Jaipur Engineering College & Research Centre, Jaipur

Notes

Software Engineering

[3CS4 - 07]

Prepared By:
Manju Vyas

SYLLABUS

UNIT 1: Introduction, software life-cycle models, software requirements specification, formal requirements specification, verification and validation.

UNIT 2: Software Project Management: Objectives, Resources and their estimation, LOC and FP estimation, effort estimation, COCOMO estimation model, risk analysis, software project scheduling.

UNIT 3: Requirement Analysis: Requirement analysis tasks, Analysis principles. Software prototyping and specification data dictionary, Finite State Machine (FSM) models. **Structured Analysis**: Data and control flow diagrams, control and process specification behavioral modeling

UNIT 4: Software Design: Design fundamentals, Effective modular design: Data architectural and procedural design, design documentation.

UNIT 5: Object Oriented Analysis: Object oriented Analysis Modeling, Data modeling. **Object Oriented Design**: OOD concepts, Class and object relationships, object modularization, Introduction to Unified Modeling Language

INTRODUCTION TO SOFTWARE ENGINEERING

The term software engineering is composed of two words, software and engineering.

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

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods. So, we can define software engineering as an engineering branch associated with the development of software product using well-defined scientific

principles, methods and procedures. **The outcome of software engineering is an efficient and reliable software product.**

IEEE defines software engineering as: The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are absolutely necessary to achieve a good quality software cost effectively.

NEED OF SOFTWARE ENGINEERING

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

Large software - It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.

Scalability- If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.

Cost- As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

Dynamic Nature- The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.

Quality Management- Better process of software development provides better and quality software product.

CHARACTERESTICS OF GOOD SOFTWARE

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

- **Operational**
- **Transitional**
- **Maintenance**

Well-engineered and crafted software is expected to have the following characteristics:

Operational: This tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency
- Correctness
- Functionality
- Dependability
- Security
- Safety

Transitional: This aspect is important when the software is moved from one platform to another:

- Portability

- Interoperability
- Reusability
- Adaptability

Maintenance: This aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products.

LAYERS OF SOFTWARE ENGINEERING

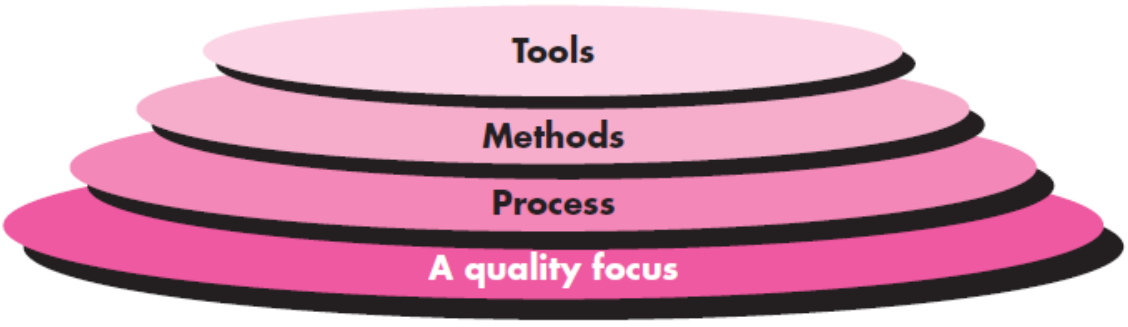


Figure: Layers Of Software Engineering

Software engineering is a layered technology. Software engineering must rest on an organizational commitment to **quality**. Total quality management, Six Sigma, and similar

philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a **quality focus.**

A **Software engineering process** is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

Software engineering methods provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering tools provide automated or semiautomated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

THE SOFTWARE PROCESS

A **process** is a collection of activities, actions, and tasks that are performed when some work product is to be created.

An **activity** strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.

An **action** (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).

A **task** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

In the context of software engineering, a process is not a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks.

The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

A GENERIC PROCESS FRAMEWORK FOR SOFTWARE ENGINEERING: A **process framework** establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.

In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire software process.

A generic process framework for software engineering encompasses five activities:

Communication: Before any technical work can commence, it is critically important to communicate and collaborate with the customer and other stakeholders and stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning: Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a software project plan—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling: Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're

going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction: This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment: The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

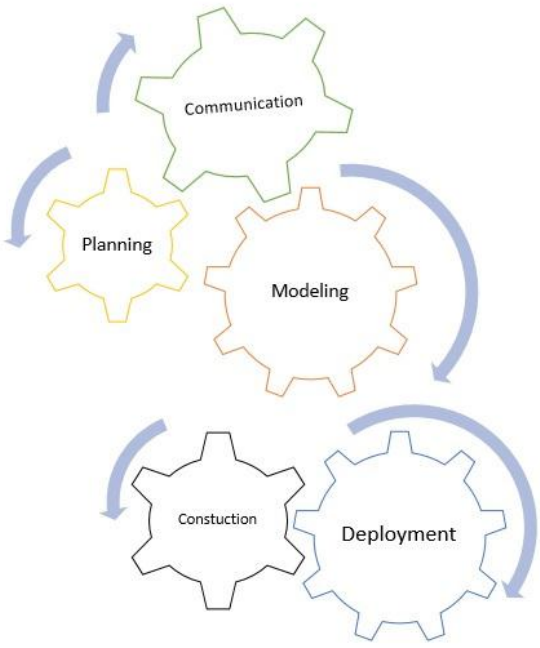


Figure: Activities in Generic process framework for software engineering

SOFTWARE LIFE CYCLE MODELS/SOFTWARE DEVELOPMENT LIFE CYCLE MODEL (SWDLC MODELS)

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle.

A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement.

Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out.

THE NEED FOR A SOFTWARE LIFE CYCLE MODEL

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner.

When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure.

This problem can be illustrated by using an example. Suppose a software development problem is divided into several parts and the parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure.

A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized. Without software life cycle models it becomes difficult for software project managers to monitor the progress of the project.

Different software life cycle models

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

Types of Software developing life cycles (SDLC)

- Waterfall Models
- Iterative Waterfall Model
- Incremental Model
- Prototyping Model
- Spiral Method
- Agile development

WATERFALL MODEL

The Waterfall Model is a **LINEAR SEQUENTIAL MODEL**. In which progress is seen as flowing steadily downwards (like a waterfall) through the phases of software implementation. This means that any phase in the development process begins only if the previous phase is complete. The waterfall approach does not define the process to go back to the previous phase to handle changes in requirement. The waterfall approach was the earliest approach and most widely known that was used for software development.

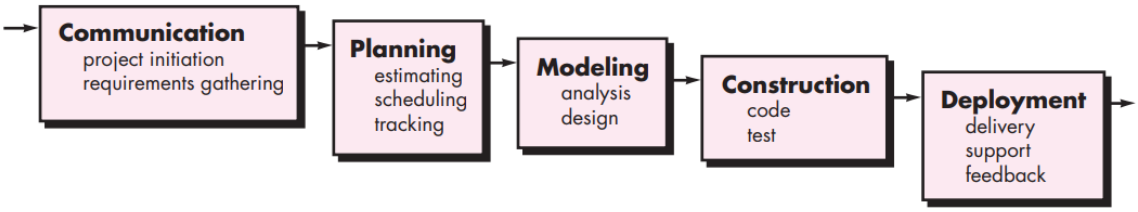


Figure: Generic Waterfall Model

Phases of Waterfall model: All work flows from **communication** towards **deployment** in a reasonably linear fashion.

- **Communication**: includes **project initiation** and **requirements gathering** activities.
- **Planning:** includes **estimating**, **scheduling** and **tracking** activities.
- **Modeling:** includes **analysis** and **design** activities.
- **Construction:** includes **coding** and **testing** activities.
- **Deployment:** includes product **delivery**, **support** and **feedback** activities.

ITERATIVE WATERFALL MODEL/ WATERFALL MODEL WITH FEEDBACK

To overcome the major shortcomings of the classical waterfall model, we come up with the iterative waterfall model.

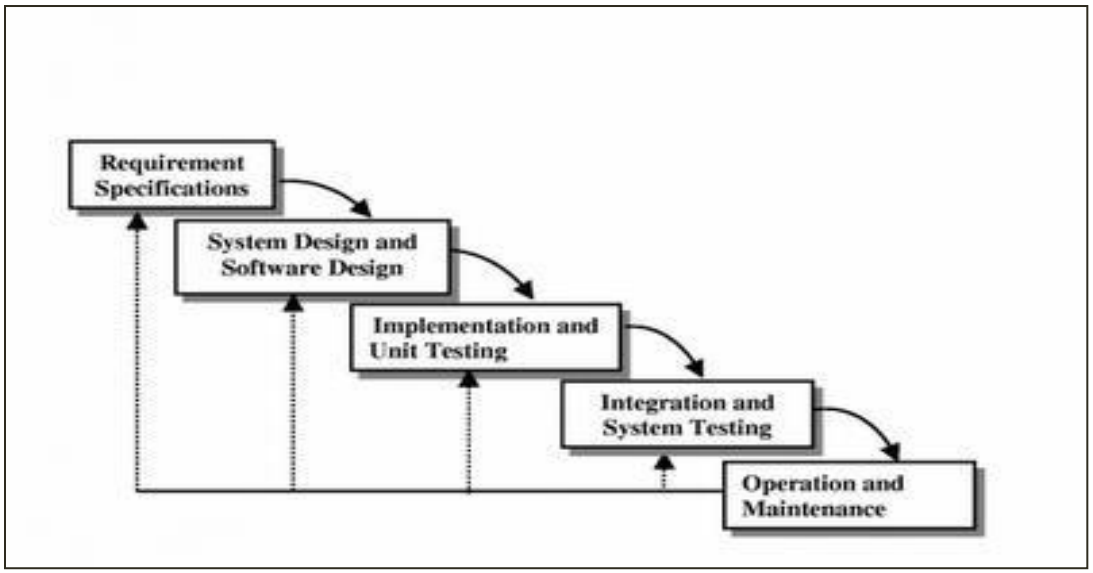


Figure : Iterative Waterfall Model

Here, we provide feedback paths for error correction as & when detected later in a phase. Though errors are inevitable, but it is desirable to detect them in the same phase in which they occur. If so, this can reduce the effort to correct the bug.

The advantage of this model is that there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.

The disadvantage with this SWDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

THE V-MODEL

A variation in the representation of the waterfall model is called the V-model. Represented in Figure, the V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities.

As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.

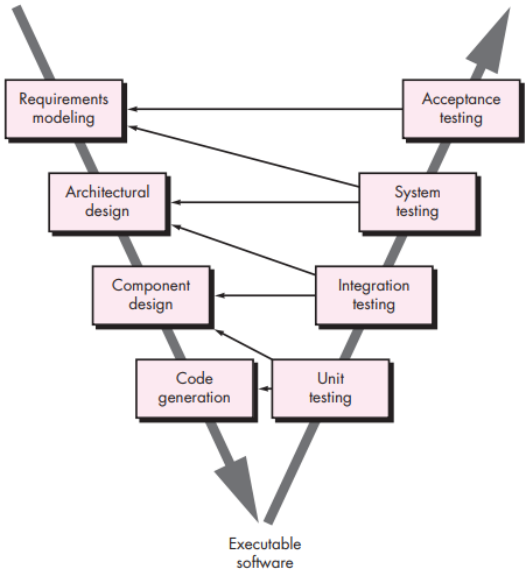


FIGURE: V- MODEL

Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.

In reality, there is no fundamental difference between the classic waterfall life cycle and the V-model.

The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

The problems encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

EVOLUTIONARY PROCESS MODELS

INCREMENTAL PROCESS MODELS

There are many situations in which initial software requirements are well defined, but it is not possible to follow a purely linear process. In addition, there may be a need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

The incremental model combines elements of linear and parallel process flows and applies linear sequences in a stepwise manner according to calendar. **Each linear sequence produces deliverable "*increments*"** of the software.

For example, word-processing software developed using the incremental model may deliver:

- basic file management, editing, and document production functions in the **first increment**;
- more sophisticated editing and document production capabilities in the **second increment**;
- spelling and grammar checking in the **third increment**;

- and advanced page layout capability in the **fourth increment**.

It should be noted that the process flow for any increment can incorporate the ***prototyping methods.***

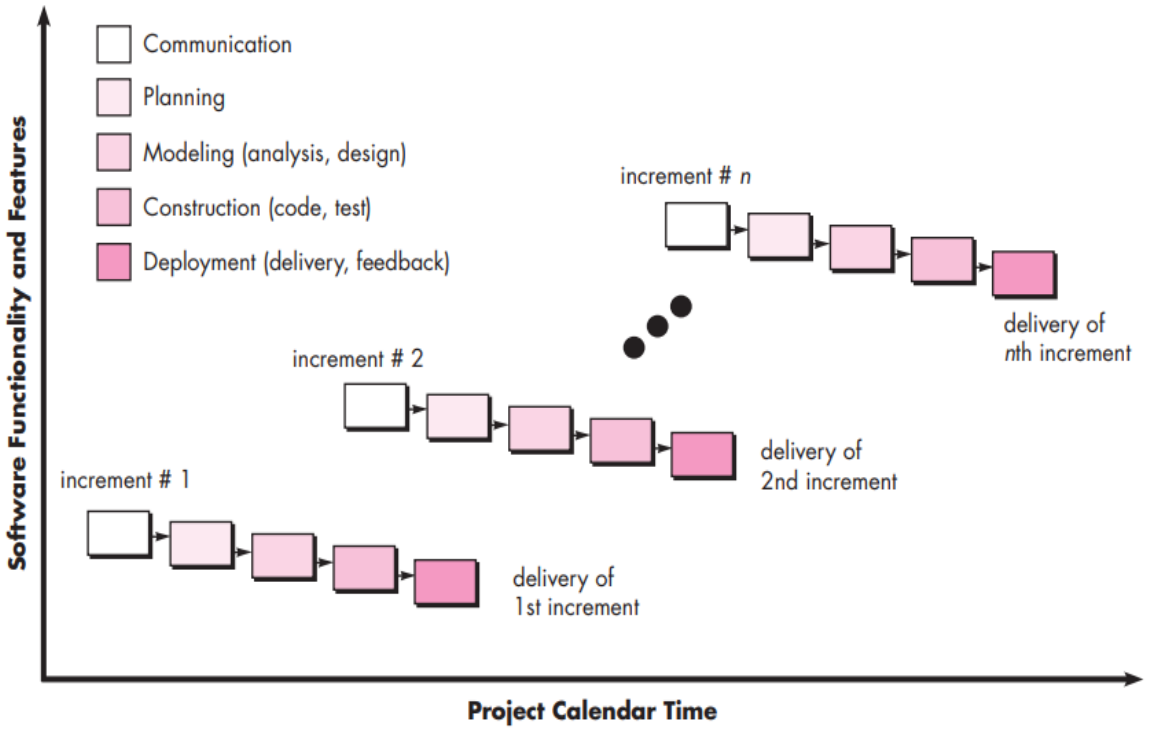


Figure: The Incremental Model

When an incremental model is used, the **first increment is often a core product**. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a plan is developed for the next increment.

The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.

Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks.

For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

PROTOTYPING MODEL

Prototyping: Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

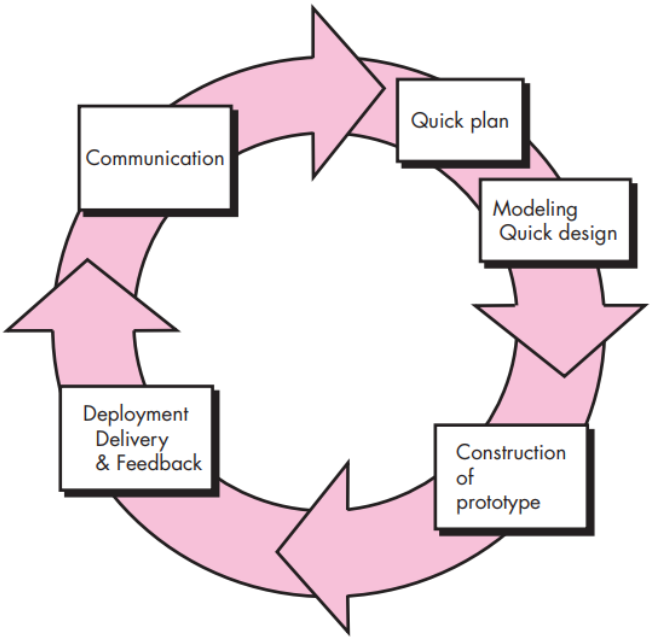


Figure: Prototyping

The prototyping paradigm begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).

The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

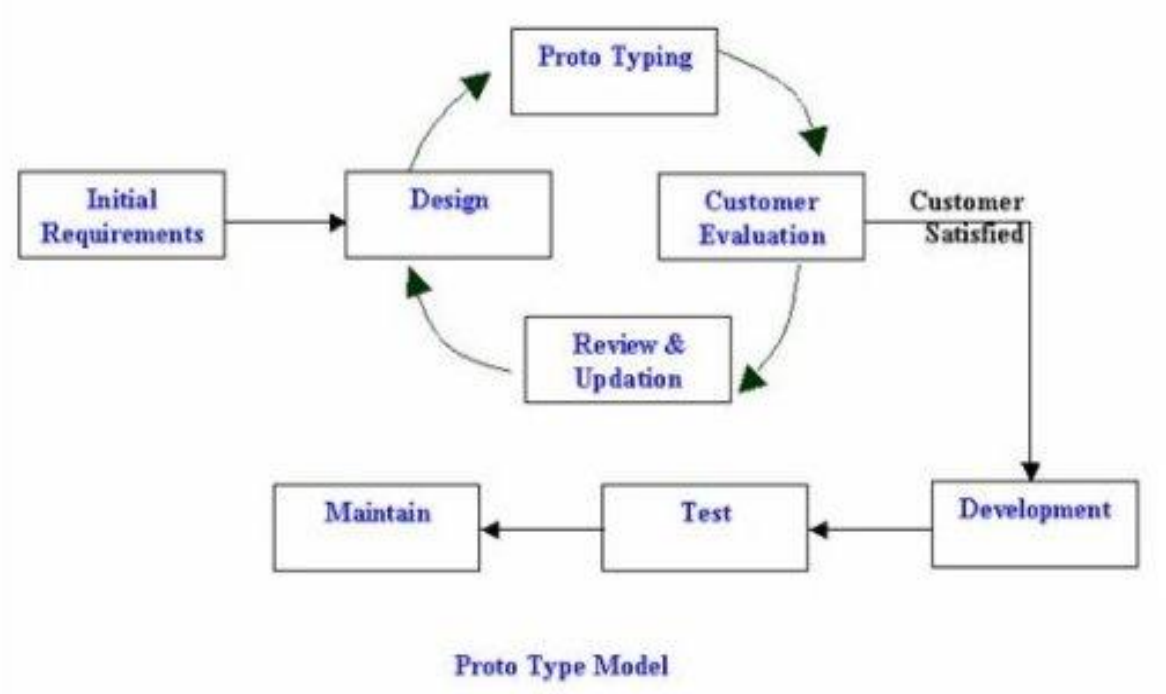


Figure: Prototype Model

Some problems may occur in Prototyping for the following reasons:

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.

2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Although these problems can occur, yet prototyping can be an effective method for software engineering.

THE SPIRAL MODEL

Originally proposed by Barry Boehm, the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.

It provides the potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner:

"The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions."

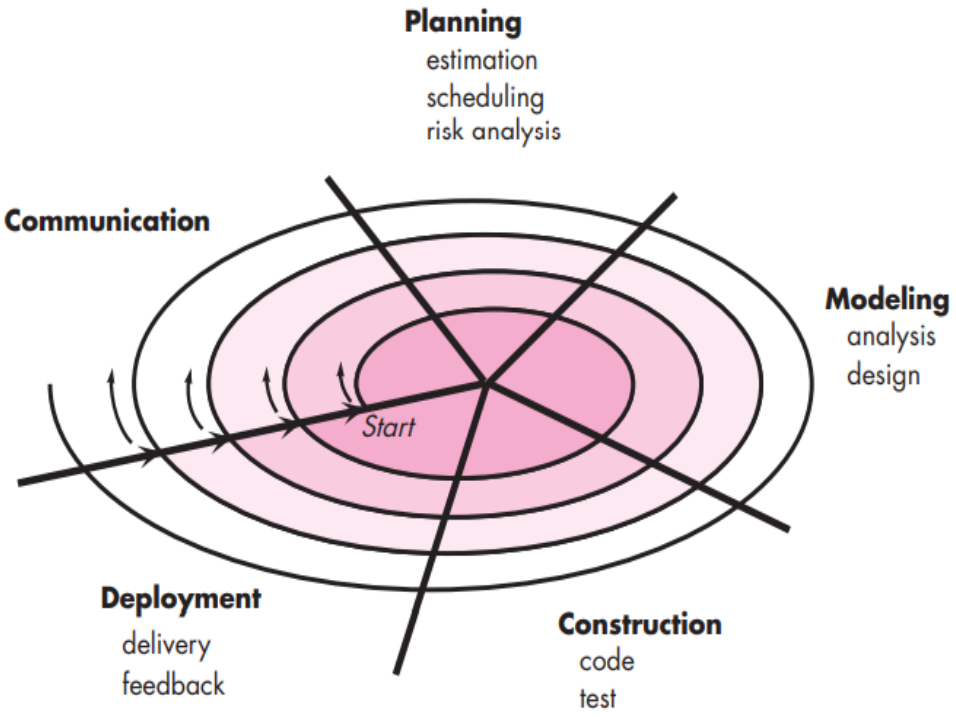


FIGURE: TYPICAL SPIRAL MODEL

[NOTE: The arrows pointing inward along the axis separating the deployment region from the communication region indicate a potential for local iteration along the same spiral path.]

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of **framework activities** defined by the software engineering team. Each of the framework activities represent one **segment** of the spiral path illustrated in Figure. The spiral model can be adapted to apply throughout the life of the computer software.

SEGMENTS:

SEGMENT 1 includes following activities: **Communication** (requirement gathering, customer evaluation and understanding)

SEGMENT 2 includes following activities: **Planning** (estimation, scheduling and risk analysis)

SEGMENT 3 includes following activities: **Modeling** (analysis and design)

SEGMENT 4 includes following activities: **Construction** (coding and testing)

SEGMENT 5 includes following activities: **Deployment** (delivery and feedback)

CIRCUITS AROUND THE SPIRAL:

As this evolutionary process begins, the software team performs activities that are implied by a **circuit** around the spiral in a clockwise direction, beginning at the center.

