SOFTWARE DESIGN PHASE

A designer's goal is to produce a model or representation of an entity that will later 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. Design begins with the requirements model. Engineers work to transform this model into four levels of design detail: the data structure, the system architecture, the interface representation, and the component level detail. The final result is the Design Specification. The specification is composed of the design models that describe data, architecture, interfaces, and components. At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with the requirements and with one another.

Design phase begins once software requirements have been analyzed and specified. The design task produces a data design, an architectural design, an interface design, and a component design. 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. 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 definedwithin the analysis model.The interface design describes how the software communicates within itself, withsystems that interoperate with it, and with humans who use it. An interface impliesa flow of information (e.g., data and/or control) and a specific type of behavior. Therefore,data and control flow diagrams provide much of the information required forinterface design.The component-level design transforms structural elements of the software architectureinto a procedural description of software components. Information obtainedfrom the process specification, control specification and state transition diagram serve as the basis for component design.

Design is the phase in software engineering process in which quality is built into the system being engineered. Design provideswith the representations of software that can be assessed for quality. Design isthe only way in which customer's requirements can be accurately translated into a finishedsoftware product or system.Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs. 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 theanalysis model, and it must accommodate all of the implicit requirementsdesired by the customer.

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

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

Some guidelines for establishing technical criteria for good design are:

1. A design should exhibit an architectural structure that (a) has been createdusing recognizable design patterns, (b) is composed of components thatexhibit good design characteristics, and (c) can be implemented in an evolutionary fashion, thereby facilitatingimplementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and subfunctions.
3. A design should contain distinct representations of data, architecture, interfaces, and components (modules).
4. A design should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

The design of a system is essentially a blueprint or a plan for a solution for the system. The design process for software systems often has two levels. At the first level the focus is on deciding which modules are needed for the system, the specifications of these modules, and how the modules should be interconnected. This is what is called the system design or top-level design. In the second level, the internal design of the modules, or how the specifications of the module can be satisfied, is decided. This design level is often called detailed design or logic design. Detailed design essentially expands the system design to contain a more detailed description of the processing logic and data structures so that the design is sufficiently complete for coding. A design methodology is a systematic approach to creating a design by applying of a set of techniques and guidelines. We discuss the function-oriented methods for design and the structured design methodology in some detail. In a function-oriented design approach, a system is viewed as a transformation function, transforming the inputs to the desired outputs. The purpose of the design phase is to specify the components for this transformation function, so that each component is also a transformation function. That is, each module in design supports a functional abstraction. The basic output of the system design phase, when a function oriented design approach is being followed, is the definition of all the major data structures in the system, all the major modules of the system, and how the modules interact with each other.

I. Function-oriented design

Design principles

To evaluate a design, some criteria/thumb rules need to be specified; a design should clearly be verifiable, complete (implements all the specifications), and traceable (all design elements can be traced to some requirements). However, the two most important properties that concern designers are efficiency and simplicity. Efficiency of any system is concerned with the proper use of scarce resources by the system. The need for efficiency arises due to cost considerations. If some resources are scarce and expensive, it is desirable that those resources be used efficiently. In computer systems, the resources that are most often considered for efficiency are processor time and memory. An efficient system is one that consumes less processor time and requires less memory. In earlier days, the efficient use of CPU and memory was

important due to the high cost of hardware. Now that the hardware costs are low compared to the software costs, for many software systems traditional efficiency concerns now take a back seat compared to other considerations. One of the exceptions is real-time systems, for which there are strict execution time constraints.
Simplicity is perhaps the most important quality criteria for software systems. Maintenance of software is usually quite expensive. Maintainability of software is one of the goals we have established. The design of a system is one of the most important factors affecting the maintainability of a system. During maintenance, the first step a maintainer has to undertake is to understand the system to be maintained. Only after a maintainer has a thorough understanding of the different modules of the system, how they are interconnected, and how modifying one will affect the others should the modification be undertaken. A simple and understandable design will go a long way in making the job of the maintainer easier. The design concepts are:

A. Abstraction

An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior. Presumably, the abstract definition of a component is much simpler than the component itself. Abstraction is an indispensable part of the design process and is essential for problem partitioning. Partitioning essentially is the exercise in determining the components of a system. However, these components are not isolated from each other; they interact with each other, and the designer has to specify how a component interacts with other components. To decide how a component interacts with other components, the designer has to know, at the very least, the external behavior of other components. If the designer has to understand the details of the other components to determine their external behavior, we have defeated the purpose of partitioning—isolating a component from others. To allow the designer to concentrate on one component at a time, abstraction of other components is used. Abstraction is used for existing components as well as components that are being designed. Abstraction of existing components plays an important role in the maintenance phase. To modify a system, the first step is understanding what the system does and how. The process of comprehending an existing system involves identifying the abstractions of subsystems and components from the details of their implementations. Using these abstractions, the behavior of the entire system can be understood. This also helps determine how modifying a component affects the system. During the design process, abstractions are used in the reverse manner than in the process of understanding a system. During design, the components do not exist, and in the design the designer specifies only the abstract specifications of the different components. The basic goal of system design is to specify the modules in a system and their abstractions. Once the different modules are specified, during the detailed design the designer can concentrate on one module at a time. The task in detailed design and implementation is essentially to implement the modules so that the abstract specifications of each module are satisfied. There are two common abstraction mechanisms for software systems: functional abstraction and data abstraction. In functional abstraction, a module is specified by the function it performs. For example, a module to compute the log of a value can be abstractly represented by the function log. Similarly, a module to sort an input array can be represented by the specification of sorting. Functional abstraction is the basis of partitioningin function-oriented approaches. That is, when the problem is being partitioned, the overall transformation function for the system is partitioned into smaller functions that comprise the system function. The decomposition of the system is in terms of functional modules. The second unit

for abstraction is data abstraction. Any entity in the real world provides some services to the environment to which it belongs. Often the entities provide some fixed predefined services. The case of data entities is similar. Certain operations are required from a data object, depending on the object and the environment in which it is used. Data abstraction supports this view. Data is not treated simply as objects, but is treated as objects with some predefined operations on them. The operations defined on a data object are the only operations that can be performed on those objects. From outside an object, the internals of the object are hidden; only the operations on the object are visible. Data abstraction forms the basis for object-oriented design. In using this abstraction, a system is viewed as a set of objects providing some services. Hence, the decomposition of the system is done with respect to the objects the system contains.

[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.]

B. Problem Partitioning/ Refinement and Hierarchy

When solving a small problem, the entire problem can be tackled at once. The complexity of large problems and the limitations of human minds do not allow large problems to be treated as huge monoliths. For solving larger problems, the basic principle is the time-tested principle of "divide and conquer." Clearly, dividing in such a manner that all the divisions have to be conquered together is not the intent of this wisdom. This principle, if elaborated, would mean "divide into smaller pieces, so that each piece can be conquered separately." For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately. It is this restriction of being able to solve each part separately that makes dividing into pieces a complex task and that many methodologies for system design aim to address. The basic rationale behind this strategy is the belief that if the pieces of a problem are solvable separately, the cost of solving the entire problem is more than the sum of the cost of solving all the pieces. However, the different pieces cannot be entirely independent of each other, as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem. This communication adds complexity, which arises due to partitioning and may not have existed in the original problem. As the number of components increases, the cost of partitioning, together with the cost of this added complexity, may become more than the savings achieved by partitioning. It is at this point that no further partitioning needs to be done. The designer has to make the judgment about when to stop partitioning. As simplicity and understandability are two of the most important quality criteria for software design, it can be argued that maintenance is minimized if each part in the system can be easily related to the application and each piece can be modified separately. If a piece can be modified separately, we call it independent of other pieces. If module A is independent of module B, then we can modify A without introducing any unanticipated side effects in B. Total independence of modules of one system is not possible, but the design process should support as much independence as possible between modules. Dependence between modules in a software system is one of the reasons for high maintenance costs. Clearly, proper partitioning will make the system easier to maintain by making the design easier to understand. Problem partitioning also aids design verification. Problem partitioning, which is essential for solving a complex problem, leads to hierarchies in the design. That is, the design produced by using problem partitioning can be represented as a hierarchy of components. The relationship between the elements in this hierarchy can

vary depending on the method used. For example, the most common is the "whole-part of" relationship. In this, the system consists of some parts; each part consists of subparts, and so on. This relationship can be naturally represented as a hierarchical structure between various system parts. In general, hierarchical structure makes it much easier to comprehend a complex system. Due to this, all design methodologies aim to produce a design that employs hierarchical structures.
Stepwise refinement is a top-down design strategy. Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

C. Modularity

Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a reader. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. This leads to a "divide and conquer" conclusion—it's easier to solve a complex problem when you break it into manageable pieces. The effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grow. There are five criteria that enable evaluation of 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 system wide 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.

