule *BR143 Validate Student Seminar Schedule*.
8. The system calculates the fees for the seminar based on the fee published in the course catalog, applicable student fees, and applicable taxes. Apply business rules *BR 180 Calculate Student Fees* and *BR45 Calculate Taxes for Seminar*.
9. The system displays the fees via *UI33 Display Seminar Fees Screen*.
10. The system asks the student if she still wants to enroll in the seminar.
11. The student indicates she wants to enroll in the seminar.
12. The system enrolls the student in the seminar.
13. The system informs the student the enrollment was successful via *UI88 Seminar Enrollment Summary Screen*.
14. The system bills the student for the seminar, according to business rule *BR100 Bill Student for Seminar*.
15. The system asks the student if she wants a printed statement of the enrollment.
16. The student indicates she wants a printed statement.
17. The system prints the enrollment statement *UI89 Enrollment Summary Report*.
18. The use case ends when the student takes the printed statement.

Alternate Course A: The Student is Not Eligible to Enroll in Seminars.

A.3. The registrar determines the student is not eligible to enroll in seminars.

A.4. The registrar informs the student he is not eligible to enroll.

A.5. The use case ends.

Alternate Course B: The Student Decides Not to Enroll in an Available Seminar

B.5. The student views the list of seminars and does not see one in which he wants to enroll.

B.6. The use case ends.

Alternate Course C: The Student Does Not Have the Prerequisites

C.6. The registrar determines the student is not eligible to enroll in the seminar he chose.

C.7. The registrar informs the student he does not have the prerequisites.

C.8. The registrar informs the student of the prerequisites he needs.

C.9. The use case continues at Step 4 in the basic course of action.

b. Action-Response Style – it is used to present use case steps in columns, one column for each actor and a second column for the system. Example:

Student

1. The student wants to enroll in a seminar.
2. The student inputs his name and student number into the system via "UI23 Security Login Screen."
5. The student indicates the seminar in which she wants to enroll.

System

3. The system verifies the student is eligible to enroll in seminars at the university, according to business rule "BR129 Determine Eligibility to Enroll."
4. The system displays "UI32 Seminar Selection Screen," which indicates the list of available seminars.

The advantage of the action-response style is, it is easier to see how actors interact with the system and how the system responds. The disadvantage is, it is little harder to understand the flow of logic of the use case.

5.4 Sequence Diagram

It is an interaction diagram that shows how the objects and classes involved in the scenario operate with one another and the sequence of messages exchanged. It is used to model the logic of usage scenarios. A usage scenario is exactly what its name indicates – the description of a potential way your system used. The logic of a usage scenario may be part of a use case. It may also be one entire pass through a use case or the logic contained in several use cases.

The boxes across the top of the diagram represent classifiers or their instances, typically use-cases, objects, classes, or actors. Because, you can send messages to both objects and classes, objects respond to messages through the invocation of an operation, and classes do so through the invocation of static operations, it makes sense to include both on sequence diagrams. Because actors initiate and take an active part in usage scenarios, they are also included in sequence diagrams. Objects have labels in the standard UML format “ObjectName: ClassName,” where ‘ObjectName’ is optional (objects that haven’t been given a name on the diagram are called anonymous objects). Classes have labels in the format ‘ClassName,’ with an indication of \llcorner and actors have names in the format ‘ActorName’ with an indication of \llcorner and for major user-interfaces should be stereotyped as \llcorner and the use-case with \llcorner stereotype.

The dashed lines (vertical lines) hanging from the boxes are called object lifelines representing the life span of the object during the scenario being modeled. The long, thin boxes on the lifelines are method-invocation boxes indicating that processing being performed by the target object/ class to fulfill a message. The X at the bottom of a method-invocation box is a UML convention to indicate that an object has been removed from memory, typically the result of receiving a message with the stereotype of \llcorner . Messages are indicated as labeled arrows, when the source and target of a message is an object or class the label is the signature of the method invoked in response to the message. However, if either the source or target is a human actor, the message is labeled with brief text describing the information being communicated. Return values are optionally indicated as using a dashed arrow with a label indicating the return value. Messages fulfill the logic of the steps of the use-case, summarized down the left-hand side of the diagram or on the label.

Generally, there are basically five types of actions explicitly used in a UML sequence diagram.

a. create – is used to create an object. It tells a class to create an instance of itself.

b. call – invokes an operation on an object.