Risk is considered as each revolution is made.

Anchor point milestones, a combination of work products and conditions that are attained along the path of the spiral, are noted for each evolutionary **pass**.

The **first circuit** around the spiral might result in the development of a product specification and concept development of project, that starts at the core of the spiral and continues for multiple iterations until concept development is complete.

subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.

Each pass through the planning region results in adjustments to the project plan.

Cost and schedule are adjusted based on feedback derived from the customer after delivery.

In addition, the project manager adjusts the planned number of iterations required to complete the software.

The **version** or **build** or **deliverable** produced at the end of Deployment phase of **the last circuit**, is the **final software product.**

SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

A software requirements specification (SRS) is a detailed description of a software system to be developed with its functional and non-functional requirements.

The SRS is developed based the agreement between customer and contractors. It may include the use cases of how user is going to interact with software system. The software requirement specification document consistent of all necessary requirements required for project development.

To develop the software system we should have clear understanding of Software system. To achieve this we need to continuous communication with customers to gather all requirements.

A good SRS defines the how Software System will interact with all internal modules, hardware, communication with other programs and human user interactions with wide range of real life scenarios.

Using the Software requirements specification (SRS) document on QA lead, managers creates test plan. It is very important that testers must be cleared with every detail specified in this document in order to avoid faults in test cases and its expected results.

It is highly recommended to review or test SRS documents before start writing test cases and making any plan for testing.

Let's see how to test SRS and the important point to keep in mind while testing it.

1. Correctness of SRS should be checked. Since the whole testing phase is dependent on SRS, it is very important to check its correctness. There are some standards with which we can compare and verify.

2. Ambiguity should be avoided. Sometimes in SRS, some words have more than one meaning and this might confused testers making it difficult to get the exact reference. It is advisable to check for such ambiguous words and make the meaning clear for better understanding.

3. Requirements should be complete. When tester writes test cases, what exactly is required from the application, is the first thing which needs to be clear. For e.g. if application needs to send the specific data of some specific size then it should be clearly mentioned in SRS that how much data and what is the size limit to send.

4. Consistent requirements. The SRS should be consistent within itself and consistent to its reference documents. If you call an input "Start and Stop" in one place, don't call it "Start/Stop" in another. This sets the standard and should be followed throughout the testing phase.

5. Verification of expected result: SRS should not have statements like "Work as expected", it should be clearly stated that what is expected since different testers would have different thinking aspects and may draw different results from this statement.

6. Testing environment: some applications need specific conditions to test and also a particular environment for accurate result. SRS should have clear documentation on what type of environment is needed to set up.

7. Pre-conditions defined clearly: one of the most important part of test cases is pre-conditions. If they are not met properly then actual result will always be different expected result. Verify that in SRS, all the pre-conditions are mentioned clearly.

8. Requirements ID: these are the base of test case template. Based on requirement Ids, test case ids are written. Also, requirements ids make it easy to categorize modules so just by looking at them, tester will know which module to refer. SRS must have them such as id defines a particular module.

9. Security and Performance criteria: security is priority when a software is tested especially when it is built in such a way that it contains some crucial information when leaked can cause harm to business. Tester should check that all the security related requirements are properly defined and are clear to him. Also, when we talk about performance of a software, it plays a very important role in business so all the requirements related to performance must be clear to the tester and he must also know when and how much stress or load testing should be done to test the performance.

10. Assumption should be avoided: sometimes when requirement is not cleared to tester, he tends to make some assumptions related to it, which is not a right way to do testing as assumptions could go wrong and hence, test results may vary. It is better to avoid assumptions and ask clients about all the "missing requirements" to have a better understanding of expected results.

11. Deletion of irrelevant requirements: there are more than one team who work on SRS so it might be possible that some irrelevant requirements are included in SRS. Based on the understanding of the software, tester can find out which are these requirements and remove them to avoid confusions and reduce work load.

12. Freeze requirements: when an ambiguous or incomplete requirement is sent to client to analyze and tester gets a reply, that requirement result will be updated in the next SRS version and client will freeze that requirement. Freezing here means that result will not change again until and unless some major addition or modification is introduced in the software.

Most of the defects which we find during testing are because of either incomplete requirements or ambiguity in SRS. To avoid such defects it is very important to test software requirements specification before writing the test cases. Keep the latest version of SRS with you for reference and keep yourself updated with the latest change made to the SRS. Best practice is to go through the document very carefully and note down all the confusions, assumptions and incomplete requirements and then have a meeting with the client to get them clear before development phase starts as it becomes costly to fix the bugs after the software is developed. After all the requirements are cleared to a tester, it becomes easy for him to write effective test cases and accurate expected results.

FORMAL Requirement SPECIFICATION

A formal software specification is a statement expressed in a language whose vocabulary, syntax, and semantics are formally defined. The need for a formal semantic definition means that the specification languages cannot be based on natural language; it must be based on mathematics.

The advantages of a formal language are:

• The development of a formal specification provides insights and understanding of the software requirements and the software design.

• Given a formal system specification and a complete formal programming language definition, it may be possible to prove that a program conforms to its specifications.

• Formal specification may be automatically processed. Software tools can be built to assist with their development, understanding, and debugging.

• Depending on the formal specification language being used, it may be possible to animate a formal system specification to provide a prototype system.

• Formal specifications are mathematical entities and may be studied and analyzed using mathematical methods.

• Formal specifications may be used as a guide to the tester of a component in identifying appropriate test cases.

Relational and State-Oriented Notations

Relational notations are used based on the concept of entities and attributes.

Entities are elements in a system; the names are chosen to denote the nature of the elements (e.g., stacks, queues).

Attributes are specified by applying functions and relations to the named entities.

Attributes specify permitted operations on entities, relationships among entities, and data flow between entities.

Relational notations include implicit equations, recurrence relations, and algebraic axioms. State-oriented specifications use the current state of the system and the current stimuli presented to the system to show the next state of the system.

The execution history by which the current state was attained does not influence the next state; it is dependent only on the current state and the current stimuli.

State-oriented notations include decision tables, event tables, transition tables, and finite-state tables.

SPECIFICATION PRINCIPLES

Principle 1: Separate functionality from implementation. A specification is a statement of what is desired, not how it is to be realized. Specifications can take two general forms. The first form is that of mathematical functions: Given some set of inputs, produce a particular set of outputs. The general form of such specifications is find [a/the/all] result such that P(input), where P represents an arbitrary predicate. In such specifications, the result to be obtained has been entirely expressed in a "what", rather than a "how" form, mainly because the result is a mathematical function of the input (the operation has well-defined starting and stopping points) and is unaffected by any surrounding environment.

Principle 2: A process-oriented systems specification language is sometimes required. If the environment is dynamic and its changes affect the behavior of some entity interacting with that environment (as in an embedded computer system), its behavior cannot be expressed as a mathematical function of its input. Rather a process-oriented description must

be employed, in which the "what" specification is achieved by specifying a model of the desired behavior in terms of functional responses to various stimuli from the environment.

Principle 3: The specification must provide the implementer all of the information he/she needs to complete the program, and no more. In particular, no information about the structure of the calling program should be conveyed.

Principle 4: The specification should be sufficiently formal that it can conceivably be tested for consistency, correctness, and other desirable properties.

Principle 5: The specification should discuss the program in terms normally used by the user and implementer alike.

SOME SPECIFICATION TECHNIQUES

1. Implicit Equations

Specify computation of square root of a number between 0 and some maximum value Y to a tolerance E.

(0<=X<=Y){ABS_VALUE[(WHAT(X)) 2-X]}<=E

2. Recurrence Relation

Good for recursive computations.

Example, Fibonacci numbers 0, 1, 1, 2, 3, 5, 8,...

FI(0) = 0;

FI(1) = 1;

FI(n) = FI(n-1) + FI(n-2); for n>=1.

Verification and Validation

We have seen the "V-Model". In the V Model Software Development Life Cycle, based on requirement specification document the development & testing activity is started.

The V-model is also called as Verification and Validation model.

The testing activity is perform in the each phase of Software Testing Life Cycle.

In the first half of the model validations testing activity is integrated in each phase like review user requirements, System Design document & in the next half the Verification testing activity is come in picture.

Verification *(ARE WE BUILDING THE PRODUCT RIGHT?)*

***Definition** : The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.*

Verification is a static practice of verifying documents, design, code and program. It includes all the activities associated with producing high quality software: inspection, design analysis and specification analysis. It is a relatively objective process.

Verification will help to determine whether the software is of high quality, but it will not ensure that the system is useful. Verification is concerned with whether the system is well-engineered and error-free.

Methods of Verification : Static Testing

- *Walkthrough*
- *Inspection*
- *Review*

Validation *(ARE WE BUILDING THE RIGHT PRODUCT?)*

***Definition**: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.*

Validation is the process of evaluating the final product to check whether the software meets the customer expectations and requirements. It is a dynamic mechanism of validating and testing the actual product.

Methods of Validation : Dynamic Testing

- *Testing according to End Users*

Difference between Verification and Validation

The distinction between the two terms is largely to do with the role of specifications.

Verification *is the process of checking that the software meets the specification. "Did I build what I need?"*

Validation *is the process of checking whether the specification captures the customer's needs. "Did I build what I said I would?"*

Verification	***Validation***
Verification is a static practice of verifying documents, design, code and program.	Validation is a dynamic mechanism of validating and testing the actual product.
It does not involve executing the code.	It always involves executing the code.
It is human based checking of documents and files.	It is computer based execution of program.
Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	Validation uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc.
Verification is to check whether the software conforms to specifications.	Validation is to check whether software meets the customer expectations and requirements.
It can catch errors that validation cannot catch. It is low level exercise.	*It can catch errors that verification cannot catch. It is High Level Exercise.*

Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
Verification is done by QA team to ensure that the software is as per the specifications in the SRS document.	***Validation*** *is carried out with the involvement of testing team.*
It generally comes first-done before **validation**.	It generally follows after **verification**.
Are we building the system right?	Are we building the right system?
Verification is the process of evaluating products of a development phase to find out whether they meet the specified requirements.	Validation is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
The objective of Verification is to make sure that the product being develop is as per the requirements and design specifications.	The objective of Validation is to make sure that the product actually meet up the user's requirements, and check whether the specifications were correct in the first place.
Following activities are involved in Verification: Reviews, Meetings and Inspections.	Following activities are involved in Validation: Testing like black box testing, white box testing, gray box testing etc.
Verification is carried out by QA team to check whether implementation software is as per specification document or not.	Validation is carried out by testing team.
Execution of code is not comes under Verification.	Execution of code is comes under Validation.

Verification process explains whether the outputs are according to inputs or not.	Validation process describes whether the software is accepted by the user or not.
Verification is carried out before the Validation.	Validation activity is carried out just after the Verification.
Following items are evaluated during Verification: Plans, Requirement Specifications, Design Specifications, Code, Test Cases etc,	Following item is evaluated during Validation: Actual product or Software under test.
Cost of errors caught in Verification is less than errors found in Validation.	Cost of errors caught in Validation is more than errors found in Verification.
It is basically manually checking the of documents and files like requirement specifications etc.	It is basically checking of developed program based on the requirement specifications documents & files.

UNIT -2 SOFTWARE PROJECT MANAGEMENT

OBJECTIVES

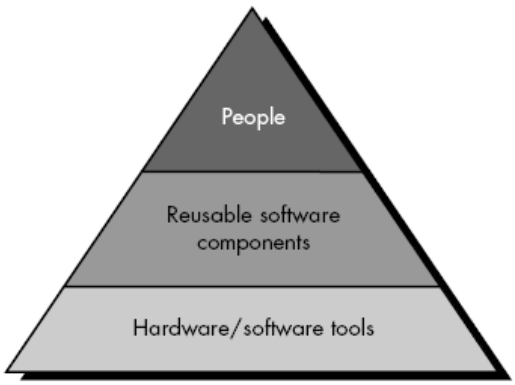
The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses. In addition, estimates should attempt to define best case and worst case scenarios so that project outcomes can be bounded. The planning objective is achieved through a process of information discovery that leads to reasonable estimates. In the following sections, each of the activities associated with software project planning is discussed.

RESOURCES

The second software planning task is estimation of the resources required to accomplish the software development effort. Figure illustrates development resources as a pyramid. The *development environment*—hardware and software tools—sits at the foundation of the resources pyramid and provides the infrastructure to support the development effort. At a higher level, we encounter reusable *software components*— software building blocks that can dramatically reduce development costs and accelerate delivery. At the top of the pyramid is the primary resource—*people.* Each resource is specified with four characteristics: description of the resource, a state- ment of availability, time when the resource will be required; duration of time that resource will be applied. The last two characteristics can be viewed as a time window.

Availability of the resource for a specified window must be established at the

earliest practical time.



HUMAN RESOURCES

The planner begins by evaluating scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client/server) are specified. For relatively small projects (one person-year or less), a single individual may perform all software engineering tasks, consulting with specialists as required.The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made.

REUSABLE SOFTWARE RESOURCES

Component-based software engineering (CBSE)5 emphasizes reusability—that is, the creation and reuse of software building blocks [HOO91]. Such building blocks, often called *components*, must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

Bennatan [BEN92] suggests four software resource categories that should be considered as planning proceeds:

Off-the-shelf components.

Existing software that can be acquired from a third party or that has been developed internally for a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.

Full-experience components.

Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full-experience components will be relatively low-risk.

Partial-experience components.

Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk.

New components.

Software components that must be built by the software team specifically for the needs of the current project.The following guidelines should be considered by the software planner when reusable components are specified as a resource.

DECOMPOSITION TECHNIQUES

Software Sizing

"Fuzzy logic" sizing. This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range. Although personal experience can be used, the planner should also have access to a historical database of projects so that estimates can be compared to actual experience.

Function point sizing.

Standard component sizing. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to determine the delivered size per standard component. To illustrate, consider an information systems application. The planner estimates that 18 reports will be generated. Historical data indicates that 967 lines of COBOL are required per report. This enables the planner to estimate that 17,000 LOC will be required for the reports component. Similar estimates and computation are made for other standard components, and a combined size value (adjusted statistically) results.

Change sizing. This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished. Using an "effort ratio"for each type of change, the size of the change may be estimated

Problem-Based Estimation

LOC-Based Estimation

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

FP BASED ESTIMATION:

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of inputs	20	24	30	24	4	97
Number of outputs	12	15	22	16	5	78
Number of inquiries	16	22	28	22	5	88
Number of files	4	4	5	4	10	42
Number of external interfaces	2	2	3	2	7	15
Count total						*320*

Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
On-line data entry	4
Input transaction over multiple screens	5
Master files updated on-line	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5
Complexity adjustment factor	1.17

Finally, the estimated number of FP is derived:

$$FP_{estimated} = \text{count-total} \times [0.65 + 0.01 \times \Sigma (F_i)]$$
$$FP_{estimated} = 375$$

Process-Based Estimation

Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function									
↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

PROJECT PLANNING OBJECTIVES

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses. In addition, estimates should attempt to define best case and worst case scenarios so that project outcomes can be bounded.

The planning objective is achieved through a process of information discovery that leads to reasonable estimates. In the following sections, each of the activities associated with software project planning is discussed.

THE COCOMO MODEL

In his classic book on "software engineering economics," Barry Boehm [BOE81] introduced a hierarchy of software estimation models bearing the name COCOMO, for *COnstructive COst MOdel.* The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called *COCOMO II* .Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

Application composition model. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.

Early design stage model. Used once requirements have been stabilized and basic software architecture has been established.

Post-architecture-stage model. Used during the construction of the software. Like all estimation models for software, the COCOMO II models require sizing information.Three different sizing options are available as part of the model hierarchy:

object points, function points, and lines of source code. The *object point* is an indirect software measure that is computed using counts of the number of

(1) screens (at the user interface)

(2) reports, and

(3) components likely to be required to build the application. Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e.,simple, medium, or difficult) using criteria suggested by Boehm [BOE96]. In essence,complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.

The Software Equation

The software equation Is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project. The model has been derived from productivity data collected for over 4000 contemporary software projects. Based on these data, an estimation model of the form

$E = [\text{LOC} \_ B0.333/P]3 \_ (1/t4)$ (5-3)

where E = effort in person-months or person-years

t = project duration in months or years

B = "special skills factor"16

P = "productivity parameter" that reflects:

• Overall process maturity and management practices

• The extent to which good software engineering practices are used

• The level of programming languages used

• The state of the software environment

• The skills and experience of the software team

• The complexity of the application

Typical values might be P = 2,000 for development of real-time embedded software; P = 10,000 for telecommunication and systems software; P = 28,000 for business systems

applications.17 The productivity parameter can be derived for local conditions using historical data collected from past development efforts. It is important to note that the software equation has two independent parameters:

(1) an estimate of size (in LOC) and (2) an indication of project duration in calendar months or years.

RISK ANALYSIS

First, risk concerns future happenings. Today and yesterday are beyond active concern, as we are already reaping what was previously sowed by our past actions. The question is, can we, therefore, by changing our actions today, create an opportunity for a different and hopefully better situation for ourselves tomorrow. This means second, that risk involves change, such as in changes of mind, opinion, actions, or places . . . [Third,] risk involves choice, and the uncertainty that choice itself entails.

What is it?

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is a potential problem—it might happen, it might not. But, regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur.

Who does it?

Everyone involved in the software process—managers, software engineers, and customers participate in risk analysis and management.

SOFTWARE RISKS

There is general agreement that risk always involves two characteristics

• *Uncertainty*—the risk may or may not happen; that is, there are no 100% probable risks.

• *Loss*—if the risk becomes a reality, unwanted consequences or losses will occur.

When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project.project complexity, size, and the degree of structural uncertainty were also defined as project (and estimation) risk factors.

Technical risks threaten the quality and timeliness of the software to be produced.If a technical risk becomes a reality, implementation may become difficult or impossible.Technical risks identify potential design, implementation, interface, verification,and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors.Technical risks occur because the problem is harder to solve than we thought it would be.

Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top five business risks are

(1) building a excellent product or system that no one really wants (market risk), (2)building a product that no longer fits into the overall business strategy for the company (strategic risk)

(3) building a product that the sales force doesn't understand

how to sell

(4) losing the support of senior management due to a change in focus or

a change in people (management risk)

(5) losing budgetary or personnel commitment (budget risks). It is extremely important to note that simple categorization won't always work. Some risks are simply unpredictable in advance.

Another general categorization of risks has been proposed by Charette [CHA89]. *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).

Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

RISK IDENTIFICATION

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories that have been presented earlier : generic risks and product-specific risks.

Generic risks are a potential threat to every software project.

Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand. To identify product-specific risks, the project plan and the software statement of scope are examined and

an answer to the following question is developed: "What special characteristics of this product may threaten our project plan?"

One method for identifying risks is to create a *risk item checklist.* The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

• *Product size*—risks associated with the overall size of the software to be built or modified.

• *Business impact*—risks associated with constraints imposed by management or the marketplace.

• *Customer characteristics*—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.

• *Process definition*—risks associated with the degree to which the software process has been defined and is followed by the development organization.

• *Development environment*—risks associated with the availability and quality of the tools to be used to build the product.

• *Technology to be built*—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.

• *Staff size and experience*—risks associated with the overall technical and project experience of the software engineers who will do the work.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answers to these questions allow the planner to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of “risk components and drivers" [AFC88] are listed along with their probability *Although generic risks are important to consider, usually the product-specific risks cause the most headaches. Be certain to spend the time to identify as many product-specific risks as possible.*

RISK COMPONENT & DRIVERS

The risk components are defined in the following manner:

• *Performance risk*—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.

• *Cost risk*—the degree of uncertainty that the project budget will be maintained.

• *Support risk*—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.

• *Schedule risk*—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

Components / Category		Performance	Support	Cost	Schedule
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of $500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of $100K to $500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of $1K to $100K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than $1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

RISK MITIGATION, MONITORING, AND MANAGEMENT

All of the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues:

• risk avoidance

• risk monitoring

• risk management and contingency planning

If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for *risk mitigation.* For example, assume that high staff turnover is noted as a project risk, *r1*. Based on past history and management intuition, the likelihood, *l1,* of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact, *x1,* is projected at level 2. That is, high turnover will have a critical impact on project cost and schedule.

To mitigate this risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are

• Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).

• Mitigate those causes that are under our control before the project starts.

• Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.

• Organize project teams so that information about each development activity is widely dispersed.

• Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.

• Conduct peer reviews of all work (so that more than one person is "up to speed”).

• Assign a backup staff member for every critical technologist.As the project proceeds, risk monitoring activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the following factors can be monitored:

• General attitude of team members based on project pressures.

• The degree to which the team has jelled.

• Interpersonal relationships among team members.

• Potential problems with compensation and benefits.

• The availability of jobs within the company and outside it.

In addition to monitoring these factors, the project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called for the definition of documentation standards and mechanisms to be sure that documents are developed in a timely manner. This is one mechanism for ensuring continuity, should a critical individual leave the project. The project manager should monitor documents carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is well underway and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, the project manager may temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to “get up to speed.” Those individuals who are leaving are asked to stop all work and spend their last weeks in “knowledge transfer mode.” This might

include video-based knowledge capture, the development of “commentary documents,” and/or meeting

with other team members who will remain on the project.

It is important to note that RMMM steps incur additional project cost. For example,spending the time to "backup" every critical technologist costs money. Part of risk management, therefore, is to evaluate when the benefits accrued by the RMMM steps are outweighed by the costs associated with implementing them. In essence, the project planner performs a classic cost/benefit analysis. If risk aversion steps for high turnover will increase both project cost and duration by an estimated 15 percent, but the predominant cost factor is "backup," management may decide not to implement this step. On the other hand, if the risk aversion steps are projected to increase costs by 5 percent and duration by only 3 percent management will likely put all into place.

For a large project, 30 or 40 risks may be identified. If between three and seven risk management steps are identified for each, risk management may become a project in itself! For this reason, we adapt the Pareto 80–20 rule to software risk. Experience indicates that 80 percent of the overall project risk (i.e., 80 percent of the potential for project failure) can be accounted for by only 20 percent of the identified risks. The work performed during earlier risk analysis steps will help the planner to determine which of the risks reside in that 20 percent (e.g., risks that lead to the highest risk exposure). For this reason, some of the risks identified, assessed, and projected may not make it into the RMMM plan—they don't fall into the critical 20 percent (the risks with highest project priority).