A system is considered modular if it consists of discreet components so that each component can be implemented separately, and a change to one component has minimal impact on other components. Modularity is a clearly a desirable property in a system. Modularity helps in system debugging—isolating the system problem to a component is easier if the system is modular; in system repair—changing a part of the system is easy as it affects few other parts; and in system building—a modular system can be easily built by "putting its modules together." A software system cannot be made modular by simply chopping it into a set of modules. For modularity, each module needs to support a well defined abstraction and have a clear interface through which it can interact with other modules. Modularity is where abstraction and partitioning come together. For easily understandable and maintainable systems,

modularity is clearly the basic objective; partitioning and abstraction can be viewed as concepts that help achieve modularity. It will be even better if the modules are also separately compilable (then changes in a module will not require recompilation of the whole system).

D. Top-Down and Bottom-Up Strategies

A system consists of components, which have components of their own; indeed a system is a hierarchy of components. The highest-level components correspond to the total system. To design such hierarchies there are two possible approaches: top-down and bottom-up. The top-down approach starts from the highest-level component of the hierarchy and proceeds through to lower levels. By contrast, a bottom-up approach starts with the lowest-level component of the hierarchy and proceeds through progressively higher levels to the top-level component.

A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved. Top-down design methods often result in some form of stepwise refinement. Starting from an abstract design, in each step the design is refined to a more concrete level, until wereach a level where no more refinement is needed and the design can be implemented directly. The top-down approach has been promulgated by many researchers and has been found to be extremely useful for design. Most design methodologies are based on the top-down approach.

A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that usethese lower-level components. Bottom-up methods work with layers ofabstraction. Starting from the very bottom, operations that provide a layerof abstraction are implemented. The operations of this layer are then used toimplement more powerful operations and a still higher layer of abstraction, until the stage is reached where the operations supported by the layer arethose desired by the system. A top-down approach is suitable only if the specifications of the systemare clearly known and the system development is from scratch. However, if asystem is to be built from an existing system, a bottom-up approach is moresuitable, as it starts from some existing components. Pure top-down or pure bottom-up approaches are often not practical. For a bottom-up approach to be successful, we must have a good notionof the top to which the design should be heading. Without a good idea about the operations needed at the higher layers, it is difficult to determine what operations the current layer should support. Top-down approachesrequire some idea about the feasibility of the components specified during design. The components specified during design should be implementable, which requires some idea about the feasibility of the lower-level parts of acomponent. A common approach to combine the two approaches is to provide a layer of abstraction for the application domain of interest throughlibraries of functions, which contains the functions of interest to the application domain. Then use a top-down approach to determine the modules in the system, assuming that the abstract machine available for implementing the system provides the operations supported by the abstraction layer. This approach is frequently used for developing systems.

Module level concepts

A. Coupling

Two modules are considered independent if one can function completelywithout the presence of other. Obviously, if two modules are independent,they are solvable and modifiable separately. However, all the modules ina system cannot be independent of

each other, as they must interact sothat together they produce the desired external behavior of the system. Themore connections between modules, the more dependent they are in the sense that more knowledge about one module is required to understand or solve the other module. Hence, the fewer and simpler the connections between modules, the easier it is to understand one without understanding the other. The notion of coupling attempts to capture this concept of "how strongly" different modules are interconnected. Coupling between modules is the strength of interconnections between modules or a measure of interdependence among modules. In general, the more we must know about module A in order to understand module B, the more closely connected A is to B. "*Highly coupled*" modules are joined by strong interconnections, while "*loosely coupled*" modules have weak interconnections. *Independent modules* have no interconnections. To solve and modify a module separately, we would like the module to be loosely coupled with other modules. The choice of modules decides the coupling between modules. Since the modules of the software system are created during system design, the coupling between modules is largely decided during system design and cannot be reduced during implementation. Coupling increases with the ***complexity and obscurity of the interface between modules***. To keep coupling low we would like to minimize the number of interfaces per module and the complexity of each interface. An interface of a module is used to pass information to and from other modules. Coupling is reduced if only the defined entry interface of a module is used by other modules (for example, passing information to and from a module exclusively through parameters). Coupling would increase if a module is used by other modules via an indirect and obscure interface, like directly using the internals of a module or using shared variables. Complexity of the interface is another factor affecting coupling. The more complex each interface is, the higher will be the degree of coupling. For example, complexity of the entry interface of a procedure depends on the number of items being passed as parameters and on the complexity of the items. Some level of complexity of interfaces is required to support the communication needed between modules. However, often more than this minimum is used. For example, if a field of a record is needed by a procedure, often the entire record is passed, rather than just passing that field of the record. By passing the record we are increasing the coupling unnecessarily. Essentially, we should keep the interface of a module as simple and small as possible. The ***type of information flow along the interfaces*** is the third major factor affecting coupling. There are two kinds of information that can flow along an interface: data or control. Passing or receiving control information means that the action of the module will depend on this control information, which makes it more difficult to understand the module and provide its abstraction. Transfer of data information means that a module passes as input some data to another module and gets in return some data as output. This allows a module to be treated as a simple input-output function that performs some transformation on the input data to produce the output data. In general, interfaces with only data communication result in the lowest degree of coupling, followed by interfaces that only transfer control data. Coupling is considered highest if the data is hybrid, that is, some data items and some control items are passed between modules.

B. Cohesion

We have seen that coupling is reduced when the relationships among elements in different modules are minimized. That is, coupling is reduced when elements in different modules have little or no bonds between them.Another way of achieving this effect is to strengthen the bond between elementsof the same module by

maximizing the relationship between elementsof the same module. With cohesion, we are interested in determining howclosely the elements of a module are related to each other. Cohesion of a module represents how tightly bound the internal elements of the module are to one another. Cohesion of a module gives the designeran idea about whether the different elements of a module belong together inthe same module. Cohesion and coupling are clearly related. Usually, thegreater the cohesion of each module in the system, the lower the coupling between modules is. This correlation is not perfect, but it has been observed in practice. There are several levels of cohesion:

- Coincidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- Functional

Coincidental is the lowest level, and functional is the highest. These levels donot form a linear scale. Functional binding is much stronger than the rest,while the first two are considered much weaker than others. Often, manylevels can be applicable when considering cohesion between two elements ofa module. In such situations, the highest level is considered. Cohesion of amodule is considered the highest level of cohesion applicable to all elementsin the module.