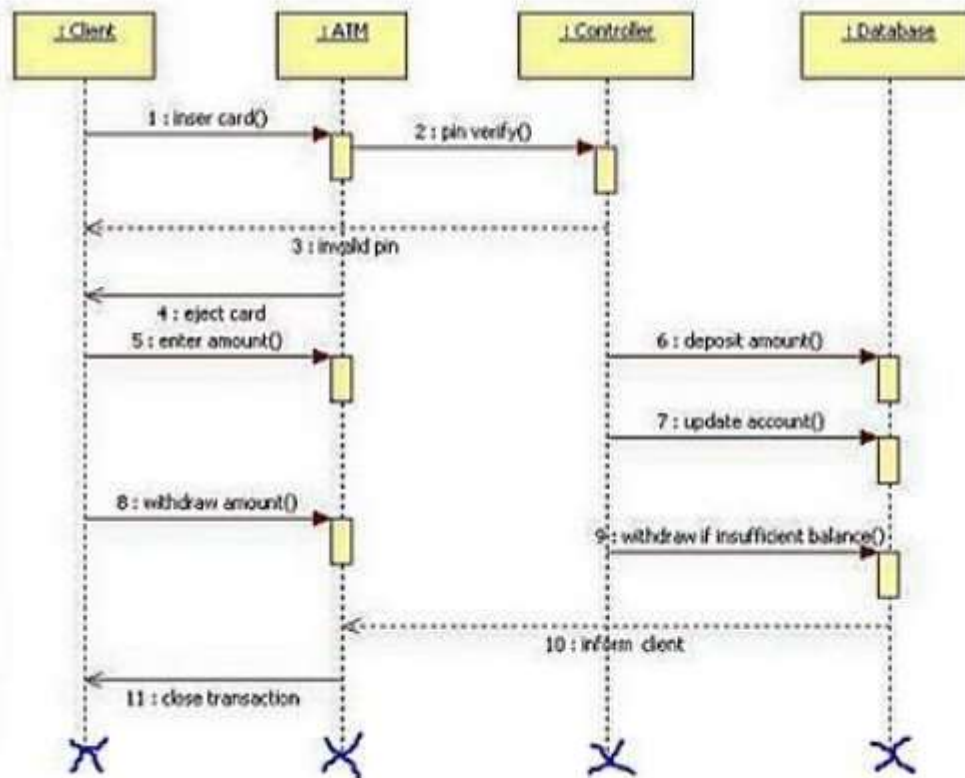
c. send – is used to send a message to an object.

d. return – is the return of a value to the caller, in response to a call action.

e. destroy – is used to destroy an object. Represented with X

Example:

The sequence of actions performed for ATM system. In this case, Client, and ATM are actors and Controller and Database are anonymous object of the concrete classes. The diagram is shown below:



To draw sequence diagram follow the following points:

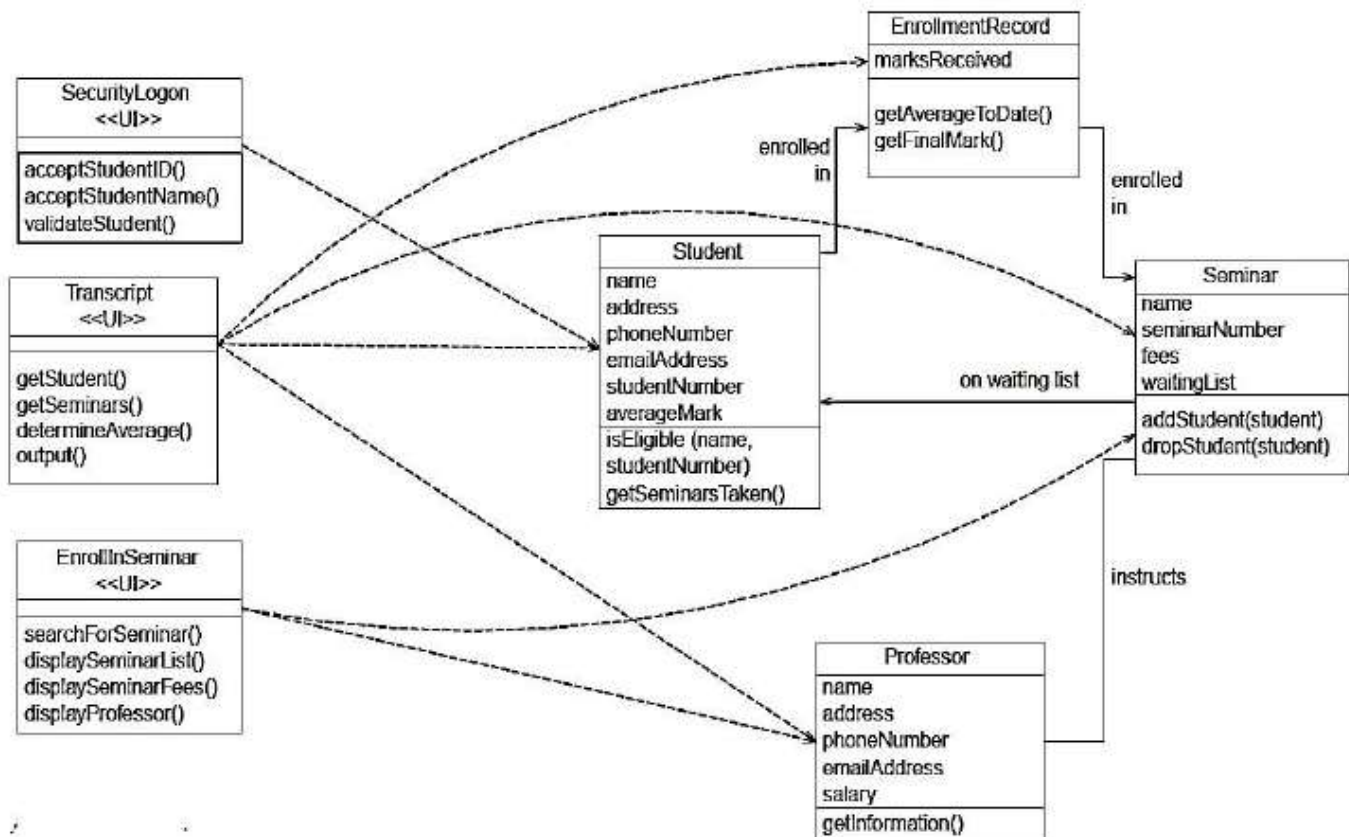
- ☐ Identify the scope of the sequence diagram.
- ☐ List the use-case steps down the left-hand side.
- ☐ Introduce boxes for each actor.
- ☐ Introduce controller class (es).
- ☐ Introduce a box for each major UI element.
- ☐ Introduce a box for each included use-case.
- ☐ Identify appropriate messages for each use-case step.
- ☐ Add method-invocation box for each invocation of a method.
- ☐ Add destruction messages where appropriate.