SOFTWARE PROJECT SCHEDULING

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. During early stages of project planning, a *macroscopic schedule* is developed. This type of schedule identifies all major software engineering activities and the product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined

into a *detailed schedule.* Here, specific software tasks (required to accomplish an activity) are identified and scheduled. Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end-date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end-date is set by the software engineering organization. Effort is distributed to make best use of resources and an end-date is defined after careful analysis of the software. Unfortunately, the first situation is encountered far more frequently than the second. Like all other areas of software engineering, a number of basic principles guide

software project scheduling:

Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed .

Interdependency. The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

Effort validation. Every project has a defined number of staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people have been scheduled at any given time. For example, consider a project that has three assigned staff members (e.g., 3 person-days are available per day of assigned effort5). On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person days of effort. More effort has been allocated than there are people to do the work.

Defined responsibilities. Every task that is scheduled should be assigned to a specific team member.

Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g. the design of a module) or a part of a work product. Work products are often combined in deliverables.

Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

Each of these principles is applied as the project schedule evolves.

SCHEDULING

Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification to software projects.

Program evaluation and review technique (PERT) and ***critical path method*** (CPM)

[MOD83] are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities:

• Estimates of effort

• A decomposition of the product function

• The selection of the appropriate process model and task set

• Decomposition of tasks

Interdependencies among tasks may be defined using a task network. Tasks, sometimes called the project *work breakdown structure* (WBS), are defined for the product as a whole or for individual functions.

Both PERT and CPM provide quantitative tools that allow the software planner to

(1) determine the *critical path*—the chain of tasks that determines the duration of the project;

(2) establish “most likely” time estimates for individual tasks by applying statistical models;

(3) calculate “boundary times” that define a time "window" for a particular task.

Boundary time calculations can be very useful in software project scheduling. Slippage in the design of one function, for example, can retard further development of other functions.

Riggs describes important boundary times that may be discerned from a PERT or CPM network:

(1) the earliest time that a task can begin when all preceding tasks are completed in the shortest possible time,

(2) the latest time for task initiation before the minimum project completion time is delayed,

(3) the earliest finish—the sum of the earliest start and the task duration,

(4) the latest finish the latest start time added to task duration, and

(5) the *total float*—the amount of surplus time or leeway allowed in scheduling tasks so that the network critical path is maintained on schedule. Boundary time calculations lead to a determination of critical path and provide the manager with a quantitative method for evaluating progress as tasks are completed.

UNIT -3 REQUIREMENT ANALYSIS

REQUIREMENT ANALYSIS-

Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design . Requirements engineering activities result in the specification of software's operational characteristics (function, data, and behavior), indicate software's interface with other system elements, and establish constraints that software must meet. Requirements analysis allows the software engineer (sometimes called analyst in this role) to refine the software allocation and build models of the data, functional, and behavioral domains that will be treated by software.

Requirements analysis provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs. Finally, the requirements specification provides the developer and the customer with the means to assess quality once software is built.

Software requirements analysis may be divided into five areas of effort:

(1) problem recognition,

(2) evaluation and synthesis,

(3) modeling,

(4) specification

(5) review.

Initially, the analyst studies the System Specification (if one exists) and the Software Project Plan. It is important to understand software in a system context and to review the software scope that was used to generate planning estimates. Next, communication for analysis must

be established so that problem recognition is ensured. The goal is recognition of the basic problem elements as perceived by the customer/users.

Problem evaluation and solution synthesis is the next major area of effort for analysis.The analyst must define all externally observable data objects, evaluate the flow and content of information, define and elaborate all software functions, understand software behavior in the context of events that affect the system, establish system interface characteristics, and uncover additional design constraints. Each of these tasks serves to describe the problem so that an overall approach or solution may be synthesized.

ANALYSIS PRINCIPLES-

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

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

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

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

represented.

4. The models that depict information, function, and behavior must be partitioned

in a manner that uncovers detail in a layered (or hierarchical) fashion.

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

By applying these principles, the analyst approaches a problem systematically. The information domain is examined so that function may be understood more completely.

Models are used so that the characteristics of function and behavior can be

communicated in a compact fashion. Partitioning is applied to reduce complexity.

Essential and implementation views of the software are necessary to accommodate the logical constraints imposed by processing requirements and the physical constraints imposed by other system elements.

Davis suggests a set of guiding principles for requirements engineering:

• ***Understand the problem before you begin to create the analysis model.*** There is a tendency to rush to a solution, even before the problem is understood. This often leads to elegant software that solves the wrong problem!

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

Since the perception of the quality of software is often based on the perception of the "friendliness" of the interface, prototyping (and the iteration that results) are highly recommended.

• ***Record the origin of and the reason for every requirement.***

This is the first step in establishing traceability back to the customer.

• ***Use multiple views of requirements***.

Building data, functional, and behavioral models provide the software engineer with three different views. This reduces the likelihood that something will be missed and increases the likelihood that inconsistency will be recognized.

• ***Rank requirements.***

Tight deadlines may preclude the implementation of every software requirement. If an incremental process model is applied, those requirements to be delivered in the first increment must be identified.

• ***Work to eliminate ambiguity.***

Because most requirements are described in a natural language, the opportunity for ambiguity abounds. The use of formal technical reviews is one way to uncover and eliminate ambiguity. A software engineer who takes these principles to heart is more likely to develop a software specification that will provide an excellent foundation for design.

The Information Domain

All software applications can be collectively called *data processing.* Interestingly, this term contains a key to our understanding of software requirements. Software is built to process data, to transform data from one form to another; that is, to accept input, manipulate it in some way, and produce output. This fundamental statement of objective is true whether we build batch software for a payroll system or real-time embedded software to control fuel flow to an automobile engine.

The first operational analysis principle requires an examination of the information

domain and the creation of a *data model.* The information domain contains three different views of the data and control as each is processed by a computer program:

(1) information content and relationships (the data model)

(2) information flow, and

(3) information structure.

To fully understand the information domain, each of these

views should be considered.

Information content represents the individual data and control objects that constitute some larger collection of information transformed by the software. For example, the data object, **paycheck,** is a composite of a number of important pieces of data: the payee's name, the net amount to be paid, the gross pay, deductions, and so forth. Therefore, the content of **paycheck** is defined by the attributes that are needed to create it. Similarly, the content of a control object called **system status** might be defined by a string of bits. Each bit represents a

separate item of information that indicates whether or not a particular device is on- or off-line.

.***Information flow*** represents the manner in which data and control change as each moves through a system. Referring to Figure 11.3, input objects are transformed to intermediate information (data and/or control), which is further transformed to output. Along this transformation path (or paths), additional information may be introduced from an existing data store (e.g., a disk file or memory buffer). The transformations applied to the data are functions or subfunctions that a program must perform. Data and control that move between two transformations (functions) define the interface for each function.

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

Modelling

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

The second and third operational analysis principles require that we build models

of function and behavior.

Functional models. Software transforms information, and in order to

accomplish this, it must perform at least three generic functions: input, processing, and output. When functional models of an application are created,

the software engineer focuses on problem specific functions. The functional

model begins with a single context level model (i.e., the name of the software

to be built). Over a series of iterations, more and more functional detail is

provided, until a thorough delineation of all system functionality is represented.

Behavioral models.

Most software responds to events from the outside world. This stimulus/response characteristic forms the basis of the behavioral model. A computer program always exists in some state—an externally observable mode of behavior (e.g., waiting, computing, printing, polling) that is changed only when some event occurs. For example, software will remain in the wait state until (1) an internal clock indicates that some time interval has passed, (2) an external event (e.g., a mouse movement) causes an interrupt, or (3) an external system signals the software to act in some manner. A behavioral model creates a representation of the states of the software and the events that cause a software to change state.

Partitioning

Problems are often too large and complex to be understood as a whole. For this reason, we tend to partition (divide) such problems into parts that can be easily understood and establish interfaces between the parts so that overall function can be accomplished. The fourth operational analysis principle suggests that the information, functional, and behavioral domains of software can be partitioned.

In essence, *partitioning* decomposes a problem into its constituent parts.

Conceptually, we establish a hierarchical representation of function or information and then partition the uppermost element by

(1) exposing increasing detail by moving vertically in the hierarchy or

(2) functionally decomposing the problem by moving horizontally in the hierarchy

Essential and Implementation Views

An ***essential view*** of software requirements presents the functions to be accomplished and information to be processed without regard to implementation details. For example, the essential view of the *SafeHome* function *read sensor status* does not concern itself with the physical form of the data or the type of sensor that is used. In fact, it could be argued that read status would be a more appropriate name for this function, since it disregards details about the input mechanism altogether.

Similarly, an essential data model of the data item **phone number** (implied by the function *dial phone number*) can be represented at this stage without regard to the underlying data structure (if any) used to implement the data item. By focusing attention on the essence of the problem at early stages of requirements engineering, we leave our options open to specify implementation details during later stages of requirements specification and software design.

The ***implementation view*** of software requirements presents the real world manifestation of processing functions and information structures. In some cases, a physical representation is developed as the first step in software design. However most computer-based systems are specified in a manner that dictates accommodation of certain implementation details. A *SafeHome* input device is a perimeter sensor (not a watch dog, a human guard, or a booby trap). The sensor detects illegal entry by sensing a break in an electronic circuit. The general characteristics of the sensor should be noted as part of a software requirements specification. The analyst must recognize the constraints imposed by predefined system elements (the sensor) and consider the implementation view of function and information when such a view is appropriate.

SOFTWARE PROTOTYPING

Analysis should be conducted regardless of the software engineering paradigm that is applied. However, the form that analysis takes will vary. In some cases it is possible to apply operational analysis principles and derive a model of software from which a design can be developed. In other situations, requirements elicitation (via FAST, QFD, use-cases, or other

"brainstorming" techniques [JOR89]) is conducted, analysis principles are applied, and a model of the software to be built, called a *prototype*, is constructed for customer and developer assessment. Finally, some circumstances require the construction of a prototype at the beginning of analysis, since the model is the only means through which requirements can be effectively derived. The model then evolves into production software.

Selecting the Prototyping Approach

The prototyping paradigm can be either close-ended or open-ended. The close-ended approach is often called ***throwaway prototyping***. Using this approach, a prototype serves solely as a rough demonstration of requirements. It is then discarded, and the software is engineered using a different paradigm. An open-ended approach, called ***evolutionary prototyping***, uses the prototype as the first part of an analysis activity that will be continued into design and construction. The prototype of the software is the first evolution of the finished system.

Before a close-ended or open-ended approach can be chosen, it is necessary to

determine whether the system to be built is amenable to prototyping. A number of prototyping candidacy factors [BOA84] can be defined: application area, application complexity, customer characteristics, and project characteristics

Because the customer must interact with the prototype in later steps, it is essential that

(1) customer resources be committed to the evaluation and refinement of the

prototype

(2) the customer is capable of making requirements decisions in a

timely fashion. Finally, the nature of the development project will have a strong bearing on the efficacy of prototyping. Is project management willing and able to work with the prototyping method? Are prototyping tools available?

Prototyping Methods and Tools

For software prototyping to be effective, a prototype must be developed rapidly so that the customer may assess results and recommend changes. To conduct rapid prototyping, three generic classes of methods and tools (e.g., [AND92], [TAN89]) are available:

Fourth generation techniques.

Fourth generation techniques (4GT) encompass a broad array of database query and reporting languages, program and application generators, and other very high-level nonprocedural languages. Because 4GT enable the software engineer to generate executable code quickly, they are ideal for rapid prototyping.

Reusable software components.

Another approach to rapid prototyping is to assemble, rather than build, the prototype by using a set of existing software components. Melding prototyping and program component reuse will work only if a library system is developed so that components that do exist can be cataloged and then retrieved. It should be noted that an existing software product can be used as a prototype for a "new, improved" competitive product. In a way, this is a form of reusability for software prototyping.

Formal specification and prototyping environments.

Over the past two decades, a number of formal specification languages and tools have been developed as a replacement for natural language specification techniques .Today, developers of these formal languages are in the process of developing interactive environments that (1) enable an analyst to interactively create language-based specifications of a system or software, (2) invoke automated tools that translate the language-based specifications into executable code, and (3) enable the customer to use the prototype executable code to refine formal requirements.

SOFTWARE REQUIREMENT SPECIFICATION

The *Software Requirements Specification* is produced at the culmination of the analysis task. The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.

The National Bureau of Standards, IEEE (Standard No. 830-1984), and the U.S. Department of Defense have all proposed candidate formats for software requirements specifications (as well as other software engineering

documentation).

The ***Introduction*** of the software requirements specification states the goals and

objectives of the software, describing it in the context of the computer-based system.Actually, the Introduction may be nothing more than the software scope of the planning document.

The ***Information Description*** provides a detailed description of the problem that the software must solve. Information content, flow, and structure are documented. Hardware, software, and human interfaces are described for external system elements and internal software functions.

A description of each function required to solve the problem is presented in the

Functional Description. A processing narrative is provided for each function, design constraints are stated and justified, performance characteristics are stated, and one or more diagrams are included to graphically represent the overall structure of the software and interplay among software functions and other system elements.

The ***Behavioral Description*** section of the specification examines the operation of the software as a consequence of external events and internally generated control characteristics.

Validation Criteria is probably the most important and, ironically, the most often

neglected section of the *Software Requirements Specification.* How do we recognize a successful implementation? What classes of tests must be conducted to validate function, performance, and constraints? We neglect this section because completing it demands a thorough understanding of software requirements—something that we often do not have at this stage. Yet, specification of validation criteria acts as an implicit review of all other requirements. It is essential that time and attention be given to this section.

Finally, the specification includes a ***Bibliography and Appendix***. The bibliography contains references to all documents that relate to the software. These include other software engineering documentation, technical references, vendor literature, and standards. The appendix contains information that supplements the specifications. Tabular data, detailed description of algorithms, charts, In many cases the *Software Requirements Specification* may be accompanied by an executable prototype (which in some cases may replace the specification), a paper prototype or a *Preliminary User's Manual.*

The ***Preliminary User's Manual*** presents the software as a black box. That is, heavy emphasis is placed on user input and the resultant output. The manual can serve as a valuable tool for uncovering problems at the human/machine interface.

A review of the *Software Requirements Specification* (and/or prototype) is conducted by both the software developer and the customer. Because the specification forms the foundation of the development phase, extreme care should be taken in conducting the review.

The review is first conducted at a macroscopic level; that is, reviewers attempt to

ensure that the specification is complete, consistent, and accurate when the overall information, functional, and behavioral domains are considered. However, to fully explore each of these domains, the review becomes more detailed, examining not only broad descriptions but the way in which requirements are worded. For example, when specifications contain "vague terms" (e.g., *some, sometimes, often, usually, ordinarily, most,* or *mostly*), the reviewer should flag the statements for further clarification.

Once the review is complete, the *Software Requirements Specification* is "signedoff" by both the customer and the developer. The specification becomes a "contract" for software development. Requests for changes in requirements after the specification is finalized will not be eliminated. But the customer should note that each after the fact change is an extension of software scope and therefore can increase cost and/or protract the schedule.

REPRESENTATION

We have already seen that software requirements may be specified in a variety of ways. However, if requirements are committed to paper or an electronic presentation medium (and they almost always should be!) a simple set of guidelines is well worth following:

Representation format and content should be relevant to the problem.

A general outline for the contents of a *Software Requirements Specification* can be developed. However, the representation forms contained within the specification are likely to vary with the application area. For example, a specification for a manufacturing automation system might use different symbology, diagrams and language than the specification for a programming language compiler.

Information contained within the specification should be nested.

Representations should reveal layers of information so that a reader can move to the level of detail required. Paragraph and diagram numbering schemes should indicate the level of detail that is being presented. It is sometimes

worthwhile to present the same information at different levels of abstraction

to aid in understanding.

Diagrams and other notational forms should be restricted in number

and consistent in use.

Confusing or inconsistent notation, whether graphical

or symbolic, degrades understanding and fosters errors.

Representations should be revisable.

The content of a specification will change. Ideally, CASE tools should be available to update all representations that are affected by each change.

Investigators have conducted numerous studies (e.g., [HOL95], [CUR85]) on human factors associated with specification. There appears to be little doubt that symbology and arrangement affect understanding. However, software engineers appear to have individual preferences for specific symbolic and diagrammatic forms. Familiarity often lies at the root of a person's preference, but other more tangible factors such as spatial arrangement, easily recognizable patterns, and degree of formality often dictate an individual's choice.

SPECIFICATION PRINCIPLES

Specification, regardless of the mode through which we accomplish it, may be viewed as a representation process. Requirements are represented in a manner that ultimately leads to successful software implementation. A number of specification principles,

adapted from the work of Balzer and Goodman [BAL86], can be proposed:

1. Separate functionality from implementation.

2. Develop a model of the desired behavior of a system that encompasses data and the functional responses of a system to various stimuli from the environment.

3. Establish the context in which software operates by specifying the manner in which other system components interact with software.

4. Define the environment in which the system operates and indicate how "a highly intertwined collection of agents react to stimuli in the environment(changes to objects) produced by those agents" .

5. Create a cognitive model rather than a design or implementation model. The cognitive model describes a system as perceived by its user community.

6. Recognize that "the specifications must be tolerant of incompleteness and augmentable." A specification is always a model—an abstraction—of some real (or envisioned) situation that is normally quite complex. Hence, it will be incomplete and will exist at many levels of detail.

7. Establish the content and structure of a specification in a way that will enable it to be amenable to change. This list of basic specification principles provides a basis for representing software requirements. However, principles must be translated into realization. In the next section we examine a set of guidelines for creating a specification of requirements.

THE ELEMENTS OF ANALYSIS MODEL

The analysis model must achieve three primary objectives:

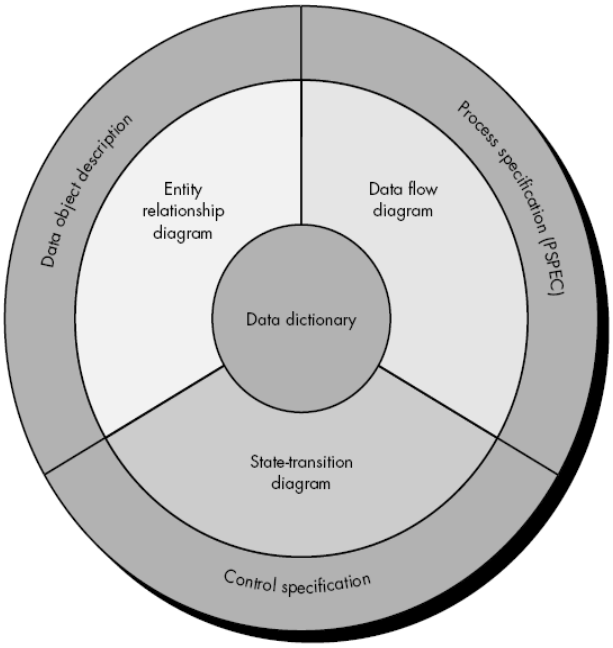
(1) to describe what the customer requires

(2) to establish a basis for the creation of a software design, and

(3) to define a set of requirements that can be validated once the software is built. To accomplish these objectives, the analysis model derived during structured analysis takes the form illustrated in Figure 1.1.

At the core of the model lies the ***data dictionary***—a repository that contains descriptions of all data objects consumed or produced by the software. Three different diagrams surround the the core.

The ***entity relation diagram*** (ERD) depicts relationships between data objects. The ERD is the notation that is used to conduct the data modeling activity. The attributes of each data object noted in the ERD can be described using a data object description.



2.1
re of
is

The ***data flow diagram*** (DFD) serves two purposes: (1) to provide an indication of how data are transformed as they move through the system and (2) to depict the functions (and subfunctions) that transform the data flow. The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function. A description of each function presented in the DFD is contained in a *process specification* (PSPEC).

The ***state transition diagram*** (STD) indicates how the system behaves as a consequence of external events. To accomplish this, the STD represents the various modes of behavior (called *states*) of the system and the manner in which transitions are made from state to state. The STD serves as the basis for behavioral modeling. Additional information about the control aspects of the software is contained in the *control specification* (CSPEC).

DATA MODELLING

Data modeling answers a set of specific questions that are relevant to any data processing application. What are the primary data objects to be processed by the system?

What is the composition of each data object and what attributes describe the

object? Where do the the objects currently reside? What are the relationships between each object and other objects? What are the relationships between the objects and the processes that transform them?

Data Objects, Attributes, and Relationships

The data model consists of three interrelated pieces of information: the data object, the attributes that describe the data object, and the relationships that connect data objects to one another.

Data objects

A *data object* is a representation of almost any composite information that must be understood by software. By *composite information*, we mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

Data objects (represented in bold) are related to one another. For example, **person** can *own* **car**, where the relationship *own* connotes a specific "connection" between **person** and **car**. The relationships are always defined by the context of the problem that is being analyzed.

A data object encapsulates data only—there is no reference within a data object

to operations that act on the data.

Attributes

Attributes define the properties of a data object and take on one of three

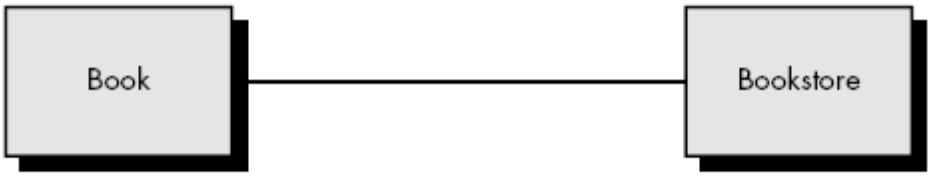
different characteristics. They can be used to

(1) name an instance of the data object,

(2) describe the instance, or

(3) make reference to another instance in another table.

In addition, one or more of the attributes must be defined as an *identifier*—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement



(a) A basic connection between objects

(b) Relationships between objects

Relationships

Data objects are connected to one another in different ways. Consider

two data objects, **book** and **bookstore**. These objects can be represented using

the simple notation illustrated in Figure 2.a. A connection is established between **book** and **bookstore** because the two objects are related. But what are the relationships?

To determine the answer, we must understand the role of books and bookstores

within the context of the software to be built. We can define a set of

object/relationship pairs that define the relevant relationships. For example,

• A bookstore orders books.

• A bookstore displays books.

• A bookstore stocks books.

• A bookstore sells books.

• A bookstore returns books.