Coincidental cohesion occurs when there is no meaningful relationshipamong the elements of a module. Coincidental cohesion can occur if anexisting program is "modularized" by chopping it into pieces and makingdifferent pieces modules. If a module is created to save duplicate code bycombining some part of code that occurs at many different places, that moduleis likely to have coincidental cohesion. In this situation, the statementsin the module have no relationship with each other, and if one of the modulesusing the code needs to be modified and this modification includes thecommon code, it is likely that other modules using the code do not want thecode modified. Consequently, the modification of this "common module"may cause other modules to behave incorrectly. The modules using thesemodules are therefore not modifiable separately and have strong interconnectionbetween them. It is a poor practiceto create a module merely to avoid duplicate code (unless the common codehappens to perform some identifiable function, in which case the statementswill have some relationship between them) or to chop a module into smallermodules to reduce the module size.

A module has *logical cohesion* if there is some logical relationship betweenthe elements of a module, and the elements perform functions that fall in the same logical class. A typical example of this kind of cohesion is amodule that performs all the inputs or all the outputs. In such a situation, ifwe want to input or output a particular record, we have to somehow conveythis to the module. Often, this will be done by passing some kind of specialstatus flag, which will be used to determine what statements to execute inthe module. Besides resulting in hybrid information flow between modules,which is generally the worst form of coupling between modules, such a modulewin usually have tricky and clumsy code. In general, logically cohesivemodules should be avoided, if possible.

Temporal cohesion is the same as logical cohesion, except that the elementsare also related in time and are executed together. Modules thatperform activities like "initialization," "clean-up," and "termination" areusually temporally bound. Even though the elements in a temporally boundmodule are logically related, temporal

cohesion is higher than logical cohesion,because the elements are all executed together. This avoids the problemof passing the flag, and the code is usually simpler.
A *procedurally cohesive* module contains elements that belong to a commonprocedural unit. For example, a loop or a sequence of decision statementsin a module may be combined to form a separate module. Procedurallycohesive modules often occur when modular structure is determinedfrom some form of flowchart. Procedural cohesion often cuts across functionalhues. A module with only procedural cohesion may contain only partof a complete function or parts of several functions.
A module with *communicational cohesion* has elements that are relatedby a reference to the same input or output data. That is, in a communicationally - bound module, the elements are together because they operate on the same input or output data. An example of this could be a module to "print and punch record." Communicationally cohesive modules may perform more than one function. However, communicational cohesion is sufficiently high as to be generally acceptable if alternative structures with higher cohesion cannot be easily identified.
When the elements are together in a module because the output of oneforms the input to another, we get *sequential cohesion*. If we have a sequenceof elements in which the output of one forms the input to another, sequentialcohesion does not provide any guidelines on how to combine them into modules.Different possibilities exist: combine all in one module, put the firsthalf in one and the second half in another, the first third in one and the restin the other, and so forth. Consequently, a sequentially bound module may contain several functions or parts of different functions.
Sequentially cohesive modules bear a close resemblance to the problem structure. However, they are considered to be far from the ideal, which is functional cohesion. *Functional cohesion* is the strongest cohesion. In a functionally boundmodule, all the elements of the module are related to performing a singlefunction. By function, we do not mean simply mathematical functions; modulesaccomplishing a single goal are also included. Functions like "compute square root" and "sort the array" are clear examples of functionally cohesivemodules.
There is no mathematical formula that can be used. We have to use our judgment forthis. A useful technique for determining if a module has functional cohesionis to write a sentence that describes, fully and accurately, the function orpurpose of the module. The following tests can then be made:

- If the sentence must be a compound sentence, if it contains a comma, or it has more than one verb, the module is probably performing morethan one function, and it probably has sequential or communicationalcohesion.
- If the sentence contains words relating to time, like "first," "next,""when," and "after" the module probably has sequential or temporalcohesion.
- If the predicate of the sentence does not contain a single specific objectfollowing the verb (such as "edit all data") the module probably haslogical cohesion.
- Words like "initialize" and "cleanup" imply temporal cohesion.

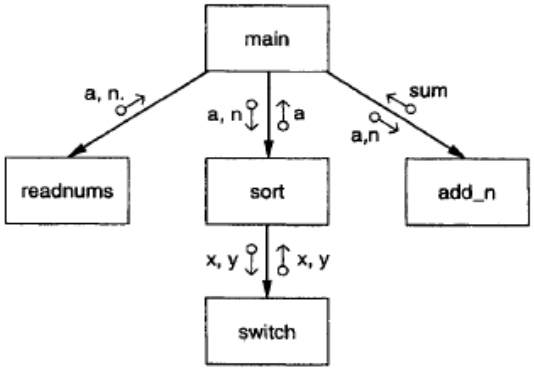
Modules with functional cohesion can always be described by a simple sentence. Functionally cohesivemodules can also be described by compound sentences.

Design Notation and Specification
During the design phase there are two things of interest: the design of thesystem, the producing of which is the basic objective of this phase, and theprocess of designing itself. It is for the latter that principles and methods areneeded. In addition, while

designing, a designer needs to record his thoughtsand decisions and to represent the design so that he can view it and playwith it. For this, design notations are used.Often, design specificationuses textual structures, with design notation helping understanding.

A. Structure Charts

For a function-oriented design, the design can be represented graphically bystructure charts. The structure of a program is made up of the modules ofthat program together with the interconnections between modules. Every computer program has a structure, and given a program its structure canbe determined. The structure chart of a program is a graphic representation of its structure. In a structure chart a module is represented by a box withthe module name written in the box. An arrow from module A to module B represents that module A invokes module B. B is called the subordinate of A, and A is called the superordinate of B. The arrow is labeled by the parametersreceived by B as input and the parameters returned by B as output, with the direction of flow of the input and output parameters represented by small arrows. The parameters can be shown to be data (unfilled circle at the tailof the label) or control (filled circle at the tail). As an example consider thestructure of a program, which enters N numbers in an array, sorts the numbers and finally returns the sum of first n numbers in the array. The structure chart for the program is shown below:



Procedural information is not represented in a structure chart,and the focus is on representing the hierarchy of modules. There are situations where the designer may wish to communicate certain procedural information explicitly, like major loops and decisions. Such information can also be represented in a structure chart. Consider asituation where module A has subordinates B, C, and D, and A repeatedly calls the modules C and D. This can be represented by a looping arrow around the arrows joining the subordinates C and D to A.

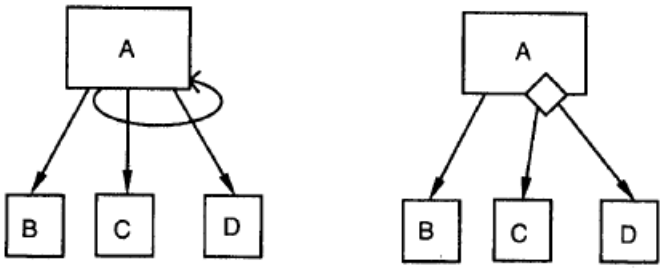
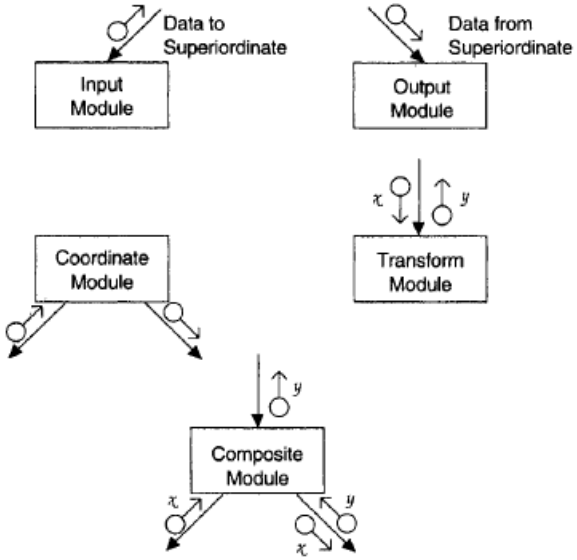


Figure : Iteration and decision representation.