5.5 Conceptual Modeling: Class Diagram

Class models are the mainstay of the OO analysis and design. Class models show the classes of the system, their interrelationships (including inheritance, aggregation, and association) and the operations and attributes of the classes. The conceptual models are used to depict your detailed understanding of the problem space and solution for your system. The easiest way to begin conceptual modeling is to convert the CRC (as a base) directly to UML class diagram. While a CRC model provides an excellent overview of a system, it doesn't provide the details needed to actually build it.

For each card in the CRC model, you create a concrete class in the class diagram, with the exception of cards that represent actors (actors exist in the real world). The CRC model for the University represented here below:

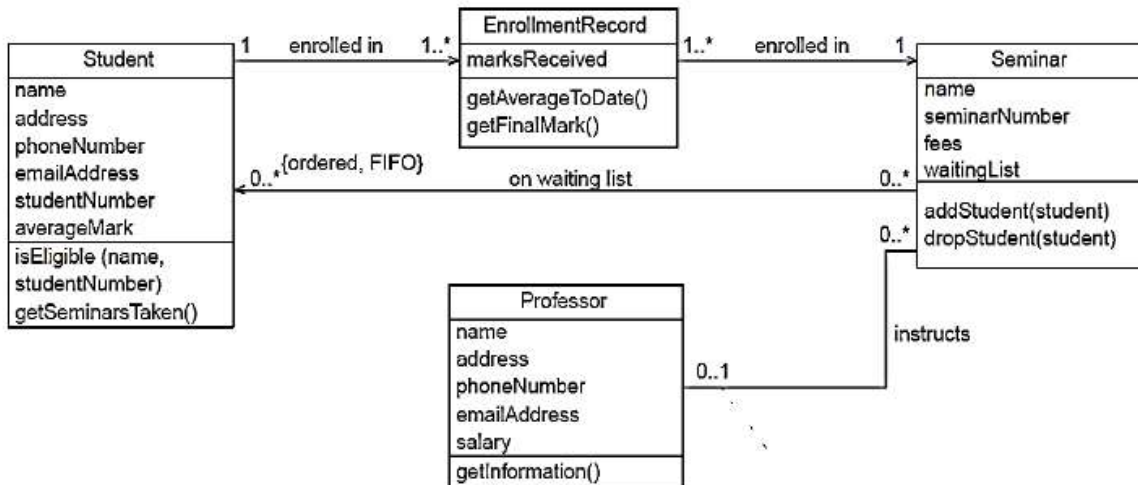
Notice how the names stayed the same (spaces were removed from the names to follow the naming convention of class name). The collaborators on the CRC cards indicate the need for an association, aggregation, association, or dependency between classes. Dependencies are modeled between UI classes and the business classes with which they collaborate because UI classes are transitory in nature, implying the associations they are involved with are transitory. The diagram consists of the association and dependency between classes. Implied the associations they are involved with is transitory, and hence should be modeled as dependencies. The corresponding class model of the CRC model:



The class models contain a wealth of information and can be used for both analysis and design of systems. To create and evolve a class model, you need to model:

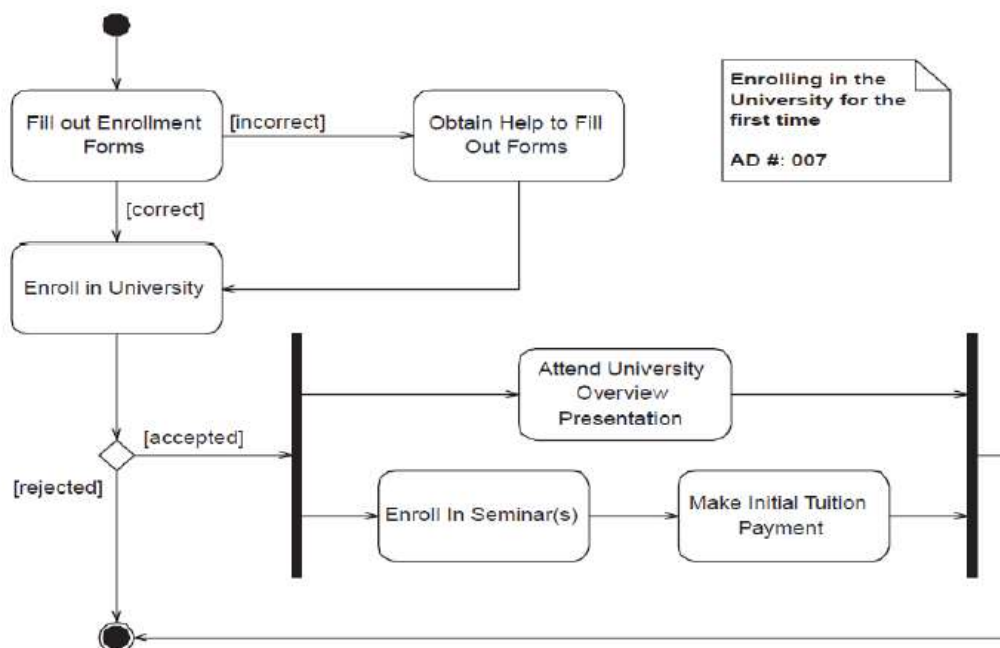
- ☐ Classes
- ☐ Methods or operations
- ☐ Attributes
- ☐ Associations
- ☐ Dependencies

A class is a representation of an object and is simply a template from which objects are created. Classes form the main building blocks of an object-oriented application. Classes are a collection of similar objects. Attributes are the information stored about an object (at least information temporarily maintained about an object), while methods are the things an object or class does. Here below, the concrete classes that made class diagrams:



5.6 Activity Diagram

Activity diagrams are used to document the logic of a single operation/ method, a single use case, or the flow of logic of a business process. In many ways, it is the object-oriented equivalent of flow charts and data-flow diagrams (DFD) from structured development. Example, the activity diagram for how someone new to the university would enroll for the first time:



The filled circle represents the starting point of the activity diagram – effectively a placeholder – and the filled circle with a border represents the ending point. The rounded rectangles represent processes or activities that are performed. The activities map reasonably closely to use-cases. The diamond represents decision points. The decision point had only two possible outcomes. The arrows represent

transitions between activities, modeling the flow order between the various activities. The text on the arrows represent conditions that must be fulfilled to proceed along the transition and are always described using the format [condition]. The thick bars represent the start and end of potentially parallel processes – after you are successfully enrolled in the university, you must attend the mandatory overview presentation and others.

To draw activity diagram consider the following points:

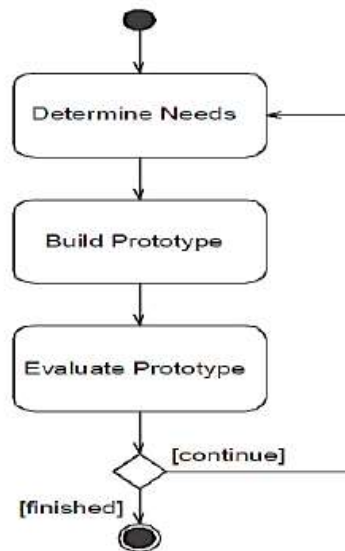
- Identify the scope of the activity diagram. Begin by identifying what you are modeling. Is it a single use case? A portion of a use case? A business process that includes several use-cases? A single method of a class?
- Add start and end points. Every activity diagram has one starting point and one ending point, so you might as well add them right way.
- Add activities. If you are modeling a use-case, introduce an activity for each major step initiated by an actor.
- Add transitions from the activities. It is always to exit from the activity, even if it is simply to an ending point. Whenever there is more than one transition out of an activity, you must label each transition appropriately.
- Add decision points. Sometimes the logic of what you are modeling calls for a decision to be made.
- Identify opportunities for parallel activities. Two activities can occur in parallel when no direct r/ship exists between them and they must both occur before a third activity can.