Cardinality and Modality

The elements of data modeling—data objects, attributes, and relationships— provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood. We have defined a set of objects and represented the object/relationship pairs that bind them. But a simple pair that states: **object X** *relates* to **object Y** does not provide enough information for software engineering purposes. We must understand how many occurrences of **object X** are related to how many occurrences of **object Y.** This leads to a data modeling concept called *cardinality.*

Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object]. Cardinality is usually expressed as simply 'one' or 'many.' For example, a husband can have only one wife (in most cultures), while a parent can have many children. Taking into consideration all combinations of 'one' and 'many,' two [objects] can be related as

• One-to-one (l:l)—An occurrence of [object] 'A' can relate to one and only one occurrence of [object] 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'

• One-to-many (l:N)—One occurrence of [object] 'A' can relate to one or many occurrences of [object] 'B,' but an occurrence of 'B' can relate to only one occurrence of 'A.'

For example, a mother can have many children, but a child can have only one mother.

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

For example, an uncle can have many nephews, while a nephew can have many uncles.

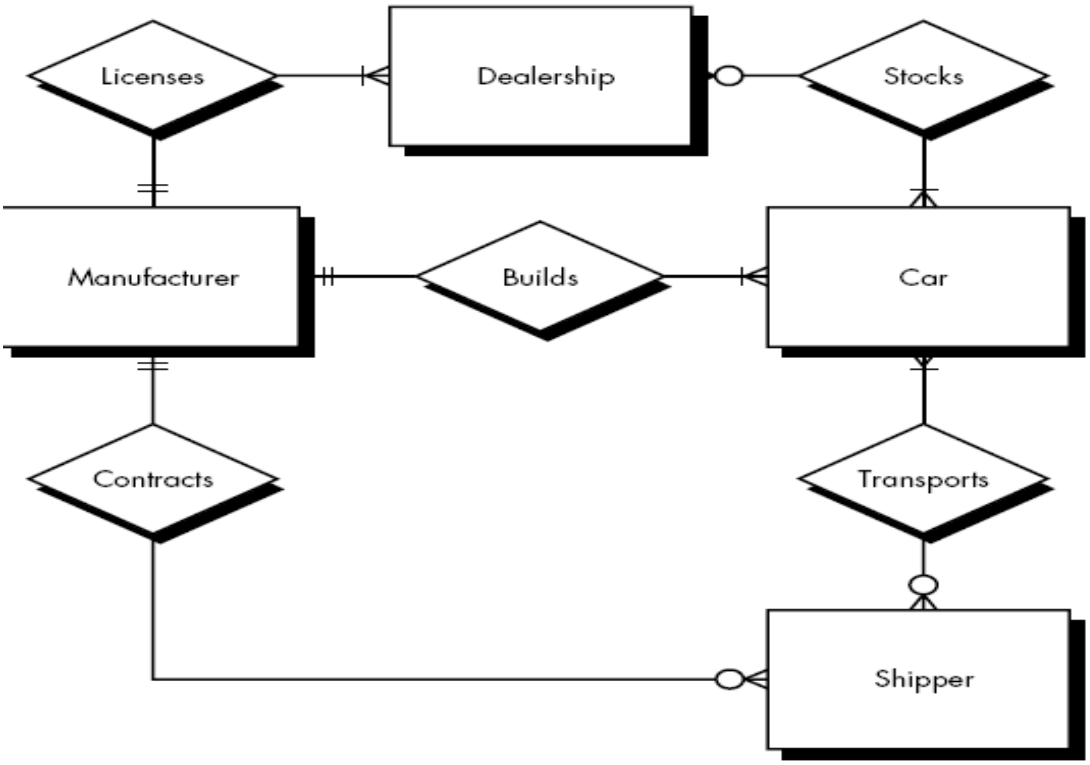
Cardinality defines "the maximum number of objects that can participate in a relationship" It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair.

Modality

The *modality* of a relationship is 0 if there is no explicit need for the relationship

to occur or the relationship is optional. The modality is 1 if an occurrence of

the relationship is mandatory. To illustrate, consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required. Figure 12.5 illustrates the relationship, cardinality, and modality between the data objects **customer** and **repair action.**



Entity/Relationship Diagrams

The object/relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the *entity/relationship diagram*. The ERD was originally proposed by Peter Chen for the design of relational database systems and has been extended by others.

A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and

relationships are established using a variety of special symbols that indicate cardinality and modality

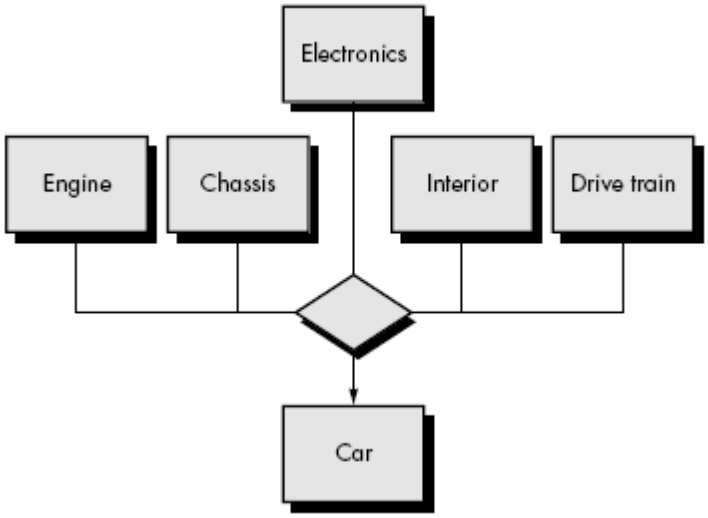


Fig 4

The relationship between the data objects **car** and **manufacturer** would be represented as shown in Figure 3. One manufacturer builds one or many cars. Given the context implied by the ERD, the specification of the data object **car.**

ERD notation also provides a mechanism that represents the associativity between objects. In the figure, each of the data objects that model the individual subsystems is associated with the data object **car.**

An *associative data object* is represented as shown in Figure 4.

Data Flow Diagrams

As information moves through software, it is modified by a series of transformations. A *data flow diagram* is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output. The basic form of a data flow diagram, also known as a *data flow graph* or a *bubble chart.* I

The data flow diagram may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. Therefore, the DFD provides a mechanism for functional modeling as well as information flow modeling. In so doing, it satisfies the second operational analysis principle (i.e., creating a functional model) .

A level 0 DFD, also called a *fundamental system model* or a *context model,* represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively.

Additional processes (bubbles) and information flow paths are represented as the level 0 DFD is partitioned to reveal more detail. For example, a level 1 DFD might contain five or six bubbles with interconnecting arrows. Each of the processes represented at level 1 is a subfunction of the overall system depicted in the context model.

BEHAVIORAL MODELLING

Behavioral modeling is an operational principle for all requirements analysis methods.Yet, only extended versions of structured analysis provide a

notation for this type of modeling. The state transition diagram represents the behavior of a system by depicting its states and the events that cause the system to change state. In addition, the STD indicates what actions (e.g., process activation) are taken as a consequence of a particular event.

A state is any observable mode of behavior. For example, states for a monitoring

and control system for pressure vessels described might be *monitoring*

state, alarm state, pressure release state, and so on. Each of these states represents a mode of behavior of the system. A state transition diagram indicates how the system moves from state to state.

Control flows are shown entering and exiting individual processes and the vertical bar representing the control specification(CSPEC) "window." For example, the **paper feed status** and **start/stop** events flow into the CSPEC bar. This implies that each of these events will cause some process represented in the CFD to be activated. If we were to examine the CSPEC internals, the **start/stop** event would be shown to activate/deactivate the *manage copying* process. Similarly, the **jammed** event (part of **paper feed status**) would activate *perform problem diagnosis.* It should be noted that all vertical bars within the CFD refer to the same CSPEC. An event flow can be input directly into

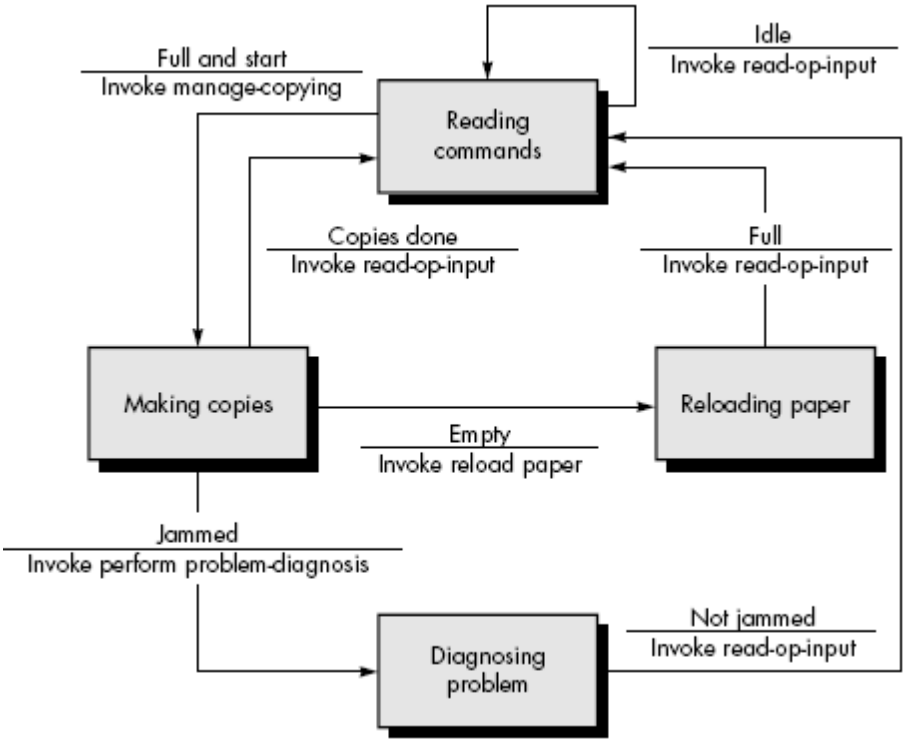
a process as shown with **repro fault.** However, this flow does not activate the process but rather provides control information for the process algorithm.

The Hatley and Pirbhai extensions to basic structured analysis notation focus

less on the creation of additional graphical symbols and more on the representation and specification of the control-oriented aspects of the software. The dashed arrow is once again

used to represent control or event flow. Unlike Ward and Mellor, Hatley and Pirbhai suggest that dashed and solid notation be represented separately. Therefore, a *control flow diagram* is defined. The CFD contains the same processes as the DFD, but shows control flow, rather than data flow.

Instead of representing control processes directly within the flow model, a notational reference (a solid bar) to a *control specification* (CSPEC) is used. In essence, the solid bar can be viewed as a "window" into an "executive" (the CSPEC) that controls the processes (functions) represented in the DFD based on the event that is passed through the window. A process specification is used to describe the inner workings of a process represented in a flow diagram.



A simplified state transition diagram for the photocopier software is shown in Figure 5. The rectangles represent system states and the arrows represent transitions between states. Each

arrow is labeled with a ruled expression. The top value indicates the event(s) that cause the transition to occur. The bottom value indicates the action that occurs as a consequence of the event. Therefore, when the paper tray is **full** and the **start** button is pressed, the system moves from the *reading commands* state to the *making copies* state. Note that states do not necessarily correspond to processes on a one-to-one basis. For example, the state *making copies* would encompass both the *manage copying* and *produce user displays* processes.

Creating an Entity/Relationship Diagram

The entity/relationship diagram enables a software engineer to fully specify the data objects that are input and output from a system, the attributes that define the properties of these objects, and their relationships. Like most elements of the analysis model, the ERD is constructed in an iterative manner. The following approach is taken:

1. During requirements elicitation, customers are asked to list the "things" that the application or business process addresses. These "things" evolve into a list of input and output data objects as well as external entities that produce or consume information.

2. Taking the objects one at a time, the analyst and customer define whether or not a connection (unnamed at this stage) exists between the data object and other objects.

3. Wherever a connection exists, the analyst and the customer create one or

more object/relationship pairs.

4. For each object/relationship pair, cardinality and modality are explored.

5. Steps 2 through 4 are continued iteratively until all object/relationships have

been defined. It is common to discover omissions as this process continues.

New objects and relationships will invariably be added as the number of iterations grows.

6. The attributes of each entity are defined.

7. An entity relationship diagram is formalized and reviewed.

Creating a Data Flow Model

The data flow diagram enables the software engineer to develop models of the information domain and functional domain at the same time. As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition of the system, thereby accomplishing the fourth operational analysis principle for function.

At the same time, the DFD refinement results in a corresponding refinement of

data as it moves through the processes that embody the application.

A few simple guidelines can aid immeasurably during derivation of a data flow

diagram:

(1) the level 0 data flow diagram should depict the software/system as a

single bubble.

(2) primary input and output should be carefully noted.

(3) refinement should begin by isolating candidate processes, data objects, and stores to be represented at the next level.

(4) all arrows and bubbles should be labeled with meaningful names.

(5) information flow continuity must be maintained from level to level, and

(6) one bubble at a time should be refined. There is a natural tendency to overcomplicate the data flow diagram.

This occurs when the analyst attempts to show too much detail too early or represents procedural aspects of the software in lieu of information flow.

THE DATA DICTIONARY

The analysis model encompasses representations of data objects, function, and control. In each representation data objects and/or control items play a role. Therefore, it is necessary to provide an organized approach for representing the characteristics of each data object and control item. This is accomplished with the data dictionary.

The data dictionary has been proposed as a quasi-formal grammar for describing

the content of objects defined during structured analysis. This important modeling

notation has been defined in the following manner :

“ The *data dictionary* is an organized listing of all data elements that are pertinent to the system, with precise, rigorous definitions so that both user and system analyst will have a common understanding of inputs, outputs, components of stores and [even] intermediate calculations “.

Today, the data dictionary is always implemented as part of a CASE "structured analysis and design tool." Although the format of dictionaries varies from tool to tool, most contain the following information:

• *Name*—the primary name of the data or control item, the data store or an

external entity.

• *Alias*—other names used for the first entry.

• *Where-used/how-used*—a listing of the processes that use the data or control

item and how it is used (e.g., input to the process, output from the process,

as a store, as an external entity.

• *Content description*—a notation for representing content.

• *Supplementary information*—other information about data types, preset values

(if known), restrictions or limitations, and so forth.

Once a data object or control item name and its aliases are entered into the data

dictionary, consistency in naming can be enforced. That is, if an analysis team member decides to name a newly derived data item **xyz,** but **xyz** is already in the dictionary, the CASE tool supporting the dictionary posts a warning to indicate duplicate names. This improves the consistency of the analysis model and helps to reduce errors.

“Where-used/how-used” information is recorded automatically from the flow models. When a dictionary entry is created, the CASE tool scans DFDs and CFDs to determine which processes use the data or control information and how it is used. Although this may appear unimportant, it is actually one of the most important benefits of the dictionary. During analysis there is an almost continuous stream of changes. For large projects, it is often quite difficult to determine the impact of a change. Many a software engineer has asked, "Where is this data object used? What else will have to change if we modify it? What will the overall impact of the change be?" Because the data dictionary can be treated as a database, the analyst can ask "where used/how used" questions, and get answers to these queries.

The notation used to develop a content description is noted in the following table:

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

The notation enables a software engineer to represent composite data in one of the three fundamental ways that it can be constructed:

1. As a sequence of data items.

2. As a selection from among a set of data items.

3. As a repeated grouping of data items. Each data item entry that is represented

as part of a sequence, selection, or repetition may itself be another composite data item that needs further refinement within the dictionary.

The data dictionary provides us with a precise definition of **telephone number** for the DFD in question. In addition it indicates where and how this data item is used and any supplementary information that is relevant to it.

The data dictionary entry begins as follows:

name: telephone number

aliases: none

where used/how used: assess against set-up (output)

dial phone (input)

description:

telephone number = [local number|long distance number]

local number = prefix + access number

long distance number = 1 + area code + local number

area code = [800 | 888 | 561]

prefix = *a three digit number that never starts with 0 or 1*

access number = * any four number string *

The content description is expanded until all composite data items have been represented as elementary items (items that require no further expansion) or until all composite items are represented in terms that would be well-known and unambiguous to all readers. It is also important to note that a specification of elementary data often restricts a system. For example, the definition of area code indicates that only three area codes (two toll-free and one in South Florida) are valid for this system.

The data dictionary defines information items unambiguously. Although we might

assume that the telephone number represented by the DFD in Figure 12.22 could accommodate a 25-digit long distance carrier access number, the data dictionary content description tells us that such numbers are not part of the data that may be used.

UNIT -4 SOFTWARE DESIGN

DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the past four decades. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied.

Each helps the software engineer to answer the following questions:

• What criteria can be used to partition software into individual components?

• How is function or data structure detail separated from a conceptual representation of the software?

• What uniform criteria define the technical quality of a software design?

Abstraction

Each step in the software process is a refinement in the level of abstraction of the

software solution. During system engineering, software is allocated as an element of a computer-based system. During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

A *data abstraction* is a named collection of data that describes a data object . In the context of the procedural abstraction *open*, we can define a data abstraction

called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door**.

Control abstraction is the third form of abstraction used in software design. Like

procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the *synchronization semaphore* used to coordinate activities in an operating system.

Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

The process of program refinement proposed by Wirth is analogous to the process of refinement and partitioning that is used during requirements analysis. The difference is in the level of implementation detail that is considered, not the approach. Refinement is actually a process of *elaboration.*We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Modularity

The concept of modularity in computer software has been espoused for almost five decades. Software architecture (described in Section 13.4.4) embodies modularity; that is, software is divided into separately named and addressable components, often called *modules*, that are integrated to satisfy problem requirements. Another important question arises when modularity is considered. How do wedefine an appropriate module of a given size? The answer lies in the method(s) used to define modules within a system. Meyer defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

Modular decomposability. If a design method provides a systematic mechanism for decomposing the problem into subproblems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.

Modular composability. If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

Modular understandability. If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change.

Modular continuity. If small changes to the system requirements result in

changes to individual modules, rather than systemwide changes, the impact of change-induced side effects will be minimized.

Modular protection. If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

Software Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". In its simplest form, architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components. In a broader sense, however, *components* can be generalized to represent major system elements and their interactions. One goal of software design is to derive an architectural rendering of a system.This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enable a software engineer to reuse design level concepts.

Control Hierarchy

Control hierarchy, also called *program structure*, represents the organization of program components (modules) and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions, or repetition of operations; nor is it necessarily applicable to all architectural styles.

The control relationship among modules is expressed in the following way: A module that controls another module is said to be *superordinate* to it, and conversely, a module controlled by another is said to be *subordinate* to the controller .

For example, referring to Figure module *M* is superordinate to modules *a, b,* and *c.* Module *h* is subordinate to module *e* and is ultimately subordinate to module *M.* Width-oriented relationships (e.g., between modules *d* and *e*) although possible to express in practice, need not be defined with explicit terminology.

The control hierarchy also represents two subtly different characteristics of the

software architecture: visibility and connectivity. *Visibility* indicates the set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. For example, a module in an object-oriented system may have access to a wide array of data objects that it has inherited, but makes use of only a small number of these data objects. All of the objects are visible to the module.

Connectivity indicates the set of components that are directly invoked or used as

data by a given component. For example, a module that directly causes another module to begin execution is connected to it

Structural Partitioning

If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically. Referring to Figure 13.4a, *horizontal partitioning* defines separate branches of the modular hierarchy for each major program function. *Control modules,* represented in a darker shade are used to coordinate communication between and execution of the functions. The simplest approach to horizontal partitioning defines three partitions—input, data transformation (often called *processing*) and output. Partitioning the architecture horizontally provides a number of distinct benefits:

• software that is easier to test

• software that is easier to maintain

• propagation of fewer side effects

• software that is easier to extend

Because major functions are decoupled from one another, change tends to be less complex and extensions to the system (a common occurrence) tend to be easier to accomplish without side effects. On the negative side, horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow (if processing requires rapid movement from one function to another).

Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure to the

representation of software architecture.

Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information. However, it is important to understand the classic methods available for organizing information and the concepts that underlie information hierarchies.

Information Hiding

The concept of modularity leads every software designer to a fundamental question:

"How do we decompose a software solution to obtain the best set of modules?"

The principle of *information hiding* suggests that modules be

"characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module

EFFECTIVE MODULAR DESIGN

All the fundamental design concepts described in the preceding section serve to precipitate modular designs. In fact, modularity has become an accepted approach in all engineering disciplines. A modular design reduces complexity facilitates change (a critical aspect of software maintainability), and results in easier implementation by encouraging parallel development of different parts of a system.

Functional Independence

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific subfunction of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules

are possible.To summarize, functional independence is a key to good design, and design is the key to software quality.

Cohesion

Cohesion is a natural extension of the information hiding concept described earlier. A cohesive module performs a single task within a software procedure,

requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

Cohesion may be represented as a "spectrum." We always strive for high cohesion, although the mid-range of the spectrum is often acceptable. The scale for cohesion is nonlinear. That is, low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion. In practice, a designer need not be concerned with categorizing cohesion in a specific module.

Rather, the overall concept should be understood and low levels of cohesion should be avoided when modules are designed. At the low (undesirable) end of the spectrum, we encounter a module that performs a set of tasks that relate to each other loosely, if at all. Such modules are termed *coincidentally cohesive.* A module that performs tasks that are related logically (e.g.a module that produces all output regardless of type) is *logically cohesive.* When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits *temporal cohesion.*

Coupling

Coupling is a measure of interconnection among modules in a software structure.

Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

In software design, we strive for lowest possible coupling. Simple connectivity

among modules results in software that is easier to understand and less prone to a "ripple effect" , caused when errors occur at one location and propagate

through a system.