All the subordinate modules activated within a common loopare enclosed in the same looping arrow.Major decisions can be represented similarly. For example, if the invocationof modules C and D in module A depends on the outcome of somedecision, that is represented by a small diamond in the box for A, with thearrows joining C and D coming out of this diamond, as shown above. Modules in a system can be categorized into few classes. There are somemodules that obtain information from their subordinates and then pass itto their superordinate. This kind of module is an input module. Similarly,there are output modules, that take information from their superordinateand pass it on to its subordinates. As the name suggests, the input andoutput modules are typically used for input and output of data from and tothe environment. The input modules get the data from the sources and getit ready to be processed, and the output modules take the output producedand prepare it for proper presentation to the environment. Then there aremodules that exist solely for the sake of transforming data into some otherform. Such a module is called a transform module. Most of the computationalmodules typically fall in this category. Finally, there are moduleswhose primary concern is managing the flow of data to and from differentsubordinates. Such modules are called coordinate modules. The structurechart representation of the different types of modules is shown in below.



A module can perform functions of more than one type of module. For e.g., the composite module above, is an input module from the point of view of its superordinate, as it feeds the data Y to the superordinate. Internally, A is a coordinate module and views its job as getting data X fromone subordinate and passing it to another subordinate, which converts it toY. Modules in actual systems are often composite modules.A structure chart is a convenient representation mechanism for a design thatuses functional abstraction. It shows the modules and their call hierarchy, the interfaces between the modules, and what information passes between modules. It is a convenient and compact notation that is very useful while creating the design. A designer can make effective use of structure charts to represent the model he is creating while he is designing. However, it is not sufficient for representing the final design, as it does not give all the information needed about the design. For example, it does not specifythe scope, structure of data, specifications of each module, etc. Hence, it is generally supplemented with textual specifications to convey design to the implementer. For a given set of requirements, many different programs can be written to satisfy the requirements, and each program can have a different structure.

So, although the structure of a given program is fixed, for a given set of requirements, programs with different structures can be obtained. The objective of the design phase using function-oriented method is to control the eventual structure of the system by fixing the structure during design.

B. Specification

Using some design rules or methodology, a conceptual design of the system can be produced in terms of a structure chart. In a structure chart, each module is represented by a box with a name. The functionality of the module is essentially communicated by the name of the box, and the interface is communicated by the data items labeling the arrows. This is alright while the designer is designing but inadequate when the design is to be communicated. To avoid these problems, a design specification should define the major data structures, modules and their specifications, and design decisions.

During system design, the major data structures for the software are identified; without these, the system modules cannot be meaningfully defined during design. In the design specification, a formal definition of these data structures should be given.

Module specification is the major part of system design specification. All modules in the system should be identified when the system design is complete, and these modules should be specified in the document. During system design only the module specification is obtained, as the internal details of the modules are defined later. To specify a module, the design document must specify (a) the interface of the module (all data items, their types, and whether they are for input and/or output), (b) the abstract behavior of the module (what the module does) by specifying the module's functionality or its input/output behavior, and (c) all other modules used by the module being specified—this information is quite useful in maintaining and understanding the design.

After a design is approved (using some verification mechanism), the modules will have to be implemented in the target language. This requires that the module "headers" for the target language first be created from the design. This translation of the design for the target language can introduce errors if it's done manually. To eliminate these translation errors, if the target language is known (as is generally the case after the requirements have been specified), it is better to have a design specification language whose module specifications can be used almost directly in programming. This minimizes the translation errors that may occur, & reduces the effort required for translating the design to programs. It also adds incentive for designers to properly specify their design, as the design is no longer a "mere" document that will be thrown away after review—it will now be used directly in coding. If, a design specification language close to C is used, then, from the design, the module headers for C can easily be created with some simple editing. To aid the comprehensibility of the design, all major design decisions made by the designers during the design process should be explained explicitly. The choices that were available and the reasons for making a particular choice should be explained. This makes a design more visible and will help in understanding the design.

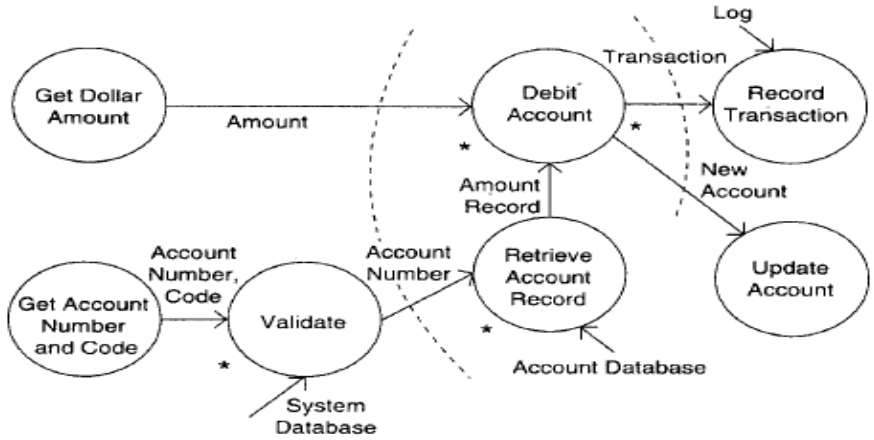
Structured Design Methodology

Creating the software system design is the major concern of the design phase. Many design techniques have been proposed over the years to provide some discipline in handling the complexity of designing large systems. The aim of design methodologies is not to reduce the process of design to a sequence of mechanical steps but to provide guidelines to aid the designer during the design process.

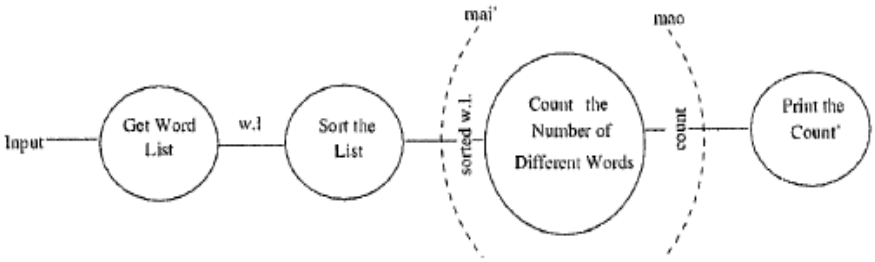
Structured design methodology (SDM) views every software system ashaving some inputs that are converted into the desired outputs by the software system. The software is viewed as a transformation function that transforms the given inputs into the desired outputs, and the central problem of designing software systems is considered to be properly designing this transformation function. Thus, the structured design methodology is primarily function-oriented and relies heavily on functional abstraction and functional decomposition. The concept of the structure of a program lies at the heart of the structureddesign method. During design, structured design methodology aims tocontrol and influence the structure of the final program. The aim is to design a system so that programs implementing the design would have a hierarchical structure, with functionally cohesive modules and as few interconnections between modules as possible. In properly designed systems it is often the case that a module with subordinates does not actually perform much computation. The bulk of actual computation is performed by its subordinates, and the module itself largely coordinates the data flow between the subordinates to get the computationdone. The subordinates in turn can get the bulk of their work done by theirsubordinates until the "atomic" modules, which have no subordinates, arereached. Factoring is the process of decomposing a module so that the bulkof its work is done by its subordinates. A system is said to be completelyfactored if all the actual processing is accomplished by bottom-level atomic modules and if non-atomic modules largely perform the jobs of control and coordination. SDM attempts to achieve a structure that is close to being completely factored. The overall strategy is to identify the input and output streams and theprimary transformations that have to be performed to produce the output. High-level modules are then created to perform these major activities, which are later refined. There are four major steps in this strategy:

1. Restate the problem as a data flow diagram
   To use the SD methodology, the first step is to construct the data flow diagram for the problem. There is a fundamental difference between the DFDs drawn during requirements analysis and those drawn during structured design. In the requirements analysis, a DFD is drawn to model the problem domain. The analyst has little control over the problem, and hence his task is to extract from the problem all the information and then represent it as a DFD. During design activity, we are dealing with the solution domain and developing a model for the eventual system. That is, the DFD during design represents how the data will flow in the system when it is built. In this modeling, the major transforms or functions in the software are decided, and the DFD shows the major transforms that the software will have and how the data will flow through different transforms. So, during drawing a DFD for design, the designer visualizes the eventual system and its processes and data flows. As the system does not yet exist, the designer has complete freedom in creating a DFD that will solve the problem stated in the SRS. The general rules of drawing a DFD remain the same; we show what transforms are needed in the software and are not concerned with the logic for implementing them. Consider the example of the simple automated teller machine that allows customers to withdraw money. A DFD for this ATM is shown below. There are two major streams of input data in this diagram. The first is the account number and the code, and the second is the amount to be debited. The DFD is self-explanatory. Notice the use of * at different places in the DFD. The transform "validate," verifies if the account number and code are valid, & needs not only the account number and code, but also information from the system database to do the validation.The transform debit account has two

outputs, one used for recordingthe transaction and the other to update the account.



Consider the problem of determining the number ofdifferent words in an input file. The data flow diagram for this problem is shown below. This problem has only one input data stream, the input file, while the desired output is the count of different words in the file. To transform the input to the desired output, the first thing we do is form a list of all the words in the file. It is best to then sort the list, as this will make identifying different words easier. This sorted list is then used to count the number of different words, and the output of this transform is the desired count, which is then printed. This sequence of data transformation is what we have in the data flow diagram.



2. Identify the input and output data elements

Most systems have some basic transformations that perform the required operations. However, in most cases the transformation cannot be easily applied to the actual physical input and produce the desired physical output. Instead, the input is first converted into a form on which the transformation can be applied with ease. Similarly, the main transformation modules often produce outputs that have to be converted into the desired physical output. The goal of this second step is to separate the transforms in the data flow diagram that convert the input or output to the desired format from the ones that perform the actual transformations. For this separation, once the data flow diagram is ready, the next step is to identify the highest abstract level of input and output. The *most abstract input* data elements are those data elements in the data flow diagram that are farthest removed from the physical inputs but can still be considered inputs to the system. The most abstract input data elements often have little resemblance to the actual physical data. These are often the data elements obtained after operations like error checking, data validation, proper formatting, and conversion are complete. Most abstract input (MAI) data elements are recognized by starting

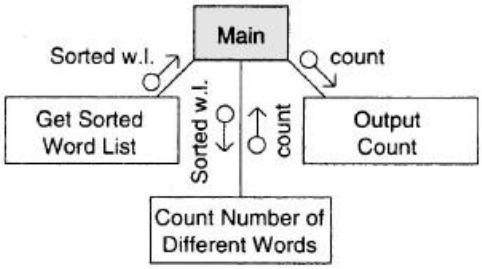
from the physical inputs and traveling toward the outputs in the data flow diagram, until the data elements are reached that can no longer be considered incoming. The aim is to go as far as possible from the physical inputs, without losing the incoming nature of the data element. This process is performed for each input stream. Identifying the most abstract data items represents a value judgment on the part of the designer, but often the choice is obvious. Similarly, we identify the *most abstract output* data elements (MAO) by starting from the outputs in the data flow diagram and traveling toward the inputs. These are the data elements that are most removed from the actual outputs but can still be considered outgoing. The MAO data elements may also be considered the logical output data items, and the transforms in the data flow diagram after these data items are basically to convert the logicaloutput into a form in which the system is required to produce the output.

There will usually be some transforms left between the most abstract input and output data items. These central transforms perform the basic transformation for the system, taking the most abstract input and transforming it into the most abstract output. The purpose of having central transforms deal with the most abstract data items is that the modules implementing these transforms can concentrate on performing the transformation without being concerned with converting the data into proper format, validating the data, and so forth. It is worth noting that if a central transform has two outputs with a + between them, it often indicates the presence of a major decision in the transform (which can be shown in the structure chart). Consider the data flow diagram shown for the word counting problem. The arcs in the data flow diagram are the most abstract input and most abstract output. The choice of the most abstract input is obvious. We start following the input. First, the input file is converted into a word list, which is essentially the input in a different form. The sorted word list is still basically the input, as it is still the same list, in a different order. This appears to be the most abstract input because the next data (i.e., count) is not just another form ofthe input data. The choice of the most abstract output is even more obvious; count is the natural choice (a data that is a form of input will not usually be a candidate for the most abstract output). So we have one central transform, count-number-of-different-words, which has one input and one output data item. Consider now the data flow diagram of the ATM. The two most abstract inputs are the dollar amount and the validated account number. The validated account number is the most abstract input, rather than the account number read in, as it is still the input—but with a guarantee that the account number is valid. The two abstract outputs are obvious. The abstract inputs and outputs are marked in the data flow diagram.

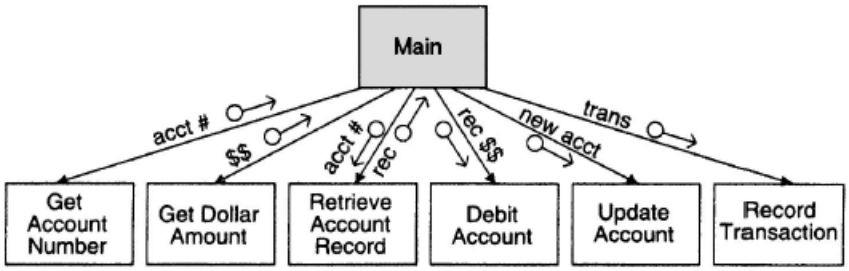
3. First-level factoring

   Having identified the central transforms and the most abstract input and output data items, we are ready to identify some modules for the system. We first specify a main module, whose purpose is to invoke the subordinates. The main module is therefore a coordinate module. For each of the most abstract input data items, an immediate subordinate module to the main module is specified. Each of these modules is an input module, whose purpose is to deliver to the main module the most abstract data item for which it is created. Similarly, for each most abstract output data item, a subordinate module that is an output module that accepts data from the main module is specified. Each of the arrows connecting these input and output subordinate modules is labeled with the

respective abstract data item flowing in the proper direction. Finally, for each central transform, a module subordinate to the main one is specified. These modules will be transform modules, whose purpose is to accept data from the main module, and then return the appropriate data back to the main module. The data items coming to a transform module from the main module are on the incoming arcs of the corresponding transform in the data flow diagram. The data items returned are on the outgoing arcs of that transform. Note that here a module is created for a transform, while input/output modules are created for data items. The structure after the first-level factoring of the word-counting problem (its data flow diagram was given earlier) is shown below.