5.7 User Interface Prototyping Evolving Sour Supplementary Specification

It is an iterative analysis technique in which users are actively involved in the mocking-up of the UI for a system. There are four high-level steps are in the UI prototyping process:

- a. Determine the needs of your users
- b. Build the prototype
- c. Evaluate the prototype
- d. Determine if you are finished.

It is the iterative steps of prototyping, and summed-up as follows:



The following are prototyping Tips and Techniques:

- ☐ Work with the real users.
- ☐ Use a prototyping tool.
- ☐ Get your SMEs (Small-Medium Enterprise) to work with the prototype.
- ☐ Understand the underlying business.
- ☐ Don't spend a lot of time making the code good.
- ☐ Only prototype features that you can actually build.
- ☐ Get an interface expert to help you design it.

5.8 Applying Analysis Patterns Effectively

Analysis pattern describe solutions to common problem found in the analysis/ business domain of a system. It is typically more specific than design patterns, because they describe a solution for a portion of a business domain. This doesn't mean an analysis pattern is applicable only to a single line of business, although it could be. The basic analysis patterns are:

- ☐ **The Business Entity Analysis Pattern:** The basic idea of this pattern is to separate the concepts of a business entity, such as a person or company, from the roles it fulfills. Example: Suppose Mr. Hosty may be a customer of your organization, as well as an employee. Furthermore, one day he may also sell services to your company, also making him a supplier. The person doesn't change, but the role(s) he has with your organization does, so you need to find a way to model this, which is what this pattern does. Each business entity has one or more roles with your organization and each role has a range during which it was applicable (the —start|| and —end|| attributes). Each role implements the behavior specific to it.

- The Contact Point Analysis Pattern: It describes an approach for keeping track of the way your organization interacts with business entities. The basic idea behind this pattern is that surface addresses, email addresses, and phone numbers are really the same sort of thing – a means by which you can contact other business entities.

Some Potential Advantages of Patterns:

- Patterns increase developer productivity.
- Patterns describe proven solutions to common problems.
- Patterns increase the consistency between applications.
- Patterns are potentially better than reusable code.
- More and more patterns are being developed every day.

5.9 Organizing your Models with Packages

Packages are UML constructs that enable you to organize model elements into groups, making your UML diagrams simpler and easier to understand. Packages are depicted as file folders and can be used on any of the UML diagrams, although they are most common on use case diagrams and class diagrams because these models have tendency to grow.

Chapter Six

Determining How to Build Your System: OO Design

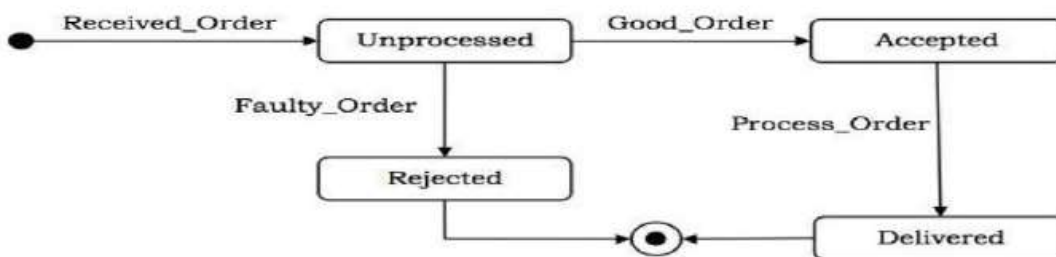
6.1 Layering Your Models

6.1.1 State Chart Modeling

A state chart diagram shows a state machine that depicts the control flow of an object from one state to another. A state machine portrays the sequences of states which an object undergoes due to events and their responses to events. State Chart Diagrams comprise of:

- ☐ States: Simple or Composite
- ☐ Transitions between states
- ☐ Events causing transitions
- ☐ Actions due to the events

State-chart diagrams are used for modeling objects which are reactive in nature. Example: In the Automated Trading House System, let us model Order as an object and trace its sequence. The following figure shows the corresponding state chart diagram:



6.1.2 Collaboration Modeling

Collaboration diagrams are interaction diagrams that illustrate the structure of the objects that send and receive messages. In these diagrams, the objects that participate in the interaction are shown using vertices. The links that connect the objects are used to send and receive messages. The message is shown as a labeled arrow. Sequence is shown by including a sequence number on the message.