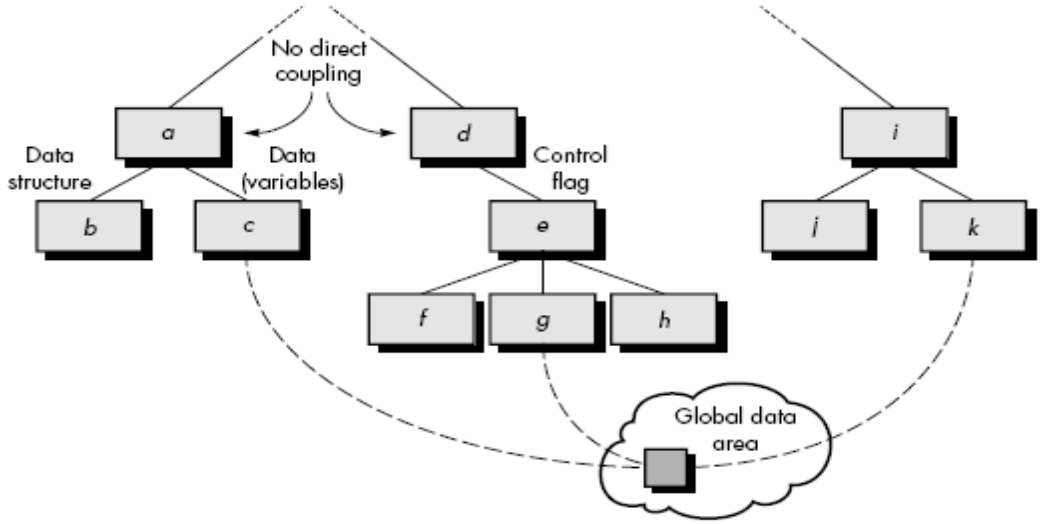


Figure provides examples of different types of module coupling. Modules *a*

and *d* are subordinate to different modules. Each is unrelated and therefore no direct coupling occurs. Module *c* is subordinate to module *a* and is accessed via a conventional argument list, through which data are passed. As long as a simple argument list is present (i.e., simple data are passed; a one-to-one correspondence of items exists), low coupling (called *data coupling*) is exhibited in this portion of structure. A variation of data coupling, called *stamp coupling,* is found when a portion of a data structure (rather than simple arguments) is passed via a module interface. This occurs between modules *b* and *a*.

“ The designer's goal is to produce a model or representation of an entity that will later be built “

Design is a meaningful engineering representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design. In the

software engineering context, design focuses on four major areas of concern: data, architecture, interfaces, and components.

Software engineers design computerbased systems, but the skills required at each level of design work are different. At the data and architectural level, design focuses on patterns as they apply to the application to be built. At the interface

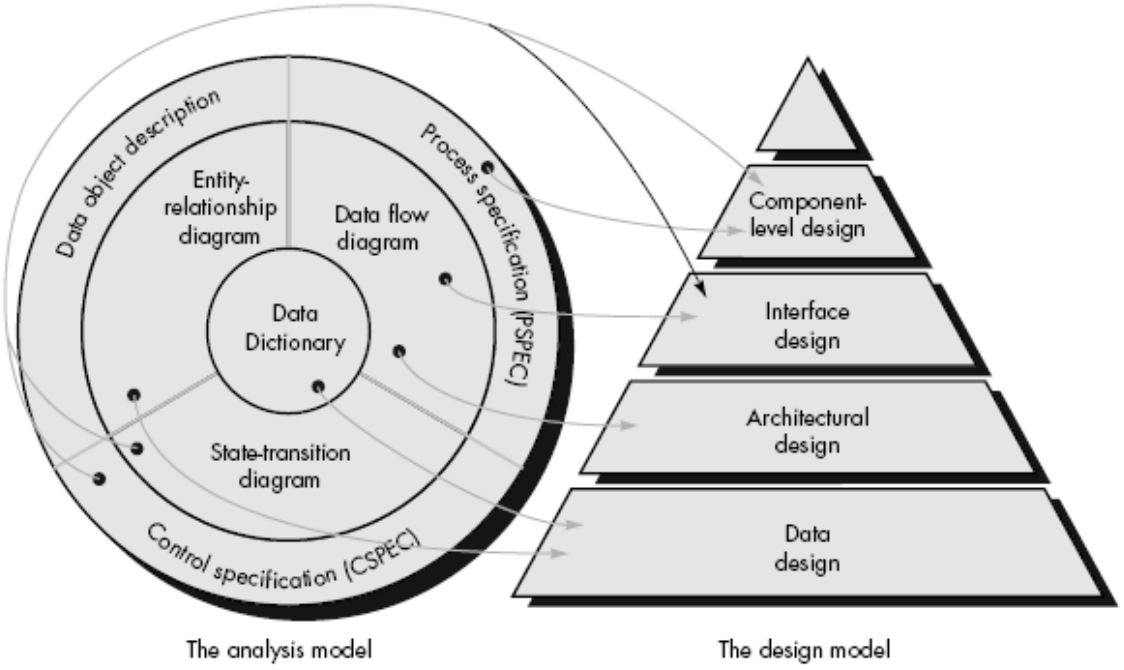
level, human ergonomics often dictate our design approach. At the component level, a "programming approach" leads us to effective data and procedural designs.

SOFTWARE DESIGN AND SOFTWARE ENGINEERING

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities—design, code generation, and test—that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software.

The *data design* transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity. Part of data design may occur in conjunction with the design of software architecture.More detailed data design occurs as each software component is designed.

The *architectural design* defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied [SHA96]. The architectural design representation the framework of a computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model.



THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

Design and Software Quality

Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs discussed in earler chapters.

McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design:

• The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.

• The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

• The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?

In order to evaluate the quality of a design representation, we must establish technical criteria for good design. Later in this chapter, we discuss design quality criteria in some detail. For the time being, we present the following guidelines:

A design should exhibit an architectural structure that

(1) has been created using recognizable design patterns

(2) is composed of components that exhibit good design characteristics

(3) can be implemented in an evolutionary fashion, thereby facilitating

implementation and testing.

A design should be modular; that is, the software should be logically partitioned

into elements that perform specific functions and subfunctions.

A design should contain distinct representations of data, architecture, interfaces,

and components (modules).

A design should lead to data structures that are appropriate for the objects to

be implemented and are drawn from recognizable data patterns.

A design should lead to components that exhibit independent functional

characteristics.

A design should lead to interfaces that reduce the complexity of connections

between modules and with the external environment.

A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

These criteria are not achieved by chance. The software design process encourages good design through the application of fundamental design principles, systematic methodology, and thorough review.

DESIGN PRINCIPLES

Software design is both a process and a model. The design *process* is a sequence of steps that enable the designer to describe all aspects of the software to be built. It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes "good" software, and an overall commitment to quality are critical success factors for a competent design. The design *model* is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software

provides a variety of different views of the computer software.

Basic design principles enable the software engineer to navigate the design process. Davis suggests a set of principles for software design, which have been

adapted and extended in the following list:

• **The design process should not suffer from "tunnel vision."** A good

designer should consider alternative approaches, judging each based on the

requirements of the problem, the resources available to do the job, and the

design concepts .

• **The design should be traceable to the analysis model.** Because a single

element of the design model often traces to multiple requirements, it is necessary

to have a means for tracking how requirements have been satisfied by

the design model.

• **The design should not reinvent the wheel.** Systems are constructed using

a set of design patterns, many of which have likely been encountered before.

These patterns should always be chosen as an alternative to reinvention.

Time is short and resources are limited! Design time should be invested in

representing truly new ideas and integrating those patterns that already exist.

• **The design should "minimize the intellectual distance"**

between the software and the problem as it exists in the real world.

That is, the structure of the software design should (whenever possible)

mimic the structure of the problem domain.

• **The design should exhibit uniformity and integration.** A design is uniform

if it appears that one person developed the entire thing. Rules of style

and format should be defined for a design team before design work begins. A

design is integrated if care is taken in defining interfaces between design

components.

• **The design should be structured to accommodate change.** The design

concepts discussed in the next section enable a design to achieve this principle.

• **The design should be structured to degrade gently, even when aberrant**

data, events, or operating conditions are encountered. Welldesigned

software should never "bomb." It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

• **Design is not coding, coding is not design.** Even when detailed procedural

designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.

• **The design should be assessed for quality as it is being created, not after the fact.** A variety of design concepts and design measures are available to assist the designer in assessing quality.

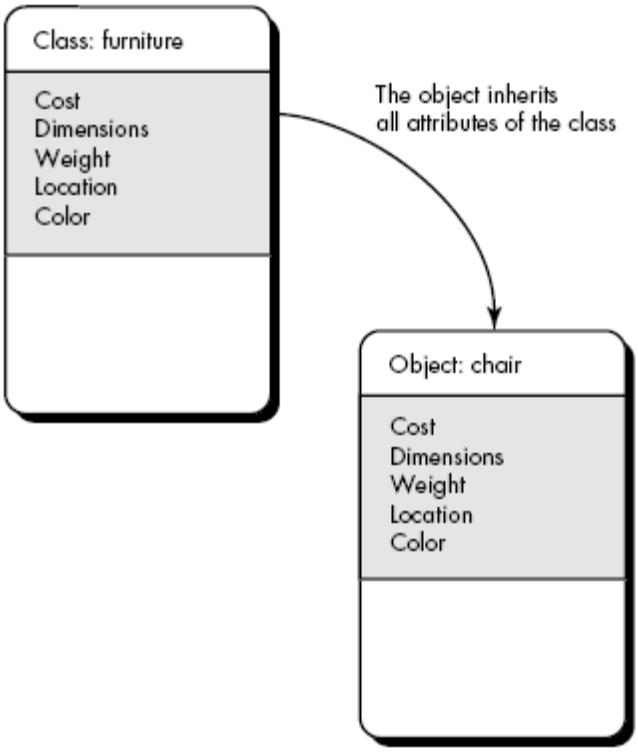
• **The design should be reviewed to minimize conceptual (semantic) errors.** There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, inconsistency) have been addressed before worrying about the syntax of the design model.

When these design principles are properly applied, the software engineer creates a design that exhibits both external and internal quality factors [MEY88]. *External quality factors* are those properties of the software that can be readily observed by users (e.g., speed,reliability, correctness, usability). *Internal quality factors* are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts.

UNIT -5 OBJECT –ORIENTED ANALYSIS

Object Oriented Analysis and Design

Software is primarily used to represent real-life players and processes inside a computer. In the past, software was considered to be a collection of information and procedures to transform that information from input to the output format. There was no explicit relationship between the information and the processes that operate on that information. The mapping between software components and their corresponding real-life objects and processes was hidden in the implementation details. There was no mechanism for sharing information and procedures among the objects that had similar properties.



To understand the object-oriented point of view, consider an example of a real world object—the thing you are sitting in right now—a chair. **Chair** is a member (the term *instance*

is also used) of a much larger class of objects that we call **furniture**. A set of generic attributes can be associated with every object in the class **furniture**. For example, all furniture has a cost, dimensions, weight, location, and color, among many possible *attributes.* These apply whether we are talking about a table or a chair, a sofa or an armoire. Because **chair** is a member of **furniture, chair** *inherits* all attributes defined for the class. Once the class has been defined, the attributes can be reused when new instances of the class are created. For example, assume that we were to define a new object called a **chable** (a cross between a chair and a table) that is a member of the class **furniture. Chable** inherits all of the attributes of **furniture**.

Classes and Objects

The fundamental concepts that lead to high-quality design apply equally to systems developed using conventional and object-oriented methods. For this reason, an OO model of computer software must exhibit data and procedural abstractions that lead to effective modularity. A class is an OO concept that encapsulates the data and procedural abstractions required to describe the content and behavior of some real world entity.

The data abstractions (attributes) that describe the class are enclosed by a "wall" of procedural abstractions (called *operations, methods,* or *services*) that are capable of manipulating the data in some way. The only way to reach the attributes (and operate on them) is to go through one of the methods that form the wall. Therefore, the class encapsulates data (inside the wall) and the processing that manipulates the data (the methods that make up the wall). This achieves information hiding and reduces the impact of side effects associated with change. Since the methods tend to manipulate a limited number of attributes, they are cohesive; and because communication occurs only through the methods that make up the "wall," the class tends to be decoupled from other elements of a system. All of these design characteristics lead to highquality software.

Encapsulation, Inheritance, and Polymorphism

As we have already noted, the OO class and the objects spawned from the class encapsulate data and the operations that work on the data in a single package. This provides a number of important benefits:

• The internal implementation details of data and procedures are hidden from the outside world (information hiding). This reduces the propagation of side effects when changes occur.

• Data structures and the operations that manipulate them are merged in a single named entity—the class. This facilitates component reuse.

• Interfaces among encapsulated objects are simplified. An object that sends a message need not be concerned with the details of internal data structures.

Hence, interfacing is simplified and the system coupling tends to be reduced. Inheritance is one of the key differentiators between conventional and OO systems. A subclass **Y** inherits all of the attributes and operations associated with its superclass, **X**. This means that all data structures and algorithms originally designed and implemented for **X** are immediately available for **Y**—no further work need be done. Reuse has been accomplished directly.

Object Oriented Design – Why?

There was a need for technology that could bridge the gap between the real-life objects and their counter-parts in the computer. Object oriented technology evolved to bridge the gap. Object-oriented technology helps in software modeling of real-life objects in a direct and explicit fashion, by encapsulating data and processes related to a real-life object or process in a single software entity. It also provides a mechanism through which the objects can inherit properties from their ancestors, just like real-life objects.

A complex system that works is invariably found to have evolved from a simple system that worked. The structure of a system also plays a very important role. It is likely that we understand only those systems that have hierarchical structure and where intracomponent linkages are generally stronger than inter-component linkages. This leads to loose coupling, high cohesion and ultimately more maintainability, which are the basic design considerations. Instead of being a collection of loosely bound data structures and functions, an object-oriented software system consists of objects that are, generally, hierarchical, highly cohesive, and loosely coupled. Some of the key advantages that make the object-oriented technology significantly attractive than other technologies include:

• Clarity and understandability of the system, as object-oriented approach is closer to the working of human cognition.

• Reusability of code resulting from low inter-dependence among objects, and provision of generalization and specialization through inheritance.

• Reduced effort in maintenance and enhancement, resulting from inheritance, encapsulation, low coupling, and high cohesion.

Object Oriented Design Components - What?

2.1 The Object and the Class

The basic unit of object-oriented design is an object. An object can be defined as a tangible entity that exhibits some well-defined behavior. An object represents an individual,

identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain. An object has a state, behavior, and identity. The state of an object encompasses all of the properties of the object and their current values. A property is an inherent or distinctive characteristic. Properties are usually static. All properties have some value. The state of an object is encapsulated within the object.

Behavior is how an object acts and reacts in terms of its state changes and message passing. The behavior of an object is completely defined by its actions. A message is some action that one object performs upon another in order to elicit a reaction. The operations that clients may perform upon an object are called methods. The structure and behavior of similar objects are defined in their common class. A class represents an abstraction - the essence or the template of an object, specifies an interface (the outside view - the public part) and defines an implementation (the inside view - the private part). The interface primarily consists of the declaration of all the operations applicable to instances of this class. The implementation of a class primarily consists of the implementation of all the operations defined in the interface of the class.

2.2 Classification

The most important and critical stage in OOA and OOD is the appropriate classification of objects into groups and classes. Proper classification requires looking at the problem from different angles and with an open mind. When looked at from different perspectives and analyzed with different sets of characteristics, the same object can be classified into different categories.

2.3 The Object Model

The elements of object-oriented design are collectively called the Object Model. The

object model encompasses the principles of abstraction, encapsulation, and hierarchy

or inheritance.

Abstraction is an extremely powerful technique for dealing with complexity. Unable to master the entirety of a complex object, we ignore its essential details, dealing instead with generalized, and the idealized model of the object. An abstraction focuses on the outside view of an object and hence serves to separate an object's external behavior from its implementation. Deciding upon the right set of abstractions for a given domain is the central problem in object-oriented design.

2.3.1 Relationship Among Objects

The object model presents a static view of the system and illustrates how different objects collaborate with one another through patterns of interaction. Inheritance, association and aggregation are the three inter-object relationships specified by the object model. Inheritance defines a "kind of" hierarchy among classes. By inheritance, we specify generalization/specialization relationship among objects. In this relationship, a class (called the subclass) shares the structure and behavior defined in another class (called the super class). A subclass augments or redefines the existing structure and behavior of its super class. By classifying objects into groups of related abstractions, we come to explicitly distinguish the common and distinct properties of different objects, which further helps us to master their inherent complexity. Identifying the hierarchy within a complex system requires the discovery of patterns among many objects.

Object Oriented Analysis

The intent of OOA is to define all classes, their relationships, and their behavior.

A number of tasks must occur:

1) Static Model

a) Identify classes (i.e. attributes and methods are defined)

b) Specify class hierarchy

c) Identify object-to-object relationships

d) Model the object behavior

2) Dynamic Model

a) Scenario Diagrams

Object Oriented Design

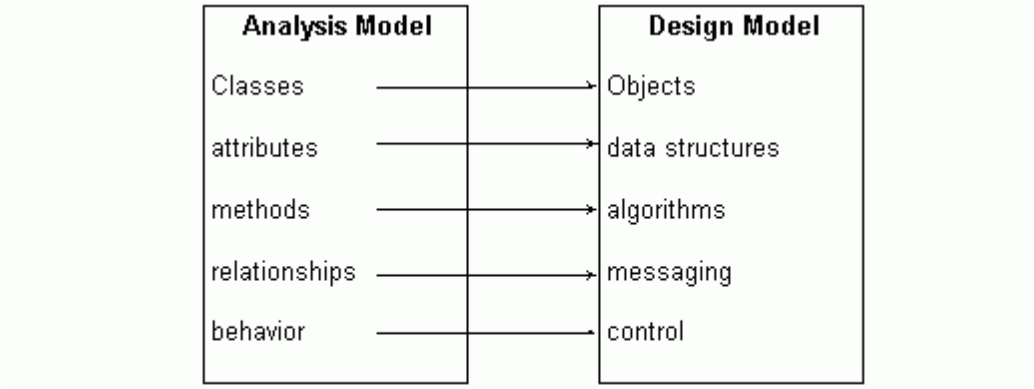
OOD transforms the analysis model into design model that serves as a blueprint for software construction. OOD results in a design that achieves a number of different levels of modularity. The four layers of the OO design pyramid are:

1) **The subsystem layer**: Contains a representation of each of the subsystems that enable the software to achieve its customer-defined requirements and to implement the technical infrastructure that supports customer requirements.

2) **The class and object layer**: Contains the class hierarchies that enable the system to be created using generalization and increasingly more targeted specializations. The layer also contains design representations for each object.

3) **The message layer**: Contains the details that enable each object to communicate with its collaborators. This layer establishes the external and internal interfaces for the system.

4) **The responsibility layer**: Contains the data structures and algorithmic design for all attributes and operations for each object.



Translating the analysis model into a design model during object design

OOA—object-oriented analysis — is based upon concepts that we first learned in kindergarten:

objects and attributes, classes and members, wholes and parts. Why it has taken us so long to apply these concepts to the analysis and specification of information systems is anyone's guess.

OOA is grounded in a set of basic principles that were introduced in earlier chapters.

In order to build an analysis model, five basic principles were applied:

(1) the information domain is modeled.

(2) function is described.

(3) behavior is represented.

(4) data, functional, and behavioral models are partitioned to expose greater detail.

(5) early models represent the essence of the problem while later models provide implementation details. These principles form the foundation for the approach to OOA presented in this chapter.

A Unified Approach to OOA

Over the past decade, Grady Booch, James Rumbaugh, and Ivar Jacobson have collaborated to combine the best features of their individual object-oriented analysis and design methods into a unified method. The result, called the *Unified Modeling Language* (UML), has become widely used throughout the industry.

UML allows a software engineer to express an analysis model using a modeling notation that is governed by a set of syntactic, semantic, and pragmatic rules. Eriksson and Penker [ERI98] explain these rules in the following way:

The syntax tells us how the symbols should look and how the symbols are combined. The syntax is compared to words in natural language; it is important to know how to spell them correctly and how to put different words together to form a sentence. The semantic rules tell us what each symbol means and how it should be interpreted by itself and in the context of other symbols; they are compared to the meanings of words in a natural language.

The pragmatic rules define the intentions of the symbols through which the purpose of the model is achieved and becomes understandable for others. This corresponds in natural language to the rules for constructing sentences that are clear and understandable. In UML, a system is represented using five different "views" that describe the system from distinctly different perspectives. Each view is defined by a set of diagrams.

The following views are present in UML:

User model view. This view represents the system (product) from the user's (called *actors* in UML) perspective. The use-case is the modeling approach of choice for the user model view. This important analysis representation describes a usage scenario from the end-user's perspective and has been discussed earlier.

Structural model view. Data and functionality are viewed from inside the system. That is, static structure (classes, objects, and relationships) is modeled.

Behavioral model view. This part of the analysis model represents the dynamic or behavioral aspects of the system. It also depicts the interactions or collaborations between various structural elements described in the user model and structural model views.

Implementation model view. The structural and behavioral aspects of the system are represented as they are to be built.

Environment model view. The structural and behavioral aspects of the environment in which the system is to be implemented are represented.

THE OOA PROCESS

The OOA process does not begin with a concern for objects. Rather, it begins with an understanding of the manner in which the system will be used—by people, if the system is human-interactive; by machines, if the system is involved in process control; or by other programs, if the system coordinates and controls applications. Once the scenario of usage has been defined, the modeling of the software begins. The sections that follow define a series of techniques that may be used to gather basic customer requirements and then define an analysis model for an object oriented system.

Use-Cases

As we noted in earlier chapter, use-cases model the system from the end-user's point of view. Created during requirements elicitation, use-cases should achieve the following objectives:

• To define the functional and operational requirements of the system (product) by defining a scenario of usage that is agreed upon by the end-user and the software engineering team.

• To provide a clear and unambiguous description of how the end-user and the system interact with one another.

• To provide a basis for validation testing.During OOA, use-cases serve as the basis for the first element of the analysis model. Using UML notation, a diagrammatic representation of a use-case, called a *use-case diagram,* can be created. Like many elements of the analysis model, the use-case diagram can be represented at many levels of abstraction. The use-case diagram contains actors and use-cases. *Actors* are entities that interact with the system. They can be human users or other machines or systems that have defined interfaces to the software.

Class-Responsibility-Collaborator Modeling

Once basic usage scenarios have been developed for the system, it is time to identify candidate classes and indicate their responsibilities and collaborations. Classresponsibility-collaborator (CRC) modeling [WIR90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements.

Ambler describes CRC modeling in the following way:

A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

In reality, the CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. Responsibilities are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does" [AMB95]. Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility.

In general, collaboration implies either a request for information or a request for some action.

Classes

To summarize, objects manifest themselves in a variety of forms: external entities, things, occurrences, or events; roles; organizational units; places; or structures. One technique for identifying these in the context of a software problem is to perform a grammatical parse on the processing narrative for the system. All nouns become potential objects. However, not every potential object makes the cut.

Six selection characteristics were defined:

1. Retained information. The potential object will be useful during analysis only if information about it must be remembered so that the system can function.

2. Needed services. The potential object must have a set of identifiable operations that can change the value of its attributes in some way.

3. Multiple attributes. During requirements analysis, the focus should be on "major" information; an object with a single attribute may, in fact, be useful during design but is probably better represented as an attribute of another object during the analysis activity.