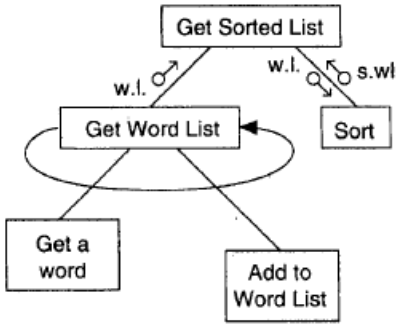
In this example, there is one input module, which returns the sorted wordlist to the main module. The output module takes from the main modulethe value of the count. There is only one central transform in this example, and a module is drawn for that. Note that the data items travelling to and from this transformation module are the same as the data items going in and out of the central transform. Let us examine the data flow diagram of the ATM. We have already seenthat this has two most abstract inputs, two most abstract outputs, and twocentral transforms. Drawing a module for each of these, we get the structure chart shown below:



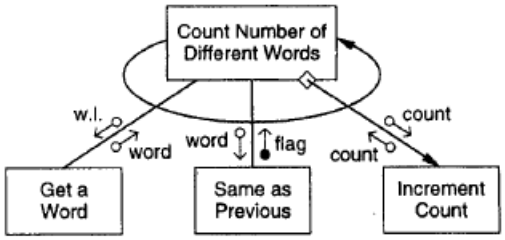
As we can see, the first-level factoring is straightforward, after the most abstract input and output data items are identified in the data flow diagram.The main module is the overall control module, which will form the main program or procedure in the implementation of the design. It is a coordinate module that invokes the input modules to get the most abstract data items, passes these to the appropriate transform modules, and delivers the resultsof the transform modules to other transform modules until the most abstract data items are obtained. These are then passed to the output modules.

4. Factoring of input, output, and transform branches
   The first-level factoring results in a very high-level structure, where each subordinate module has a lot of processing to do. To simplify these modules, they must be factored into subordinate modules that will distribute the work of a module. Each of the input, output, and transformation modules must be considered for factoring. Let us start with the input modules. The purpose of an

input module, as viewed by the main program, is to produce some data. To factor an input module, the transform in the data flow diagram that produced the data item is now treated as a central transform. The process performed for the first-level factoring is repeated here with this new central transform, with the input module being considered the main module. A subordinate input module is created for each input data stream coming into this new central transform, and a subordinate transform module is created for the new central transform. The new input modules now created can then be factored again, until the physical inputs are reached. Factoring of input modules will usually not yield any output subordinate modules. The factoring of the input module get-sorted-list in the first-level structure is shown below.



The transform producing the input returned by this module (i.e., the sort transform) is treated as a central transform. Its input is the word list. Thus, in the first factoring we have an input module to get the list and a transform module to sort the list. The input module can be factored further, as the module needs to perform two functions, getting a word and then adding it to the list. Note that the looping arrow is used to show the iteration.The factoring of the output modules is symmetrical to the factoring ofthe input modules. For an output module we look at the next transform tobe applied to the output to bring it closer to the ultimate desired output.This now becomes the central transform, and an output module is createdfor each data stream going out of this transform. During the factoring ofoutput modules, there will usually be no input modules. In our example,there is only one transform after the most abstract output, so this factoringneed not be done.If the data flow diagram of the problem is sufficiently detailed, factoringof the input and output modules is straightforward. However, there are nosuch rules for factoring the central transforms. The goal is to determine sub-transforms that will together compose the overall transform and then repeatthe process for the newly found transforms, until we reach the atomic modules.Factoring the central transform is essentially an exercise in functionaldecomposition and will depend on the designers' experience and judgment.One way to factor a transform module is to treat it as a problem inits own right and start with a data flow diagram for it. The inputs to thedata flow diagram are the data coming into the module and the outputs arethe data being returned by the module. Each transform in this data flow diagram represents a sub-transform of this transform. The central transformcan be factored by creating a subordinate transform module for each of thetransforms in this data flow diagram. This process can be repeated for thenew transform modules that are created, until we reach atomic modules.The factoring of the central transform count-the-number-of-different-wordsis shown below.



This was a relatively simple transform, and we did not need to drawthe data flow diagram. To determine the number of words, we have to geta word repeatedly, determine if it is the same as the previous word (for asorted list, this checking is sufficient to determine if the word is differentfrom other words), and then count the word if it is different. For each ofthe three different functions, we have a subordinate module, and we get the structure shown above. It should be clear that the structure that is obtained depends a good deal on what are the most abstract inputs and most abstract outputs. And as mentioned earlier, determining the most abstract inputs and outputs requires making a judgment. However, if the judgment is different, though the structure changes, it is not affected dramatically. The net effect is thata bubble that appears as a transform module at one level may appear asa transform module at another level. For example, suppose in the wordcountingproblem we make a judgment that word-list is another form of thebasic input but sorted-word-list is not. If we use word-list as the most abstractinput, the net result is that the transform module corresponding tothe sort bubble shows up as a transform module one level above. That is, now it is a central transform (i.e., subordinate to the main module) ratherthan a subordinate to the input module "get-sorted-word-list." So, the SDM has the desired property that it is not very sensitive to some variations in the identification of the most abstract input and most abstract output.

Design Heuristics

Design steps make the program structure reflect the problem as closely as possible. With this in mind the structure obtained by the methodology described earlier should be treated as an initial structure, which may need to be modified. Some heuristics can be used to modify the structure, ifnecessary.

Module size is often considered an indication of module complexity. In terms of the structure of the system, modules that are very large may not be implementing a single function and can therefore be broken into many modules, each implementing a different function. On the other hand, modules that are too small may not require any additional identity and can be combined with other modules. However, the decision to split a module or combine different modules should not be based on size alone. Cohesion and coupling of modules should be the primary guiding factors. A module should be split into separate modules only if the cohesion of the original module was low, the resulting modules have a higher degree of cohesion, and the coupling between modules does not increase. Similarly, two or more modules should be combined only if the resulting module has a high degree of cohesion and the coupling of the resulting module is not greater than the coupling of the sub modules. Furthermore, a module usually should not be split or combined with another module if it is subordinate to many different modules. As a rule of thumb, the designer should take a hard look at modules that will be larger than about 100 lines of source code or will be less than a couple of lines.

Another parameter that can be considered while "fine-tuning" the structureis the *fan-in and fan-out of modules*. Fan-in of a module is the numberof arrows coming in the

module, indicating the number of superordinates of a module. Fan-out of a module is the number of arrows going out ofthat module, indicating the number of subordinates of the module. A very high fan-out is not very desirable, as it means that the module has to control and coordinate too many modules and may therefore be too complex. Fan-out can be reduced by creating a subordinate and making many of the current subordinates subordinate to the newly created module. In general the fan-out should not be increased above five or six.Whenever possible, the fan-in should be maximized. Of course, this should not be obtained at the cost of increasing the coupling or decreasing the cohesion of modules. For example, implementing different functions into a single module, simply to increase the fan-in, is not a good idea. Fan-in can often be increased by separating out common functions from different modules and creating a module to implement that function.

Another important factor that should be considered is the *correlation of the scope of effect and scope of control*. The scope of effect of a decision (in a module) is the collection of all the modules that contain any processing that is conditional on that decision or whose invocation is dependent on the outcome of the decision. The scope of control of a module is the module itself and all its subordinates (not just the immediate subordinates). The system is usually simpler when the scope of effect of a decision is a subset of the scope of control of the module in which the decision is located. Ideally,the scope of effect should be limited to the modules that are immediate subordinates of the module in which the decision is located. Violation of this rule of thumb often results in more coupling between modules.

Verification

The output of the system design phase, like the output of other phases in thedevelopment process, should be verified before proceeding with the activitiesof the next phase. If the design is expressed in some formal notation forwhich analysis tools are available, then through tools it can be checked forinternal consistency (e.g., those modules used by another are defined, theinterface of a module is consistent with the way others use it, data usage isconsistent with declaration, etc.) If the design is not specified in a formal,executable language, it cannot be processed through tools, and other meansfor verification have to be used. The most common approach for verificationis design review or inspections. The purpose of design reviews is to ensure that the design satisfies therequirements and is of "good quality." If errors are made during the designprocess, they will ultimately reflect themselves in the code and the finalsystem. As the cost of removing faults caused by errors that occur duringdesign increases with the delay in detecting the errors, it is best if designerrors are detected early, before they manifest themselves in the system.Detecting errors in design is the purpose of design reviews.The system design review process is similar to the inspection process, in that a group of people get together to discuss the design with the aimof revealing design errors or undesirable properties. The review group mustinclude a member of both the system design team and the detailed designteam, the author of the requirements document, the author responsible formaintaining the design document, and an independent software quality engineer.As with any review, it should be kept in mind that the aim of themeeting is to uncover design errors not to try to fix them; fixing is donelater. Some of the common forms of design errors are discussed below.