A collaboration diagram is formed by:

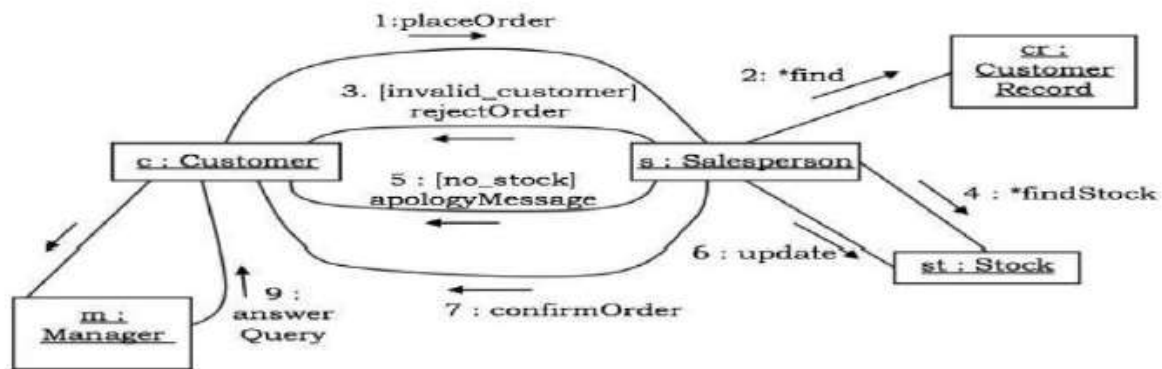
- ☐ Placing the objects that participate in the interaction as the vertices in a graph
- ☐ Rendering the links that connect these objects as the arcs of this graph
- ☐ Adorning these links with the messages that the objects send and receive

Collaboration diagram has four key elements:

- ☐ Objects appear at the vertices of the graph
- ☐ Paths that indicate how one object is linked to another

- Sequence number to indicate the time order of a message
- Messages show the actions that objects perform on each other and on themselves

Example, collaboration diagram for the Automated Trading House System is illustrated in the figure below:

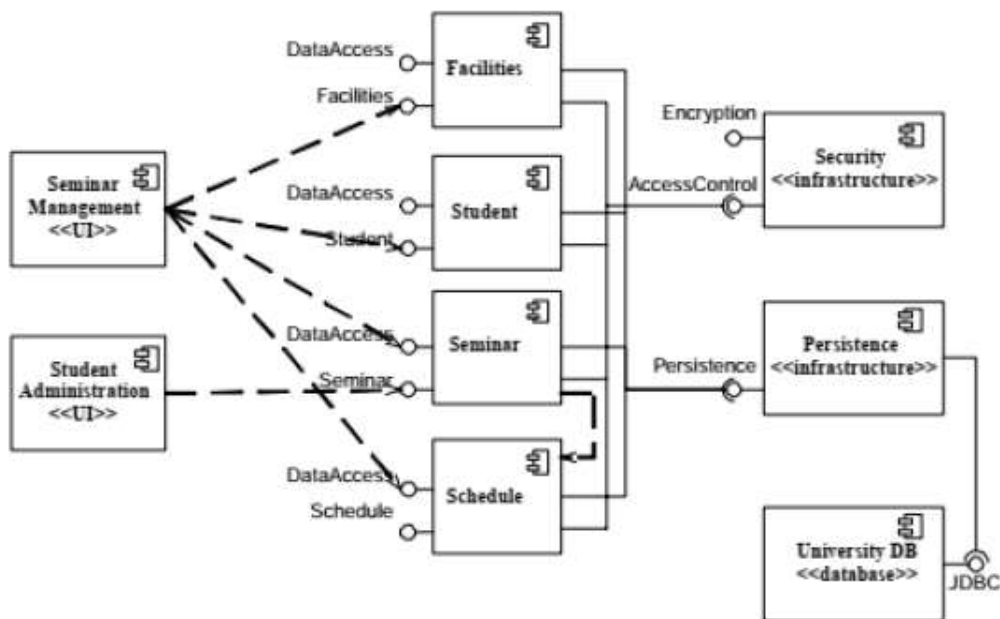


6.1.3 Component Modeling

Component diagrams show the organization and dependencies among a group of components.

Component diagrams comprise of:

- Components
- Interfaces
- Relationships
- Packages and Subsystems (optional) Component diagrams are used for: Constructing systems through forward and reverse engineering. Modeling configuration management of source code files while developing a system using an object-oriented programming language.
- Representing schemas in modeling databases.
- Modeling behaviors of dynamic systems.