4. Common attributes. A set of attributes can be defined for the potential object and these attributes apply to all occurrences of the object.

5. Common operations. A set of operations can be defined for the potential object and these operations apply to all occurrences of the object.

6. Essential requirements. External entities that appear in the problem space and produce or consume information that is essential to the operation of any solution for the system will almost always be defined as objects in the requirements model.

Class name:	
Class type: (e.g., device, property, role, event)	
Class characteristic: (e.g., tangible, atomic, concurrent)	
responsibilities:	collaborations:

A CRC MODEL INDEX CARD

A potential object should satisfy all six of these selection characteristics if it is to be considered for inclusion in the CRC model.

System intelligence should be evenly distributed.

Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of different ways. “Dumb” classes (those that have few responsibilities) can be modeled to act as servants to a few “smart” classes (those having many responsibilities. Although this approach makes the flow of control in a system straightforward, it has a few disadvantages:

(1) It concentrates all intelligence within a few classes, making changes more difficult

(2) it tends to require more classes, hence more development effort.

Therefore, system intelligence should be evenly distributed across the classes in an application. Because each object knows about and does only a few things (that are generally well focused), the cohesiveness of the system is improved. In addition, side effects due to change tend to be dampened because system intelligence has been decoupled across many objects.

Each responsibility should be stated as generally as possible.

This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses). In addition, polymorphism should beused in an effort to define operations that generally apply to the superclass but are implemented differently in each of the subclasses.

Information and the behavior related to it should reside within the same class.

This achieves the OO principle that we have called *encapsulation* .Data and the processes that manipulate the data should be packaged as a cohesive unit.

Information about one thing should be localized with a single class, not distributed across multiple classes.

A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes.

If information is distributed, software becomes more difficult to maintain and more challenging to test.

Responsibilities should be shared among related classes, when appropriate.

There are many cases in which a variety of related objects must all exhibit the same behavior at the same time. As an example, consider a video game that must display the following objects: player, player-body, player-arms, player-legs, player-head. Each of these objects has its own attributes (e.g., position, orientation, color, speed) and all must be updated and displayed as the user manipulates a joy stick. The responsibilities *update* and *display* must therefore be shared by each of the objects noted. Player knows when something has changed and *update* is required. It collaborates with the other objects to achieve a new position or orientation, but each object controls its own display.

Object-oriented analysis methods enable a software engineer to model a problem by representing both static and dynamic characteristics of classes and their relationships as the primary modeling components. Like earlier OO analysis methods, the Unified Modeling Language builds an analysis model that has the following characteristics:

(1) representation of classes and class hierarchies.

(2) creation of objectrelationship models

(3) derivation of object-behavior models.

Analysis for object-oriented systems occurs at many different levels of abstraction. At the business or enterprise level, the techniques associated with OOA can be coupled with a business process engineering approach. This technique is often called domain analysis. At an application level, the object model focuses on specific customer requirements as those

requirements affect the application to be built. The OOA process begins with the definition of use-cases—scenarios that describe how the OO system is to be used. The class-responsibility-collaborator modeling technique is then applied to document classes and their attributes and operations. It also provides an initial view of the collaborations that occur among objects. The next step in the OOA process is classification of objects and the creation of a class hierarchy.

Subsystems (packages) can be used to encapsulate related objects. The objectrelationship model provides an indication of how classes are connected to one another, and the object-behavior model indicates the behavior of individual objects and the overall behavior of the OO system.

UML – Overview:

UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.

OMG is continuously making efforts to create a truly industry standard.

- UML stands for Unified Modeling Language.
- UML is different from the other common programming languages such as C++, Java, COBOL, etc.
- UML is a pictorial language used to make software blueprints.
- UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
- Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.

UML is not a programming language but tools can be used to generate code in various languages using UML diagrams. UML has a direct relation with object oriented analysis and design. After some standardization, UML has become an OMG standard.

Goals of UML

A picture is worth a thousand words, this idiom absolutely fits describing UML. Object-oriented concepts were introduced much earlier than UML. At that point of time, there were no standard methodologies to organize and consolidate the object-oriented development. It was then that UML came into picture.

There are a number of goals for developing UML but the most important is to define some general purpose modeling language, which all modelers can use and it also needs to be made simple to understand and use.

UML diagrams are not only made for developers but also for business users, common people, and anybody interested to understand the system. The system can be a software or non-software system. Thus it must be clear that UML is not a development method rather it accompanies with processes to make it a successful system.

In conclusion, the goal of UML can be defined as a simple modeling mechanism to model all possible practical systems in today's complex environment.

A Conceptual Model of UML

To understand the conceptual model of UML, first we need to clarify what is a conceptual model? and why a conceptual model is required?

- A conceptual model can be defined as a model which is made of concepts and their relationships.

- A conceptual model is the first step before drawing a UML diagram. It helps to understand the entities in the real world and how they interact with each other.

As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements −

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

Object-Oriented Concepts

UML can be described as the successor of object-oriented (OO) analysis and design.

An object contains both data and methods that control the data. The data represents the state of the object. A class describes an object and they also form a hierarchy to model the real-world system. The hierarchy is represented as inheritance and the classes can also be associated in different ways as per the requirement.

Objects are the real-world entities that exist around us and the basic concepts such as abstraction, encapsulation, inheritance, and polymorphism all can be represented using UML.

UML is powerful enough to represent all the concepts that exist in object-oriented analysis and design. UML diagrams are representation of object-oriented concepts only. Thus, before learning UML, it becomes important to understand OO concept in detail.

Following are some fundamental concepts of the object-oriented world −

- Objects − Objects represent an entity and the basic building block.
- Class − Class is the blue print of an object.
- Abstraction − Abstraction represents the behavior of an real world entity.
- Encapsulation − Encapsulation is the mechanism of binding the data together and hiding them from the outside world.
- Inheritance − Inheritance is the mechanism of making new classes from existing ones.
- Polymorphism − It defines the mechanism to exists in different forms.

OO Analysis and Design

OO can be defined as an investigation and to be more specific, it is the investigation of objects. Design means collaboration of identified objects.

Thus, it is important to understand the OO analysis and design concepts. The most important purpose of OO analysis is to identify objects of a system to be designed. This analysis is also done for an existing system. Now an efficient analysis is only possible when we are able to start thinking in a way where objects can be identified. After identifying the objects, their relationships are identified and finally the design is produced.

The purpose of OO analysis and design can described as −

- Identifying the objects of a system.
- Identifying their relationships.
- Making a design, which can be converted to executables using OO languages.

There are three basic steps where the OO concepts are applied and implemented. The steps can be defined as

OO Analysis → OO Design → OO implementation using OO languages

The above three points can be described in detail as −

- During OO analysis, the most important purpose is to identify objects and describe them in a proper way. If these objects are identified efficiently, then the next job of design is easy. The objects should be identified with responsibilities. Responsibilities are the functions performed by the object. Each and every object has some type of responsibilities to be performed. When these responsibilities are collaborated, the purpose of the system is fulfilled.
- The second phase is OO design. During this phase, emphasis is placed on the requirements and their fulfilment. In this stage, the objects are collaborated according to their intended association. After the association is complete, the design is also complete.

- The third phase is OO implementation. In this phase, the design is implemented using OO languages such as Java, C++, etc.

Role of UML in OO Design

UML is a modeling language used to model software and non-software systems. Although UML is used for non-software systems, the emphasis is on modeling OO software applications. Most of the UML diagrams discussed so far are used to model different aspects such as static, dynamic, etc. Now whatever be the aspect, the artifacts are nothing but objects.

If we look into class diagram, object diagram, collaboration diagram, interaction diagrams all would basically be designed based on the objects.

Hence, the relation between OO design and UML is very important to understand. The OO design is transformed into UML diagrams according to the requirement. Before understanding the UML in detail, the OO concept should be learned properly. Once the OO analysis and design is done, the next step is very easy. The input from OO analysis and design is the input to UML diagrams.

UML - Building Blocks

As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements −

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

This chapter describes all the UML building blocks. The building blocks of UML can be defined as −

- Things

- Relationships
- Diagrams

Things

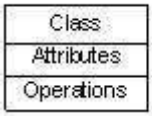
Things are the most important building blocks of UML. Things can be −

- Structural
- Behavioral
- Grouping
- Annotational

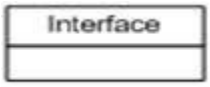
Structural Things

Structural things define the static part of the model. They represent the physical and conceptual elements. Following are the brief descriptions of the structural things.

Class − Class represents a set of objects having similar responsibilities.



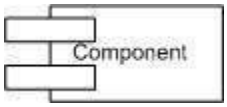
Interface − Interface defines a set of operations, which specify the responsibility of a class.



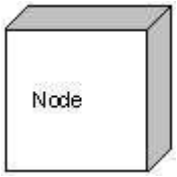
Collaboration −Collaboration defines an interaction between elements.

Use case −Use case represents a set of actions performed by a system for a specific goal.

Component −Component describes the physical part of a system.



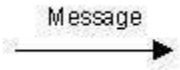
Node − A node can be defined as a physical element that exists at run time.



Behavioral Things

A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things −

Interaction − Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



State machine − State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change

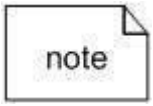
Grouping Things

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available −

Package − Package is the only one grouping thing available for gathering structural and behavioral things.

Annotational Things

Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note -** It is the only one Annotational thing available. A note is used to render comments, constraints, etc. of an UML element.



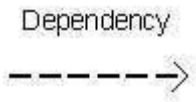
Relationship

Relationship is another most important building block of UML. It shows how the elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

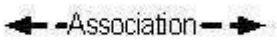
Dependency

Dependency is a relationship between two things in which change in one element also affects the other.



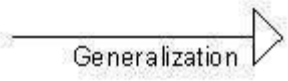
Association

Association is basically a set of links that connects the elements of a UML model. It also describes how many objects are taking part in that relationship.



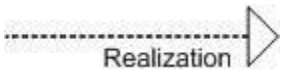
Generalization

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes the inheritance relationship in the world of objects.



Realization

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility, which is not implemented and the other one implements them. This relationship exists in case of interfaces.



UML Diagrams

UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it complete.

UML includes the following nine diagrams, the details of which are described in the subsequent chapters.

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram

- Collaboration diagram
- Activity diagram
- Statechart diagram
- Deployment diagram
- Component diagram

UML - Architecture

Any real-world system is used by different users. The users can be developers, testers, business people, analysts, and many more. Hence, before designing a system, the architecture is made with different perspectives in mind. The most important part is to visualize the system from the perspective of different viewers. The better we understand the better we can build the system.

UML plays an important role in defining different perspectives of a system. These perspectives are −

- Design
- Implementation
- Process
- Deployment

The center is the **Use Case** view which connects all these four. A **Use Case** represents the functionality of the system. Hence, other perspectives are connected with use case.

Design of a system consists of classes, interfaces, and collaboration. UML provides class diagram, object diagram to support this.

Implementation defines the components assembled together to make a complete physical system. UML component diagram is used to support the implementation perspective.

Process defines the flow of the system. Hence, the same elements as used in Design are also used to support this perspective.

Deployment represents the physical nodes of the system that forms the hardware. UML deployment diagram is used to support this perspective.

UML - Standard Diagrams

In the previous chapters, we have discussed about the building blocks and other necessary elements of UML. Now we need to understand where to use those elements.

The elements are like components which can be associated in different ways to make a complete UML picture, which is known as diagram. Thus, it is very important to understand the different diagrams to implement the knowledge in real-life systems.

Any complex system is best understood by making some kind of diagrams or pictures. These diagrams have a better impact on our understanding. If we look around, we will realize that the diagrams are not a new concept but it is used widely in different forms in different industries.

We prepare UML diagrams to understand the system in a better and simple way. A single diagram is not enough to cover all the aspects of the system. UML defines various kinds of diagrams to cover most of the aspects of a system.

You can also create your own set of diagrams to meet your requirements. Diagrams are generally made in an incremental and iterative way.

There are two broad categories of diagrams and they are again divided into subcategories −

- Structural Diagrams

- Behavioral Diagrams

Structural Diagrams

The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable.

These static parts are represented by classes, interfaces, objects, components, and nodes. The four structural diagrams are −

- Class diagram
- Object diagram
- Component diagram
- Deployment diagram

Class Diagram

Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations, and collaboration. Class diagrams basically represent the object-oriented view of a system, which is static in nature.

Active class is used in a class diagram to represent the concurrency of the system.

Class diagram represents the object orientation of a system. Hence, it is generally used for development purpose. This is the most widely used diagram at the time of system construction.

Object Diagram

Object diagrams can be described as an instance of class diagram. Thus, these diagrams are more close to real-life scenarios where we implement a system.

Object diagrams are a set of objects and their relationship is just like class diagrams. They also represent the static view of the system.

The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from a practical perspective.

Component Diagram

Component diagrams represent a set of components and their relationships. These components consist of classes, interfaces, or collaborations. Component diagrams represent the implementation view of a system.

During the design phase, software artifacts (classes, interfaces, etc.) of a system are arranged in different groups depending upon their relationship. Now, these groups are known as components.

Finally, it can be said component diagrams are used to visualize the implementation.

Deployment Diagram

Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed.

Deployment diagrams are used for visualizing the deployment view of a system. This is generally used by the deployment team.

Note − If the above descriptions and usages are observed carefully then it is very clear that all the diagrams have some relationship with one another. Component diagrams are dependent upon the classes, interfaces, etc. which are part of class/object diagram. Again, the deployment diagram is dependent upon the components, which are used to make component diagrams.

Behavioral Diagrams

Any system can have two aspects, static and dynamic. So, a model is considered as complete when both the aspects are fully covered.

Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspect can be further described as the changing/moving parts of a system.

UML has the following five types of behavioral diagrams −

- Use case diagram
- Sequence diagram
- Collaboration diagram
- Statechart diagram
- Activity diagram

Use Case Diagram

Use case diagrams are a set of use cases, actors, and their relationships. They represent the use case view of a system.

A use case represents a particular functionality of a system. Hence, use case diagram is used to describe the relationships among the functionalities and their internal/external controllers. These controllers are known as **actors**.

Sequence Diagram

A sequence diagram is an interaction diagram. From the name, it is clear that the diagram deals with some sequences, which are the sequence of messages flowing from one object to another.

Interaction among the components of a system is very important from implementation and execution perspective. Sequence diagram is used to visualize the sequence of calls in a system to perform a specific functionality.

Collaboration Diagram

Collaboration diagram is another form of interaction diagram. It represents the structural organization of a system and the messages sent/received. Structural organization consists of objects and links.

The purpose of collaboration diagram is similar to sequence diagram. However, the specific purpose of collaboration diagram is to visualize the organization of objects and their interaction.

Statechart Diagram

Any real-time system is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system.

Statechart diagram is used to represent the event driven state change of a system. It basically describes the state change of a class, interface, etc.

State chart diagram is used to visualize the reaction of a system by internal/external factors.

Activity Diagram

Activity diagram describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched.

Activities are nothing but the functions of a system. Numbers of activity diagrams are prepared to capture the entire flow in a system.

Activity diagrams are used to visualize the flow of controls in a system. This is prepared to have an idea of how the system will work when executed.

Note − Dynamic nature of a system is very difficult to capture. UML has provided features to capture the dynamics of a system from different angles. Sequence diagrams and collaboration diagrams are isomorphic, hence they can be converted from one another without losing any information. This is also true for Statechart and activity diagram.

UML - Class Diagram

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

Class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of objectoriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

Purpose of Class Diagrams

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application, however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as −

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

How to Draw a Class Diagram?

Class diagrams are the most popular UML diagrams used for construction of software applications. It is very important to learn the drawing procedure of class diagram.

Class diagrams have a lot of properties to consider while drawing but here the diagram will be considered from a top level view.

Class diagram is basically a graphical representation of the static view of the system and represents different aspects of the application. A collection of class diagrams represent the whole system.

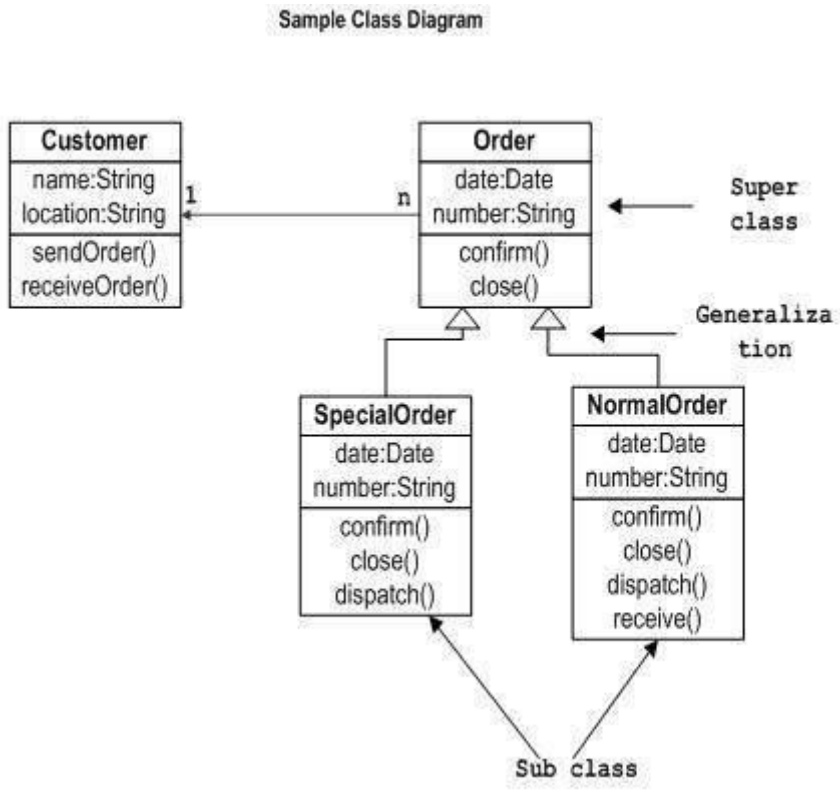
The following points should be remembered while drawing a class diagram −

- The name of the class diagram should be meaningful to describe the aspect of the system.
- Each element and their relationships should be identified in advance.
- Responsibility (attributes and methods) of each class should be clearly identified
- For each class, minimum number of properties should be specified, as unnecessary properties will make the diagram complicated.
- Use notes whenever required to describe some aspect of the diagram. At the end of the drawing it should be understandable to the developer/coder.
- Finally, before making the final version, the diagram should be drawn on plain paper and reworked as many times as possible to make it correct.

The following diagram is an example of an Order System of an application. It describes a particular aspect of the entire application.

- First of all, Order and Customer are identified as the two elements of the system. They have a one-to-many relationship because a customer can have multiple orders.
- Order class is an abstract class and it has two concrete classes (inheritance relationship) SpecialOrder and NormalOrder.
- The two inherited classes have all the properties as the Order class. In addition, they have additional functions like dispatch () and receive ().

The following class diagram has been drawn considering all the points mentioned above.



Where to Use Class Diagrams?

Class diagram is a static diagram and it is used to model the static view of a system. The static view describes the vocabulary of the system.

Class diagram is also considered as the foundation for component and deployment diagrams. Class diagrams are not only used to visualize the static view of the system but they are also used to construct the executable code for forward and reverse engineering of any system.

Generally, UML diagrams are not directly mapped with any object-oriented programming languages but the class diagram is an exception.

Class diagram clearly shows the mapping with object-oriented languages such as Java, C++, etc. From practical experience, class diagram is generally used for construction purpose.

In a nutshell it can be said, class diagrams are used for −

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.

- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.

UML - Object Diagrams

Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.

Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams. Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.

Object diagrams are used to render a set of objects and their relationships as an instance.

Purpose of Object Diagrams

The purpose of a diagram should be understood clearly to implement it practically. The purposes of object diagrams are similar to class diagrams.

The difference is that a class diagram represents an abstract model consisting of classes and their relationships. However, an object diagram represents an instance at a particular moment, which is concrete in nature.

It means the object diagram is closer to the actual system behavior. The purpose is to capture the static view of a system at a particular moment.

The purpose of the object diagram can be summarized as −

- Forward and reverse engineering.
- Object relationships of a system
- Static view of an interaction.
- Understand object behaviour and their relationship from practical perspective

How to Draw an Object Diagram?

We have already discussed that an object diagram is an instance of a class diagram. It implies that an object diagram consists of instances of things used in a class diagram.

So both diagrams are made of same basic elements but in different form. In class diagram elements are in abstract form to represent the blue print and in object diagram the elements are in concrete form to represent the real world object.

To capture a particular system, numbers of class diagrams are limited. However, if we consider object diagrams then we can have unlimited number of instances, which are unique in nature. Only those instances are considered, which have an impact on the system.

From the above discussion, it is clear that a single object diagram cannot capture all the necessary instances or rather cannot specify all the objects of a system. Hence, the solution is –

- First, analyze the system and decide which instances have important data and association.
- Second, consider only those instances, which will cover the functionality.
- Third, make some optimization as the number of instances are unlimited.

Before drawing an object diagram, the following things should be remembered and understood clearly −

- Object diagrams consist of objects.
- The link in object diagram is used to connect objects.
- Objects and links are the two elements used to construct an object diagram.

After this, the following things are to be decided before starting the construction of the diagram −

- The object diagram should have a meaningful name to indicate its purpose.

- The most important elements are to be identified.
- The association among objects should be clarified.
- Values of different elements need to be captured to include in the object diagram.
- Add proper notes at points where more clarity is required.

The following diagram is an example of an object diagram. It represents the Order management system which we have discussed in the chapter Class Diagram. The following diagram is an instance of the system at a particular time of purchase. It has the following objects.