Perhaps the most significant design error is *omission or misinterpretation of specified requirements*. Clearly, if the system designer has misinterpretedor not accounted for some requirement it will be reflected later as a faultin the system. Sometimes, this design error is caused by ambiguities in therequirements.

There are some other *quality factors* that are not strictly design errors but that have implications on the reliability and maintainability of the system.An example of this is weak modularity (that is, weak cohesion and/orstrong coupling). During reviews, elements of design that are not conduciveto modification and expansion or elements that fail to conform to designstandards should also be considered "errors."
The use of checklists can be extremely useful for any review. The checklist can be used by each member during private studyof the design and during the review meeting. For best results the checklist should be tailored to the project at hand, to uncover problem-specific errors. A few general items that can be used to construct a checklist fora design review are:

- Is each of the functional requirements taken into account?
- Are there analyses to demonstrate that performance requirements canbe met?
- Are all assumptions explicitly stated, and are they acceptable?
- Are there any limitations or constraints on the design beyond those inthe requirements?
- Are external specifications of each module completely specified?
- Have exceptional conditions been handled?
- Are all the data formats consistent with the requirements?
- Are the operator and user interfaces properly addressed?
- Is the design modular, and does it conform to local standards?
- Are the sizes of data structures estimated? Are provisions made toguard against overflow?

Early design work concentrated on criteria for the development of modular programsand methods for refining software structures in a top-down manner. Procedural aspects of design definition evolved into a philosophy called structured programming. Later work proposed methods for the translation of data flow or data structure into a design definition. Newer design approaches proposed an object-oriented approach to design derivation. Today, the emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures. A set of fundamental software design concepts has been explained below. Each of the concepts 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?

Software Architecture

It 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.
Certain properties should be specified as part of an architectural design, these are:

- ✓ Structural properties: This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.

For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

- ✓ Extra-functional properties: The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- ✓ Families of related systems: The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models. Structural models represent architecture as an organized collection of program components. Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications. Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. Process models focus on the design of the business or technical process that the system must accommodate. Finally, functional models can be used to represent the functional hierarchy of a system. A number of different architectural description languages (ADLs) have been developed to represent these models. ADLs provide mechanisms for describing system components and the manner in which they are connected to one another.

II. Object-oriented design

An OO model closely represents the problem domain,which makes it easier to produce and understand designs. As requirements change, the objects in a system are less immune to these changes, thereby permitting changes more easily. Inheritance and close association of objects in design to problem domain entities encourage more reuse, that is, new applications can use existing modules more effectively, thereby reducing development cost and cycle time. Object-oriented approaches are believed tobe more natural and provide richer structures for thinking and abstraction.Common design patterns have also been uncovered that allow reusability at a higher level. Object-oriented design approach is fundamentally different from the function-oriented design approaches primarily due to the different abstraction that is used. During object-oriented design the focus is on identification of the modules that the system should have, and their interfaces and relationships.

OO Analysis and OO Design

Pure object-oriented development requires that object-oriented techniques be used during the analysis, design, and implementation of the system. Much of the focus of the object-oriented approach to software developmenthas been on analysis and design. Various methods have been proposed for analysis and design, many of which propose a combined analysis and design technique. We will refer to a combined method as object-oriented analysis and design (OOAD). In OOAD the boundary between analysis and design is blurred. One reason for this is the similarity of basic constructs (i.e., objects and classes) that are used in analysis and design.The fundamental difference between object-oriented analysis (OOA) and object-oriented design (OOD) is that the former models the problem domain, leading to an understanding and specification of the problem, while the latter models the solution to the problem. In OOAD it is believed that the solution domain

representation, created by OOD, generally contains much of the representation created by OOA, and more. This is shown in figure below:

As the objective of both OOA and OOD is to model some domain, frequently the OOA and OOD processes (i.e., the methodologies) and the representations look quite similar. This contributes to the blurring of the boundariesbetween analysis and design. It is often not clear where analysis endsand design begins. The separating line is a matter of perception. The lack of clear separation between analysis and design can also be considered one of the strong points of the object-oriented approach—the transition from analysis to design is "seamless." The main difference between OOA and OOD, due to the different domains of modeling, is in the type of objects that come out of the analysis and design processes. The objects during OOA focus on the problem domain and generally represent some things or concepts in the problem. These objects are sometimes called semantic objects as they have a meaning in the problem domain. The solution domain, on the other hand, consists of semantic objects as well as other objects. During design, as the focus is on finding and defining a solution, the semantic objects identified during OOA may be refined and extended from the point of view of implementation, and other objects are added that are specific to the solution domain. The solution domain objects include interface, application & utility objects. The interface objects deal with the user interface, which is not directly a part of the problem domain but represents some aspect of the solution desired by the user. The application objects specify the control mechanisms for the proposed solution. They are driver objects that are specific to the application needs. Utility objects are those needed to support the services of the semantic objects or to implement them efficiently (e.g. queues, trees, and tables). These objects are frequently general-purpose objects and are not application-dependent.
The basic goal of the analysis and design activities is to identify the classes in the system and their relationships, and frequently represented by class diagrams. In addition, the dynamic behavior of the system has to be studied to make sure that the final design supports the desired dynamic behaviors. Due to this, some dynamic modeling of the system is desired before the design is complete. Another way to view the difference between modeling and design is that in design, a model is built for the (eventual) implementation. As a consequence, implementation issues drive the modeling process during design. The models built during object-oriented analysis form the starting point of object-oriented design, and the model built by OOD forms the basis for object-oriented implementation.

Object oriented concepts

a) Classes and Objects

Classes and objects are the basic building blocks of an OOD, just like functions(and procedures) are for a function-oriented design. During design, we are not dealing just with abstractions of real-world objects (as is the case with analysis), but we are also dealing with abstract software objects. During analysis, we viewed an object as an entity in the problem domain that had clearly defined boundaries and behavior. During design, this has to be extended to accommodate software objects.

b) Encapsulation

In general, objects entities are considered to be providing some services to be used by a client, which could be another object, program, or a user. The basic property of an object is encapsulation: it encapsulates the data and information it contains, and supports a well-defined abstraction. For this, an object provides some well-defined services its clients can use, with the additional constraint that a client can access the object only through these

services. This encapsulation of information along with the implementationof the operations performed on the information such that from outside a set of services is available is a key concept in object orientation. The set of services that can be requested from outside the object forms the interface of the object. An object may have operations defined only for internal use that cannot be used from outside. Such operations do not form part of the interface. The interface defines all ways in which an object can be used from outside.