6.1.4 Deployment Modeling

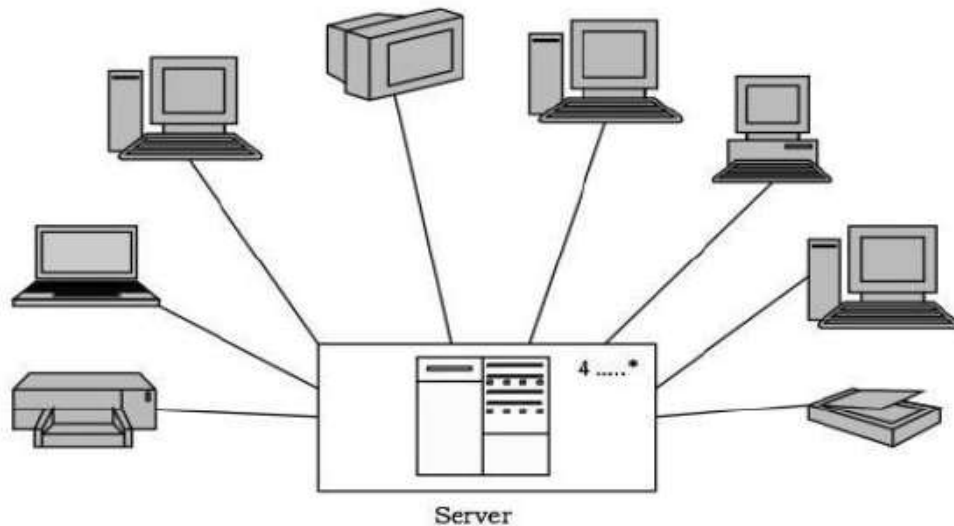
A deployment diagram puts emphasis on the configuration of runtime processing nodes and their components that live on them. They are commonly comprised of nodes and dependencies, or associations between the nodes. Deployment diagrams are used to:

- Model devices in embedded systems that typically comprise of software-intensive collection of hardware.
- Represent the topologies of client/server systems.
- Model fully distributed systems.

Example: The following figure shows the topology of a computer system that follows client/server architecture. The figure illustrates a node stereotyped as server that comprises of processors. The figure indicates that four or more servers are deployed at the system.

Connected to the server are the client nodes, where each node represents a terminal device such as workstation, laptop, scanner, or printer.

The nodes are represented using icons that clearly depict the real-world equivalent as shown below:



6.1.5 Persistence Modeling

If a system wants to have the data irrespective of a single execution instance, there will be a need to have a facility to put the data permanently. The persistent objects are often extracted from the application domain. For example, for a sales system you need to keep the customer, sales items, sales invoices, and similar domain objects persistently. One of the options to store data is through the use of relational databases which needs transformation of the class diagrams (from class diagrams to RDBs). Rules to transform/convert class diagrams to the relational database:

1. Classes are converted/mapped to tables. The attributes become columns
2. Inheritance:
 - ☐ Map in to a single table(type attribute could be used to differentiate the two classes)
 - ☐ Map the sub classes in to tables
 - ☐ Map all classes in to tables
3. Mapping relationship:
 - ☐ One-to-one: Maintain through a foreign key in either of the table
 - ☐ One-to-many: Maintained through a foreign key on the many side
 - ☐ Many-to-Many: Maintain it through another intermediate associate table

6.2 Applying Design Patterns Effectively

While designing applications, some commonly accepted solutions are adopted for some categories of problems. These are the patterns of design. A pattern can be defined as a documented set of building

blocks that can be used in certain types of application development problems. Some commonly used design patterns are:

- ☐ Façade pattern
- ☐ Model view separation pattern
- ☐ Observer pattern
- ☐ Model view controller patter
- ☐ Publish subscribe pattern
- ☐ Proxy pattern

Chapter Seven

Object Oriented Testing and Maintenance

8.1 Introduction

No program or system design is perfect. Communication between the user and the designer is not always complete or clear and time is usually short. The result is errors. The number and nature of errors in a new design depend on several factors.

1. Communication between the user and the designer.
2. The programmer's ability to generate a code that reflects exactly the system specifications.

These factors put an increasing burden on systems analysts to ensure the success of the system developed. The quality of a system depends on its design, development, testing and implementation.

8.2 Maintenance of Systems

When systems are installed, they generally are used for long periods. The average life of a system is 4 to 6 years, with the oldest application often in use for over 10 years. However, this period of use brings with it the need to continually maintain the system. Because of the use a system receives after it is fully implemented, analysts must take precautions to ensure that the need for maintenance is controlled through design and testing and the ability to perform it is provided through proper design practices.

The keys to reducing the need for maintenance, while making it possible to do essential tasks more efficiently, are these:

1. More accurately defining the user's requirements during systems development.
2. Assembling better system documentation.
3. Using more effective methods for designing processing logic and communicating it to project team members.
4. Making better use of existing tools and techniques.
5. Managing the systems engineering process effectively.