- Customer
- Order
- SpecialOrder
- NormalOrder

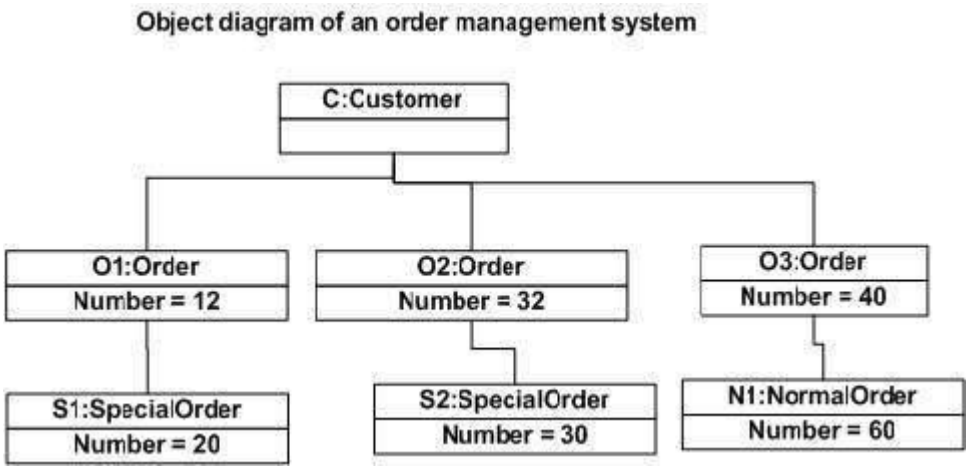
Now the customer object (C) is associated with three order objects (O1, O2, and O3). These order objects are associated with special order and normal order objects (S1, S2, and N1). The customer has the following three orders with different numbers (12, 32 and 40) for the particular time considered.

The customer can increase the number of orders in future and in that scenario the object diagram will reflect that. If order, special order, and normal order objects are observed then you will find that they have some values.

For orders, the values are 12, 32, and 40 which implies that the objects have these values for a particular moment (here the particular time when the purchase is made is considered as the moment) when the instance is captured

The same is true for special order and normal order objects which have number of orders as 20, 30, and 60. If a different time of purchase is considered, then these values will change accordingly.

The following object diagram has been drawn considering all the points mentioned above



Where to Use Object Diagrams?

Object diagrams can be imagined as the snapshot of a running system at a particular moment. Let us consider an example of a running train

Now, if you take a snap of the running train then you will find a static picture of it having the following −

- A particular state which is running.
- A particular number of passengers. which will change if the snap is taken in a different time

Here, we can imagine the snap of the running train is an object having the above values. And this is true for any real-life simple or complex system.

In a nutshell, it can be said that object diagrams are used for −

- Making the prototype of a system.
- Reverse engineering.
- Modeling complex data structures.
- Understanding the system from practical perspective.

UML - Component Diagrams

Component diagrams are different in terms of nature and behavior. Component diagrams are used to model the physical aspects of a system. Now the question is, what are these physical aspects? Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.

Component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose of Component Diagrams

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.

The purpose of the component diagram can be summarized as −

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

How to Draw a Component Diagram?

Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executables, libraries, etc

The purpose of this diagram is different. Component diagrams are used during the implementation phase of an application. However, it is prepared well in advance to visualize the implementation details.

Initially, the system is designed using different UML diagrams and then when the artifacts are ready, component diagrams are used to get an idea of the implementation.

This diagram is very important as without it the application cannot be implemented efficiently. A well-prepared component diagram is also important for other aspects such as application performance, maintenance, etc.

Before drawing a component diagram, the following artifacts are to be identified clearly −

- Files used in the system.
- Libraries and other artifacts relevant to the application.
- Relationships among the artifacts.

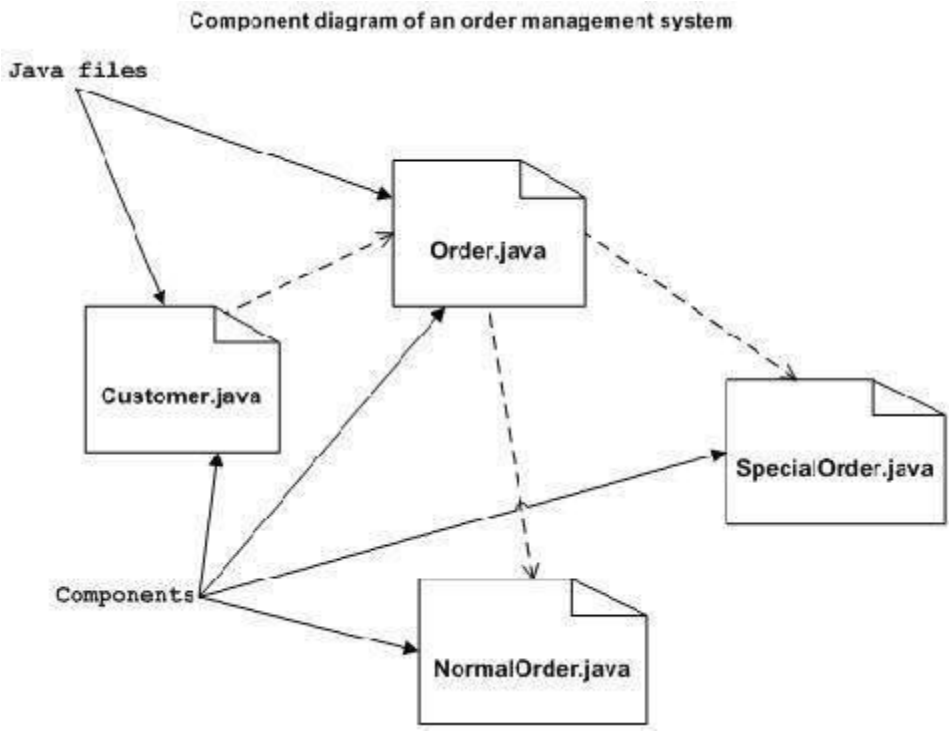
After identifying the artifacts, the following points need to be kept in mind.

- Use a meaningful name to identify the component for which the diagram is to be drawn.
- Prepare a mental layout before producing the using tools.
- Use notes for clarifying important points.

Following is a component diagram for order management system. Here, the artifacts are files. The diagram shows the files in the application and their relationships. In actual, the component diagram also contains dlls, libraries, folders, etc.

In the following diagram, four files are identified and their relationships are produced. Component diagram cannot be matched directly with other UML diagrams discussed so far as it is drawn for completely different purpose.

The following component diagram has been drawn considering all the points mentioned above.



Where to Use Component Diagrams?

We have already described that component diagrams are used to visualize the static implementation view of a system. Component diagrams are special type of UML diagrams used for different purposes.

These diagrams show the physical components of a system. To clarify it, we can say that component diagrams describe the organization of the components in a system.

Organization can be further described as the location of the components in a system. These components are organized in a special way to meet the system requirements.

As we have already discussed, those components are libraries, files, executables, etc. Before implementing the application, these components are to be organized. This component organization is also designed separately as a part of project execution.

Component diagrams are very important from implementation perspective. Thus, the implementation team of an application should have a proper knowledge of the component details

Component diagrams can be used to −

- Model the components of a system.
- Model the database schema.
- Model the executables of an application.
- Model the system's source code.

UML - Deployment Diagrams

Deployment diagrams are used to visualize the topology of the physical components of a system, where the software components are deployed.

Deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationships.

Purpose of Deployment Diagrams

The term Deployment itself describes the purpose of the diagram. Deployment diagrams are used for describing the hardware components, where software components are deployed. Component diagrams and deployment diagrams are closely related.

Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware.

UML is mainly designed to focus on the software artifacts of a system. However, these two diagrams are special diagrams used to focus on software and hardware components.

Most of the UML diagrams are used to handle logical components but deployment diagrams are made to focus on the hardware topology of a system. Deployment diagrams are used by the system engineers.

The purpose of deployment diagrams can be described as −

- Visualize the hardware topology of a system.
- Describe the hardware components used to deploy software components.
- Describe the runtime processing nodes.

How to Draw a Deployment Diagram?

Deployment diagram represents the deployment view of a system. It is related to the component diagram because the components are deployed using the deployment diagrams. A deployment diagram consists of nodes. Nodes are nothing but physical hardware used to deploy the application.

Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important as it controls the following parameters −

- Performance
- Scalability
- Maintainability
- Portability

Before drawing a deployment diagram, the following artifacts should be identified −

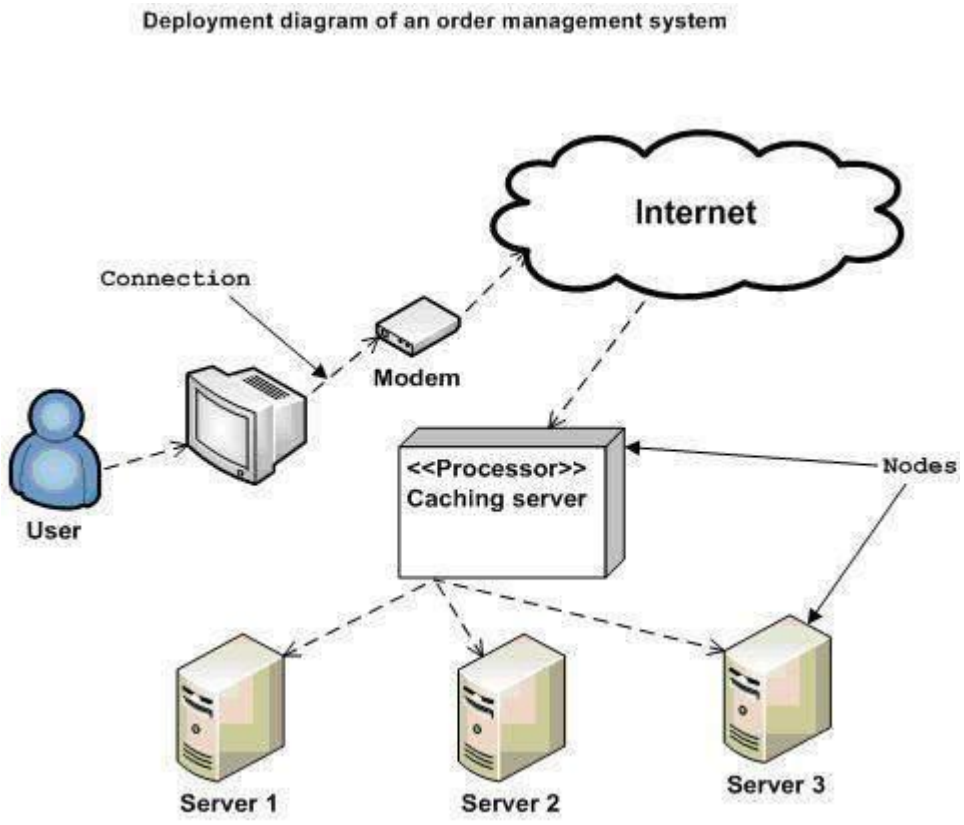
- Nodes
- Relationships among nodes

Following is a sample deployment diagram to provide an idea of the deployment view of order management system. Here, we have shown nodes as −

- Monitor
- Modem
- Caching server
- Server

The application is assumed to be a web-based application, which is deployed in a clustered environment using server 1, server 2, and server 3. The user connects to the application using the Internet. The control flows from the caching server to the clustered environment.

The following deployment diagram has been drawn considering all the points mentioned above.



Where to Use Deployment Diagrams?

Deployment diagrams are mainly used by system engineers. These diagrams are used to describe the physical components (hardware), their distribution, and association.

Deployment diagrams can be visualized as the hardware components/nodes on which the software components reside.

Software applications are developed to model complex business processes. Efficient software applications are not sufficient to meet the business requirements. Business requirements can be described as the need to support the increasing number of users, quick response time, etc.

To meet these types of requirements, hardware components should be designed efficiently and in a cost-effective way.

Now-a-days software applications are very complex in nature. Software applications can be standalone, web-based, distributed, mainframe-based and many more. Hence, it is very important to design the hardware components efficiently.

Deployment diagrams can be used −

- To model the hardware topology of a system.
- To model the embedded system.
- To model the hardware details for a client/server system.
- To model the hardware details of a distributed application.
- For Forward and Reverse engineering.

UML - Use Case Diagrams

To model a system, the most important aspect is to capture the dynamic behavior. Dynamic behavior means the behavior of the system when it is running/operating.

Only static behavior is not sufficient to model a system rather dynamic behavior is more important than static behavior. In UML, there are five diagrams available to model the dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case diagram is dynamic in nature, there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. Use case diagrams consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system.

Hence to model the entire system, a number of use case diagrams are used.

Purpose of Use Case Diagrams

The purpose of use case diagram is to capture the dynamic aspect of a system. However, this definition is too generic to describe the purpose, as other four diagrams (activity, sequence, collaboration, and Statechart) also have the same purpose. We will look into some specific purpose, which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified.

When the initial task is complete, use case diagrams are modelled to present the outside view.

In brief, the purposes of use case diagrams can be said to be as follows −

- Used to gather the requirements of a system.
- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.
- Show the interaction among the requirements are actors.

How to Draw a Use Case Diagram?

Use case diagrams are considered for high level requirement analysis of a system. When the requirements of a system are analyzed, the functionalities are captured in use cases.

We can say that use cases are nothing but the system functionalities written in an organized manner. The second thing which is relevant to use cases are the actors. Actors can be defined as something that interacts with the system.

Actors can be a human user, some internal applications, or may be some external applications. When we are planning to draw a use case diagram, we should have the following items identified.

- Functionalities to be represented as use case
- Actors
- Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. After identifying the above items, we have to use the following guidelines to draw an efficient use case diagram

- The name of a use case is very important. The name should be chosen in such a way so that it can identify the functionalities performed.
- Give a suitable name for actors.
- Show relationships and dependencies clearly in the diagram.
- Do not try to include all types of relationships, as the main purpose of the diagram is to identify the requirements.
- Use notes whenever required to clarify some important points.

Following is a sample use case diagram representing the order management system. Hence, if we look into the diagram then we will find three use cases **(Order, SpecialOrder, and NormalOrder)** and one actor which is the customer.

The SpecialOrder and NormalOrder use cases are extended from *Order* use case. Hence, they have extended relationship. Another important point is to identify the system boundary, which is shown in the picture. The actor Customer lies outside the system as it is an external user of the system.

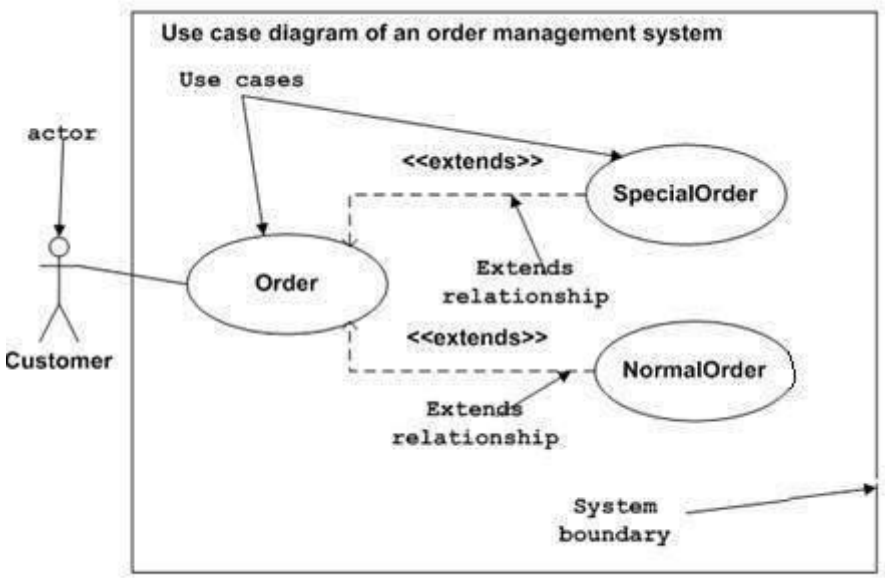


Figure: Sample Use Case diagram

Where to Use a Use Case Diagram?

As we have already discussed there are five diagrams in UML to model the dynamic view of a system. Now each and every model has some specific purpose to use. Actually these specific purposes are different angles of a running system.

To understand the dynamics of a system, we need to use different types of diagrams. Use case diagram is one of them and its specific purpose is to gather system requirements and actors.

Use case diagrams specify the events of a system and their flows. But use case diagram never describes how they are implemented. Use case diagram can be imagined as a black box where only the input, output, and the function of the black box is known.

These diagrams are used at a very high level of design. This high level design is refined again and again to get a complete and practical picture of the system. A well-structured use case also describes the pre-condition, post condition, and exceptions. These extra elements are used to make test cases when performing the testing.

Although use case is not a good candidate for forward and reverse engineering, still they are used in a slightly different way to make forward and reverse engineering. The same is true for reverse engineering. Use case diagram is used differently to make it suitable for reverse engineering.

In forward engineering, use case diagrams are used to make test cases and in reverse engineering use cases are used to prepare the requirement details from the existing application.

Use case diagrams can be used for −

- Requirement analysis and high level design.
- Model the context of a system.
- Reverse engineering.
- Forward engineering.

UML - Interaction Diagrams

From the term Interaction, it is clear that the diagram is used to describe some type of interactions among the different elements in the model. This interaction is a part of dynamic behavior of the system.

This interactive behavior is represented in UML by two diagrams known as **Sequence diagram** and **Collaboration diagram**. The basic purpose of both the diagrams are similar.

Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.

Purpose of Interaction Diagrams

The purpose of interaction diagrams is to visualize the interactive behavior of the system. Visualizing the interaction is a difficult task. Hence, the solution is to use different types of models to capture the different aspects of the interaction.

Sequence and collaboration diagrams are used to capture the dynamic nature but from a different angle.

The purpose of interaction diagram is −

- To capture the dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.

How to Draw an Interaction Diagram?

As we have already discussed, the purpose of interaction diagrams is to capture the dynamic aspect of a system. So to capture the dynamic aspect, we need to understand what a dynamic aspect is and how it is visualized. Dynamic aspect can be defined as the snapshot of the running system at a particular moment

We have two types of interaction diagrams in UML. One is the sequence diagram and the other is the collaboration diagram. The sequence diagram captures the time sequence of the message flow from one object to another and the collaboration diagram describes the organization of objects in a system taking part in the message flow.

Following things are to be identified clearly before drawing the interaction diagram

- Objects taking part in the interaction.
- Message flows among the objects.
- The sequence in which the messages are flowing.
- Object organization.

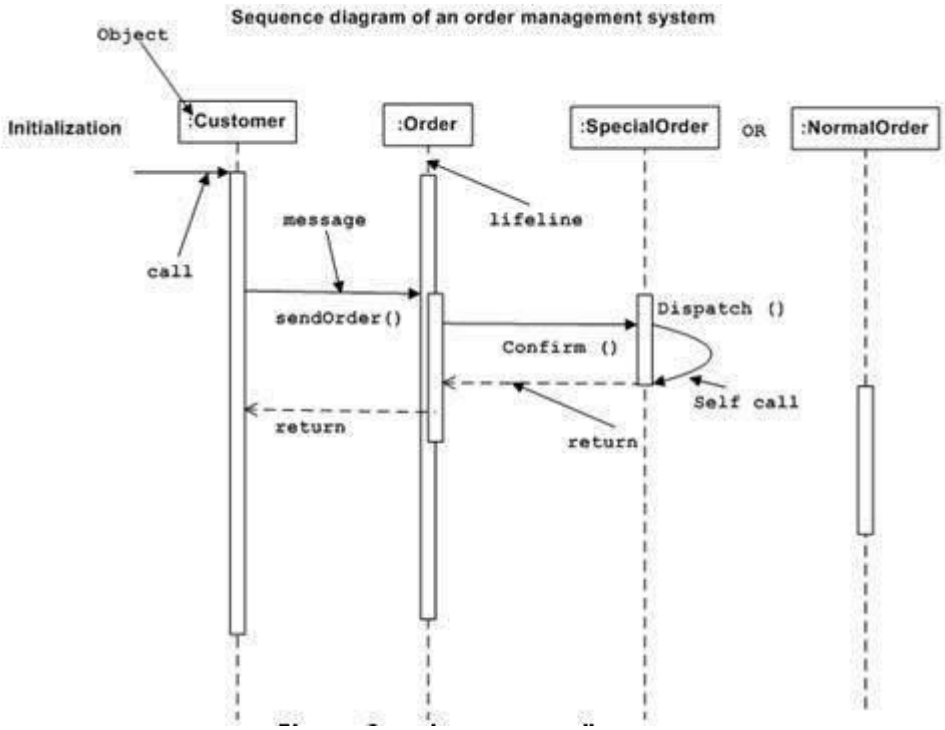
Following are two interaction diagrams modeling the order management system. The first diagram is a sequence diagram and the second is a collaboration diagram

The Sequence Diagram

The sequence diagram has four objects (Customer, Order, SpecialOrder and NormalOrder).

The following diagram shows the message sequence for *SpecialOrder* object and the same can be used in case of *NormalOrder* object. It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

The first call is *sendOrder ()* which is a method of *Order object*. The next call is *confirm ()* which is a method of *SpecialOrder* object and the last call is *Dispatch ()* which is a method of *SpecialOrder* object. The following diagram mainly describes the method calls from one object to another, and this is also the actual scenario when the system is running.

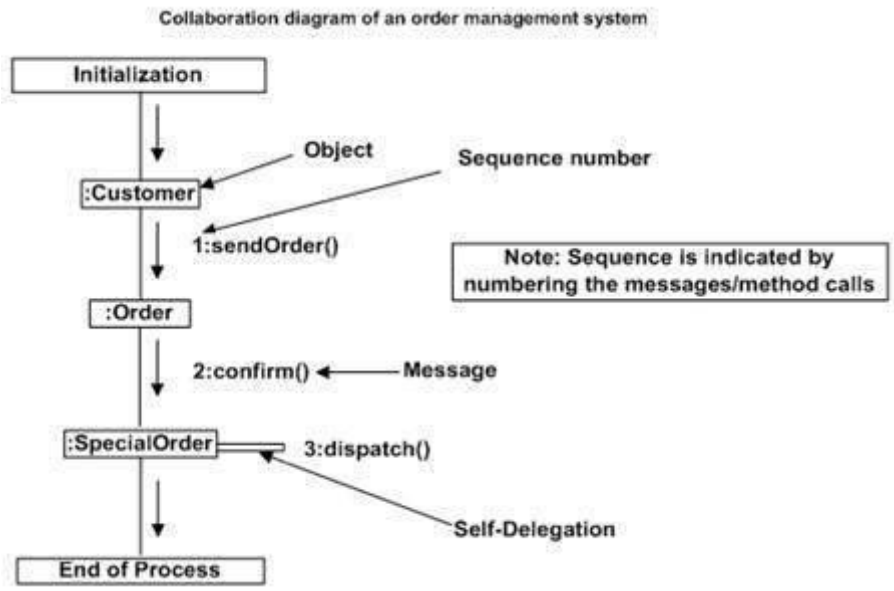


The Collaboration Diagram

The second interaction diagram is the collaboration diagram. It shows the object organization as seen in the following diagram. In the collaboration diagram, the method call sequence is indicated by some numbering technique. The number indicates how the methods are called one after another. We have taken the same order management system to describe the collaboration diagram.