8.3 Managing Quality Assurance

Quality assurance is the review of software products and related documentation for completeness, correctness, reliability, and maintainability. And, of course, it includes assurance that the system meets the specifications and the requirements for its intended use and performance.

8.3.1 Levels of Assurance

Analysts use four levels of quality assurance: testing, verification, validation, and certification.

a) Testing

Systems testing is an expensive but critical process that can take as much as 50 percent of the budget for program development. The common view of testing held by users is that it is performed to prove that there are no errors in a program. However, this is virtually impossible, since analysts cannot prove that software is free and clear of errors.

Therefore, the most useful and practical approach is with the understanding that testing is the process of executing a program with explicit intention of finding errors that is, making the program fail. The tester, who may be an analyst, programmer, or specialist trained in software testing, is actually trying to make the program fail. A successful test, then, is one that finds an error.

Analysts know that an effective testing program does not guarantee systems reliability. Reliability is a design issue. Therefore, reliability must be designed into the system. Developers cannot test for it.

b) Verification and validation

Like testing, verification is also intended to find errors. Executing a program in a simulated environment performs it. Validation refers to the process of using software in a live environment on order to find errors.

When commercial systems are developed with the explicit intention of distributing them to dealers for sale or marketing them through company – owned field offices, they first go through verification, sometimes called alpha testing. The feedback from the validation phase generally produces changes in the software to deal with errors and failures that are uncovered.

c) Certification

Software certification is an endorsement of the correctness of the program, an issue that is rising in importance for information systems applications. There is an increasing dependence on the purchase or

lease of commercial software rather than on its in-house development. However, before analysts are willing to approve the acquisition of a package, they often require certification of the software by the developer or an unbiased third party.

8.4. Testing Plans

The philosophy behind testing is to find errors. Test cases are devised with this purpose in mind. A test case is a set of data that the system will process as normal input. However, the data are created with the express intent of determining whether the system will process them correctly.

8.4.1 Code Testing

The code-testing strategy examines the logic of the program. To follow this testing method, the analyst develops test cases that result in executing every instruction in the program or module; that is, every path through the program is tested. A path is a specific combination of conditions that is handled by the program.

However, even if code testing can be performed in its entirety, it does not guarantee against software failures. This testing strategy does not indicate whether the code meets its specifications nor does it determine whether all aspects are even implemented.

8.4.2 Specification Testing

To perform specification testing, the analyst examines the specifications stating what the program should do and how it should perform under various conditions. Then test cases are developed for each condition or combination of conditions and submitted for processing. By examining the results, the analyst can determine whether the program performs according to its specified requirements.

This strategy treats the program as if it were a black box: the analyst does not look into the program to study the code and is not concerned about whether every instruction or path through the program is tested. In that sense, specification testing is not complete testing. However, the assumption is that, if the program meets the specifications, it will not fail.

Neither code nor specification testing strategy is ideal. However, specification testing is a more efficient strategy, since it focuses on the way software is expected to be used. It also shows once again how important the specifications developed by the analysts are throughout the entire systems development process.

8.4.3 Unit Testing

In unit testing the analyst tests the programs making up a system. (For this reason unit testing is sometimes called program testing.) The software units in a system are the modules and routines that are assembled and integrated to perform a specific function. In a large system, many modules at different levels are needed.

Unit testing focuses first on the modules, independently of one another, to locate errors. This enables the tester to detect errors in coding and logic that are contained within that module alone. Those resulting from the interaction between modules are initially avoided.

Unit testing can be performed from the bottom up, starting with the smallest and lowest – level modules and proceeding one at a time. For each module in bottom-up testing, a short program (called a driver program because it drives or runs the module) executes the module and provides the needed data, so that the module is asked to perform the way it will when embedded within the larger system. When bottom-level modules are tested, attention turns to those on the next level that use the lower – level ones. They are tested individually and then linked with the previously examined lower – level modules.

Top-down testing, as the name implies, begins with the upper – level modules. However, since the detailed activities usually performed in lower-level routines are not provided (because those routines are not being tested), stubs are written. A stub is a module shell that can be called by the upper – level module and that, when reached properly, will return a message to the calling module, indicating that proper interaction occurred. No attempt is made to verify the correctness of the lower-level module.

8.4.4 Integration testing

Integration testing does not test the software per se but rather the integration of each module in the system. It also tests to find discrepancies between the system and its original objective, current specifications, and systems documentation. The primary concern is the compatibility of individual modules. Analysts are trying to find areas where modules have been designed with different specifications for data length, type, and data element name.

Integration testing must also verify that file sizes are adequate and that indices have been built properly. Sorting and reindexing procedures assumed to be present in lower-level modules must be tested at the systems level to see that they in fact exist and achieve the results modules expect.