Method calls are similar to that of a sequence diagram. However, difference being the sequence diagram does not describe the object organization, whereas the collaboration diagram shows the object organization.

To choose between these two diagrams, emphasis is placed on the type of requirement. If the time sequence is important, then the sequence diagram is used. If organization is required, then collaboration diagram is used.



Where to Use Interaction Diagrams?

We have already discussed that interaction diagrams are used to describe the dynamic nature of a system. Now, we will look into the practical scenarios where these diagrams are used. To understand the practical application, we need to understand the basic nature of sequence and collaboration diagram.

The main purpose of both the diagrams are similar as they are used to capture the dynamic behavior of a system. However, the specific purpose is more important to clarify and understand.

Sequence diagrams are used to capture the order of messages flowing from one object to another. Collaboration diagrams are used to describe the structural organization of the objects taking part in the interaction. A single diagram is not sufficient to describe the dynamic aspect of an entire system, so a set of diagrams are used to capture it as a whole.

Interaction diagrams are used when we want to understand the message flow and the structural organization. Message flow means the sequence of control flow from one object to another. Structural organization means the visual organization of the elements in a system.

Interaction diagrams can be used −

- To model the flow of control by time sequence.
- To model the flow of control by structural organizations.
- For forward engineering.
- For reverse engineering.

UML - Statechart Diagrams

The name of the diagram itself clarifies the purpose of the diagram and other details. It describes different states of a component in a system. The states are specific to a component/object of a system.

A Statechart diagram describes a state machine. State machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.

Activity diagram explained in the next chapter, is a special kind of a Statechart diagram. As Statechart diagram defines the states, it is used to model the lifetime of an object.

Purpose of Statechart Diagrams

Statechart diagram is one of the five UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events. Statechart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of Statechart diagram is to model lifetime of an object from creation to termination.

Statechart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

Following are the main purposes of using Statechart diagrams −

- To model the dynamic aspect of a system.
- To model the life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model the states of an object.

How to Draw a Statechart Diagram?

Statechart diagram is used to describe the states of different objects in its life cycle. Emphasis is placed on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

Statechart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

Before drawing a Statechart diagram we should clarify the following points −

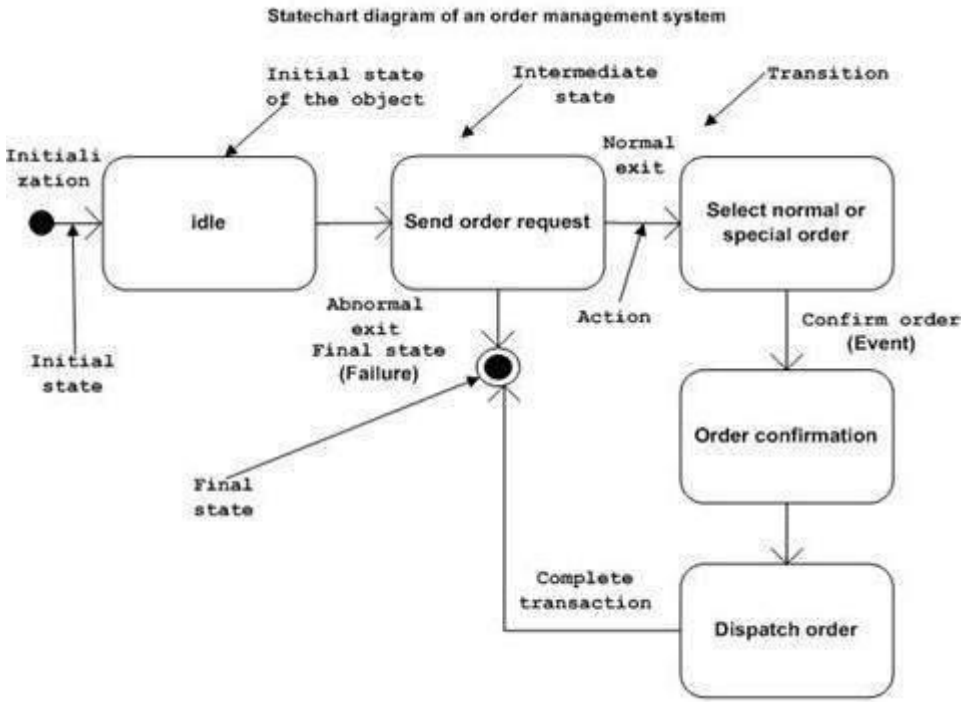
- Identify the important objects to be analyzed.
- Identify the states.
- Identify the events.

Following is an example of a Statechart diagram where the state of Order object is analyzed

The first state is an idle state from where the process starts. The next states are arrived for events like send request, confirm request, and dispatch order. These events are responsible for the state changes of order object.

During the life cycle of an object (here order object) it goes through the following states and there may be some abnormal exits. This abnormal exit may occur due to some problem in the

system. When the entire life cycle is complete, it is considered as a complete transaction as shown in the following figure. The initial and final state of an object is also shown in the following figure.



Where to Use Statechart Diagrams?

From the above discussion, we can define the practical applications of a Statechart diagram. Statechart diagrams are used to model the dynamic aspect of a system like other four diagrams discussed in this tutorial. However, it has some distinguishing characteristics for modeling the dynamic nature.

Statechart diagram defines the states of a component and these state changes are dynamic in nature. Its specific purpose is to define the state changes triggered by events. Events are internal or external factors influencing the system.

Statechart diagrams are used to model the states and also the events operating on the system. When implementing a system, it is very important to clarify different states of an object during its life time and Statechart diagrams are used for this purpose. When these states and events are identified, they are used to model it and these models are used during the implementation of the system.

If we look into the practical implementation of Statechart diagram, then it is mainly used to analyze the object states influenced by events. This analysis is helpful to understand the system behavior during its execution.

The main usage can be described as −

- To model the object states of a system.
- To model the reactive system. Reactive system consists of reactive objects.
- To identify the events responsible for state changes.
- Forward and reverse engineering.

UML - Activity Diagrams

Activity diagram is another important diagram in UML to describe the dynamic aspects of the system.

Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system.

The control flow is drawn from one operation to another. This flow can be sequential, branched, or concurrent. Activity diagrams deal with all type of flow control by using different elements such as fork, join, etc

Purpose of Activity Diagrams

The basic purposes of activity diagrams is similar to other four diagrams. It captures the dynamic behavior of the system. Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable

system by using forward and reverse engineering techniques. The only missing thing in the activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is sometimes considered as the flowchart. Although the diagrams look like a flowchart, they are not. It shows different flows such as parallel, branched, concurrent, and single.

The purpose of an activity diagram can be described as −

- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.

How to Draw an Activity Diagram?

Activity diagrams are mainly used as a flowchart that consists of activities performed by the system. Activity diagrams are not exactly flowcharts as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane, etc.

Before drawing an activity diagram, we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities, we need to understand how they are associated with constraints and conditions.

Before drawing an activity diagram, we should identify the following elements −

- Activities
- Association
- Conditions
- Constraints

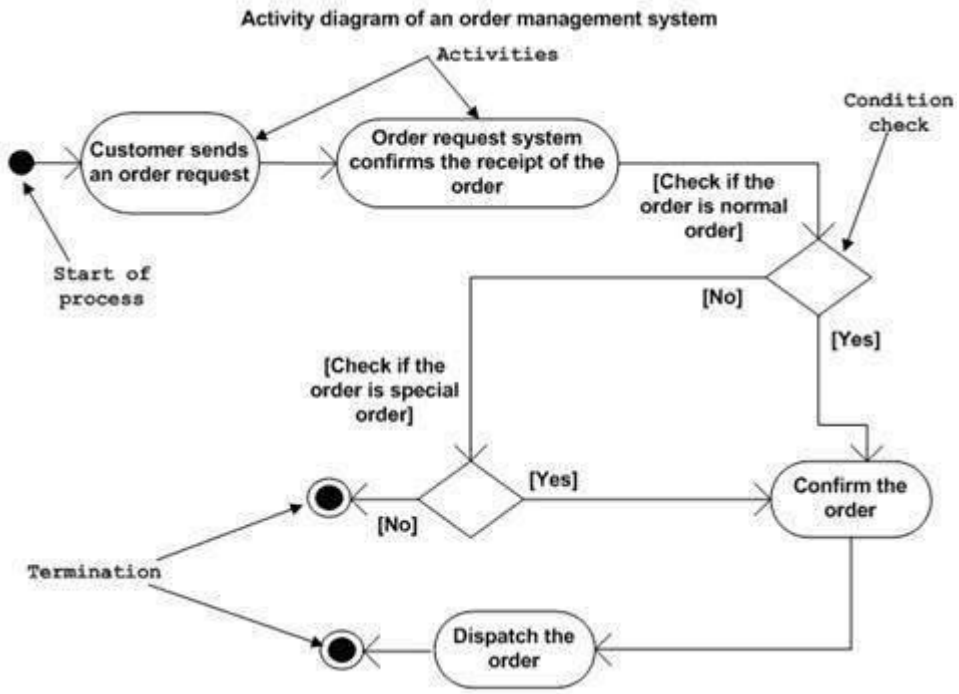
Once the above-mentioned parameters are identified, we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

Following is an example of an activity diagram for order management system. In the diagram, four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and is mainly used by the business users

Following diagram is drawn with the four main activities −

- Send order by the customer
- Receipt of the order
- Confirm the order
- Dispatch the order

After receiving the order request, condition checks are performed to check if it is normal or special order. After the type of order is identified, dispatch activity is performed and that is marked as the termination of the process.



Where to Use Activity Diagrams?

The basic usage of activity diagram is similar to other four UML diagrams. The specific usage is to model the control flow from one activity to another. This control flow does not include messages.

Activity diagram is suitable for modeling the activity flow of the system. An application can have multiple systems. Activity diagram also captures these systems and describes the flow from one system to another. This specific usage is not available in other diagrams. These systems can be database, external queues, or any other system.

We will now look into the practical applications of the activity diagram. From the above discussion, it is clear that an activity diagram is drawn from a very high level. So it gives high level view of a system. This high level view is mainly for business users or any other person who is not a technical person.

This diagram is used to model the activities which are nothing but business requirements. The diagram has more impact on business understanding rather than on implementation details.

Activity diagram can be used for −

- Modeling work flow by using activities.
- Modeling business requirements.
- High level understanding of the system's functionalities.
- Investigating business requirements at a later stage.

UML 2.0 - Overview

UML 2.0 is totally a different dimension in the world of Unified Modeling Language. It is more complex and extensive in nature. The extent of documentation has also increased compared to UML 1.5 version. UML 2.0 has added new features so that its usage can be more extensive.

UML 2.0 adds the definition of formal and completely defined semantics. This new possibility can be utilized for the development of models and the corresponding systems can be generated from these models. However, to utilize this new dimension, a considerable effort has to be made to acquire knowledge.

New Dimensions in UML 2.0

The structure and documentation of UML was completely revised in the latest version of UML 2.0. There are now two documents available that describe UML −

- UML 2.0 Infrastructure defines the basic constructs of the language on which UML is based. This section is not directly relevant to the users of UML. This is directed more towards the developers of modeling tools. This area is not in the scope of this tutorial.

- UML 2.0 Superstructure defines the user constructs of UML 2.0. It means those elements of UML that the users will use at the immediate level. This is the main focus for the user community of UML.

This revision of UML was created to fulfil a goal to restructure and refine UML so that usability, implementation, and adaptation are simplified.

UML infrastructure is used to −

- Provide a reusable meta-language core. This is used to define UML itself.

- Provide mechanisms to adjustment the language.

UML superstructure is used to −

- Provide better support for component-based development.

- Improve constructs for the specification of architecture.

- Provide better options for the modeling of behavior.

The important point to note is the major divisions described above. These divisions are used to increase the usability of UML and define a clear understanding of its usage.

There is another dimension which is already proposed in this new version. It is a proposal for a completely new Object Constraint Language (OCL) and Diagram Interchange. These features all together form the complete UML 2.0 package.

Modeling Diagrams in UML 2.0

Modeling Interactions

The interaction diagrams described in UML 2.0 is different than the earlier versions. However, the basic concept remains the same as the earlier version. The major difference is the enhancement and additional features added to the diagrams in UML 2.0.

UML 2.0 models object interaction in the following four different ways.

- **Sequence diagram** is a time dependent view of the interaction between objects to accomplish a behavioral goal of the system. The time sequence is similar to the earlier version of sequence diagram. An interaction may be designed at any level of abstraction within the system design, from subsystem interactions to instancelevel.

- **Communication diagram** is a new name added in UML 2.0. Communication diagram is a structural view of the messaging between objects, taken from the Collaboration diagram concept of UML 1.4 and earlier versions. This can be defined as a modified version of collaboration diagram.

- **Interaction Overview diagram** is also a new addition in UML 2.0. An Interaction Overview diagram describes a high-level view of a group of interactions combined into a logic sequence, including flow-control logic to navigate between the interactions.

- **Timing diagram** is also added in UML 2.0. It is an optional diagram designed to specify the time constraints on the messages sent and received in the course of an interaction.

From the above description, it is important to note that the purpose of all the diagrams are to send/receive messages. The handling of these messages are internal to the objects. Hence, the objects also have options to receive and send messages, and here comes another important

aspect called interface. Now these interfaces are responsible for accepting and sending messages to one another.

It can thus be concluded that the interactions in UML 2.0 are described in a different way and that is the reason why the new diagram names have come into picture. If we analyze the new diagrams then it is clear that all the diagrams are created based upon the interaction diagrams described in the earlier versions. The only difference is the additional features added in UML 2.0 to make the diagrams more efficient and purpose oriented.

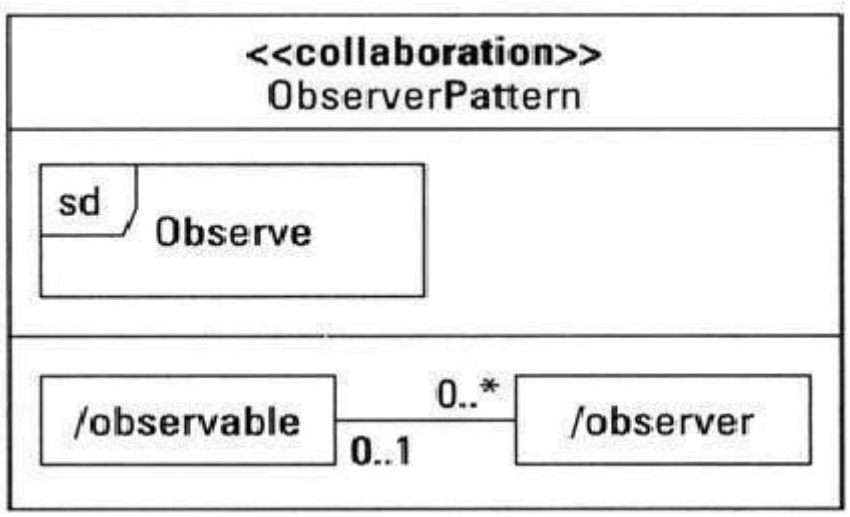
Modeling Collaborations

As we have already discussed, collaboration is used to model common interactions between objects. We can say that collaboration is an interaction where a set of messages are handled by a set of objects having pre-defined roles.

The important point to note is the difference between the collaboration diagram in the earlier version and in UML 2.0 version. To distinguish, the name of the collaboration diagram has been changed in UML 2.0. In UML 2.0, it is named as **Communication diagram.**

Consequently, collaboration is defined as a class with attributes (properties) and behavior (operations). Compartments on the collaboration class can be user defined and may be used for interactions (Sequence diagrams) and structural elements (Composite Structure diagram).

Following figure models the Observer design pattern as collaboration between an object in the role of an observable item and any number of objects as the observers.



Modeling Communication

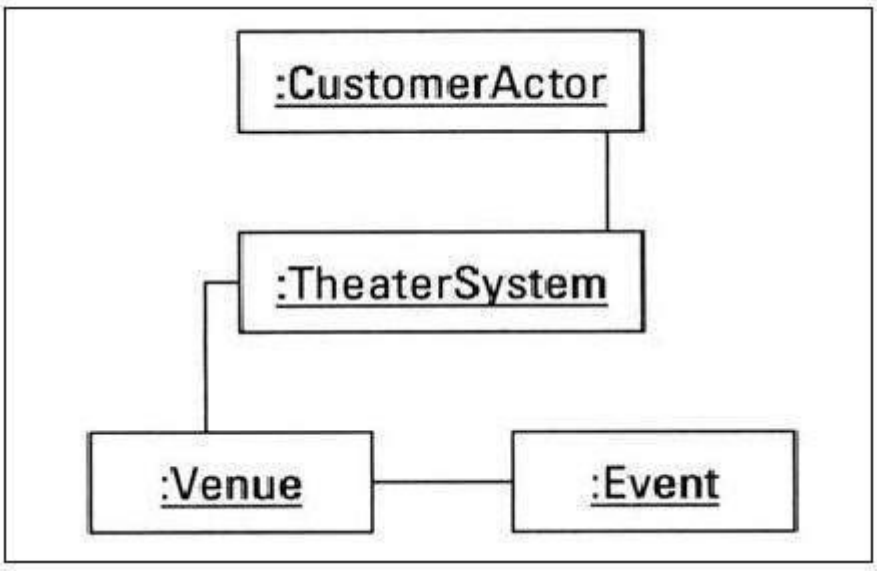
Communication diagram is slightly different than the collaboration diagrams of the earlier versions. We can say it is a scaled back version of the earlier UML versions. The distinguishing factor of the communication diagram is the link between objects.

This is a visual link and it is missing in the sequence diagram. In the sequence diagram, only the messages passed between the objects are shown even if there is no link between them.

Communication diagram is used to prevent the modeler from making this mistake by using an Object diagram format as the basis for messaging. Each object on a Communication diagram is called an object lifeline.

The message types in a Communication diagram are the same as in a Sequence diagram. Communication diagram may model synchronous, asynchronous, return, lost, found, a object-creation messages.

Following figure shows an Object diagram with three objects and two links that form the basis for the Communication diagram. Each object on a Communication diagram is called an object lifeline.



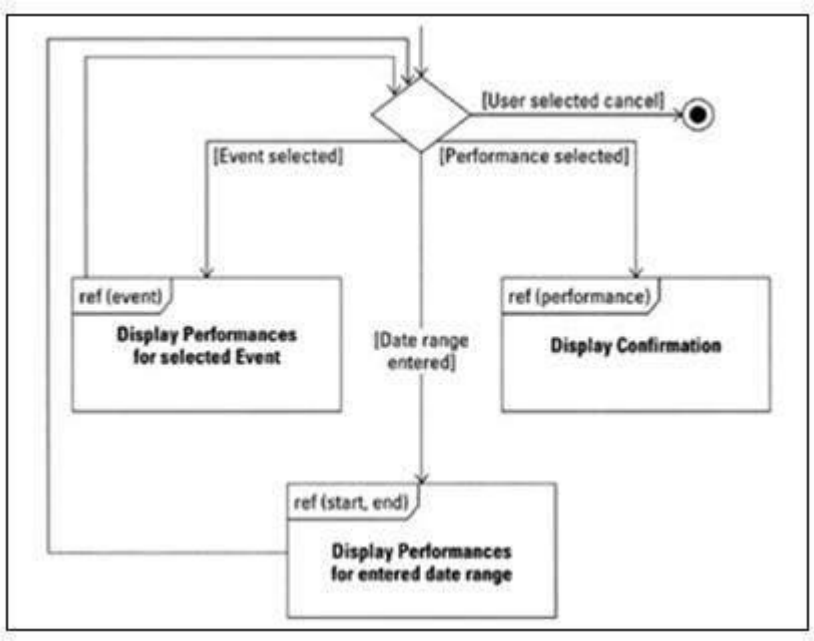
Modeling an Interaction Overview

In practical usage, a sequence diagram is used to model a single scenario. A number of sequence diagrams are used to complete the entire application. Hence, while modeling a single scenario, it is possible to forget the total process and this can introduce errors.

To solve this issue, the new interaction overview diagram combines the flow of control from an activity diagram and messaging specification from the sequence diagram.

Activity diagram uses activities and object flows to describe a process. The Interaction Overview diagram uses interactions and interaction occurrences. The lifelines and messages found in Sequence diagrams appear only within the interactions or interaction occurrences. However, the lifelines (objects) that participate in the Interaction Overview diagram may be listed along with the diagram name.

Following figure shows an interaction overview diagram with decision diamonds, frames, and termination point.



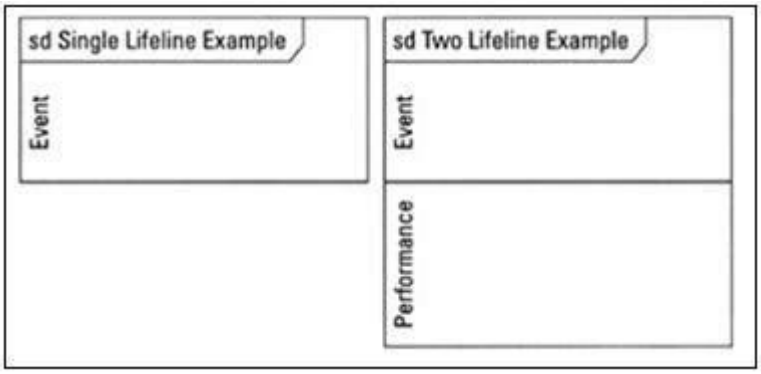
Modeling a Timing Diagram

The name of this diagram itself describes the purpose of the diagram. It basically deals with the time of the events over its entire lifecycle.

A timing diagram can therefore be defined as a special purpose interaction diagram made to focus on the events of an object in its life time. It is basically a mixture of state machine and interaction diagram. The timing diagram uses the following timelines −

- State time line
- General value time line

A lifeline in a Timing diagram forms a rectangular space within the content area of a frame. It is typically aligned horizontally to read from the left to right. Multiple lifelines may be stacked within the same frame to model the interaction between them.



Summary

UML 2.0 is an enhanced version where the new features are added to make it more usable and efficient. There are two major categories in UML 2.0, one is UML super structure and another is UML infrastructure. Although the new diagrams are based on the old concepts, they still have some additional features.

UML 2.0 offers four interaction diagrams, the Sequence diagram, Communication diagram, Interaction Overview diagram, and an optional Timing diagram. All four diagrams utilize the frame notation to enclose an interaction. The use of frames support the reuse of interactions as interaction occurrences