Consider an object, **directory (**of telephone numbers) that has add-name(), change-number(), and find-number() operations as part of the interface. These are the operations that can be invoked from outsideon the object directory. It may also have internal operations like hash()and insert0 that are used to support the operations in the interface but donot form part of the interface. These operations can only be invoked fromwithin the object directory (i.e., by the operations defined on the object). A major advantage of encapsulation is that access to the encapsulated data is limited to the operations defined on the data. Hence, it becomes much easier to ensure that the integrity of data is preserved, something very hard to do if any program from outside can directly manipulate the data structures of an object. This is an extremely desirable property when building large systems, without which things can be very chaotic. In function-oriented systems, this is usually supported through self-discipline by providing access functions to some data and requiring or suggesting that other programs access the information through the access functions. In OO languages, this is enforced by the language, and no program from outside can directly access the encapsulated data. Encapsulation, leading to the separation of the interface and its implementation, has another major consequence. As long as the interface is preserved, implementation of an object can be changed without affecting any user of the object. Consider the directory object discussed earlier. Suppose the object uses an array of words to implement the operations defined on directory. Later, if the implementation is changed from the array to a B-tree or by using hashing, only the internals of the object need to be changed (i.e., the data definitions and the implementation of theoperations). From the outside, the directory object can continue to be used in the same manner as before, because its interface is not changed.

c) State, Behavior, and Identity

An object has state, behavior, and identity. The encapsulated data for an object defines the state of the object. An important property of objects is that this state persists, in contrast to the data defined in afunction or procedure, which is generally lost once the function stops being active (finishes its current execution). In an object, the state is preservedand it persists through the life of the object, i.e., unless the object is destroyed. The various components of the information an object encapsulates can be viewed as "attributes" of the object. That is, an object can be viewed as having various attributes, whose values (together with the information about the relationship of the object to the other objects) form the state of the object. The relationship between attributes and encapsulated data is that the former is in terms of concepts that may have some meaning in the problem domain: they essentially represent the abstract information being modeled by the components of the data structures. The state and services of an object together define its behavior. The behavior of an object is how an object reacts in terms of state changes when it is acted on, and how it acts upon other objects by requesting services and operations. Generally, for an object, the defined operations together specify the behavior of the object. However, it should be pointed out that although the operations specify the behavior, the actual behavior also depends on the state of the object as an operation acts on the state and the sequence of actions it performs can depend on the state. A side effect of performing an operation may be that the state of the object is modified. As operations

are the only means by which some activity can be performed by the object, it should also be clear that the current state of an object represents the sequence of operations that have been performed on it.Finally, an object has identity. Identity is the property of an object that distinguishes it from all other objects. In most programming languages, variable names are used to distinguish objects from each other. So, for example, one can declare objects s1, s2, ... of class type Stack. Each of these variables s1, s2, ... will refer to a unique stack having a state of its own (which depends on the operations performed on the stack representedby the variable).

d) Classes

Objects represent the basic run-time entities in an OO system; they occupy space in memory that keeps its state and is operated on by the defined operations on the object. A class, on the other hand, defines a possible set of objects. Objects have some attributes, whose values constitute much of the state of an object. What attributes an object has are defined by the class of the object. Similarly, the operations allowed on an object or the services it provides, are defined by the class of the object. But a class is merely a definition that does not create any objects and cannot hold any values. When objects of a class are created, memory for the objects is allocated. A class can be considered a template that specifies the properties forobjects of the class. Classes have:

1. An interface that defines which parts of an object of a class can beaccessed from outside and how
2. A class body that implements the operations in the interface
3. Instance variables that contain the state of an object of that class

Each object, when it is created, gets a private copy of the instance variables, and when an operation defined on the class is performed on the object, it is performed on the state of the particular object. A class represents a set of objects that share a common structure and a common behavior, whereas an object is an instance of a class. The interface of the objects of a class—the behavior and the state space (i.e., the states an object can take)—are all specified by the class. The class specifies the operations that can be performed on the objects of that class and the interface of each of the operations. Note that classes can be viewed as abstract data types. Abstract datatypes (ADTs) were promulgated in the 1970s, and a considerable amountof work has been done on specification and implementation of ADTs. The major differences between ADTs and class are inheritance and polymorphism. Classes without inheritance are essentially ADTs, but with inheritance, which is considered a central property of object orientation, their semantics are richer than that of an ADT. Not all operations defined on a class can be invoked on objects of that class from outside the object—some operations are defined that are entirely for internal use. The case for data declarations within the class is similar. Although generally it is fully encapsulated, in some languages it is possibleto have some data visible from outside. The distinction of what is visible from outside has to be enforced by the language. Using the C++ classification, the data and operations of a class (sometimes collectively referred to as features) can be declared as one of three types:

• Public: These are (data or operation) declarations that are accessible from outside the class to anyone who can access an object of this class.
• Protected: These are declarations that are accessible from within theclass itself and from within subclasses (actually also to those classesthat are declared as friends).
• Private: These are declarations that are accessible only from withinthe class itself (and to those classes that are declared as friends).

Different programming languages provide different access restrictions, but public and private separation is generally needed. At least one operation is needed to create (and initialize) an object and one is needed to destroy an object. The operation creating and

initializing objects is called constructor, and the operation destroying objects is called destructor. The remaining operations can be broadly divided into two categories: modifiers and value-ops. Modifiers are operations that modify the state of the object, whilevalue-ops are operations that access the object state but do not alter it. The operations defined on a class are also called methods of that class. When a client requests some operations on an object, the request is actually bound to a method defined on the class of the object. Then that method is executed, using the state of the object on which the operation is to be executed. In other words, the object itself provides the state while the class provides the actual procedure for performing the operation on the object.

e) Relationships among Objects

An object, as a stand-alone entity, has very limited capabilities—it can only provide the services defined on it. Any complex system will be composed of many objects of different classes, and these objects will interact with each other so that the overall system objectives are met. In object-oriented systems, an object interacts with another by sending a message to the object to perform some service it provides. On receiving the message, the object invokes the requested service or the method and sends the result, if needed. Frequently, the object providing the service is called the server and the object requesting the service is called the client. This form of client-server interaction is a direct fall out of encapsulation and abstraction supported by objects.

If an object invokes some services in other objects, we can say that the two objects are related in some way to each other. All objects in a system are not related to all other objects. In fact, in most programming languages, an object cannot even access all objects, but can access only those objects that have been explicitly programmed or located for this purpose. During design, which objects are related has to be clearly defined so that the system can be properly implemented.

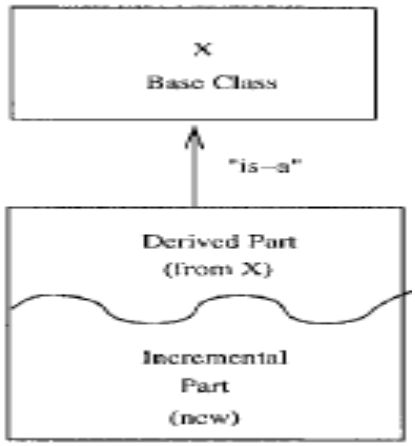
If an object uses some services of another object, there is an association between the two objects. This association is also called a link—a link exists from one object to another if the object uses some services of the other object. Links frequently show up as pointers when programming. A link captures the fact that a message is flowing from one object to another. However, when a link exists, though the message flows in the direction of the link, information can flow in both directions (e.g., the server may return someresults).

With associations comes the issue of visibility, that is, which object is visible to which. However, this is not an important issue during analysis and is therefore rarely dealt with during OOA. The basic issue here is that if there is a link from object A to object B, for A to be able to send a message to B, B must be visible to A in the final program.There are different ways to provide this visibility. Links between objects capture the client/server type of relationship. Another type of relationship between objects is aggregation, which reflects the whole/part-of relationship. Though not necessary, aggregation generally implies containment. That is, if an object A is an aggregation of objects B and C, then objects B and C will generally be within object A (though there are situations where the conceptual relationship of aggregation may not get reflected as actual containment of objects). The main implication of this is that a contained object cannot survive without its containing object. With links, that is not the case.

f) Inheritance and Polymorphism

Inheritance is a concept unique to object orientation. Some of the other concepts,such as information hiding, can be supported by non-object-oriented languages through self-discipline, but inheritance cannot generally be supported by such languages. It is also the concept central to many of the arguments claiming that software reuse can be better

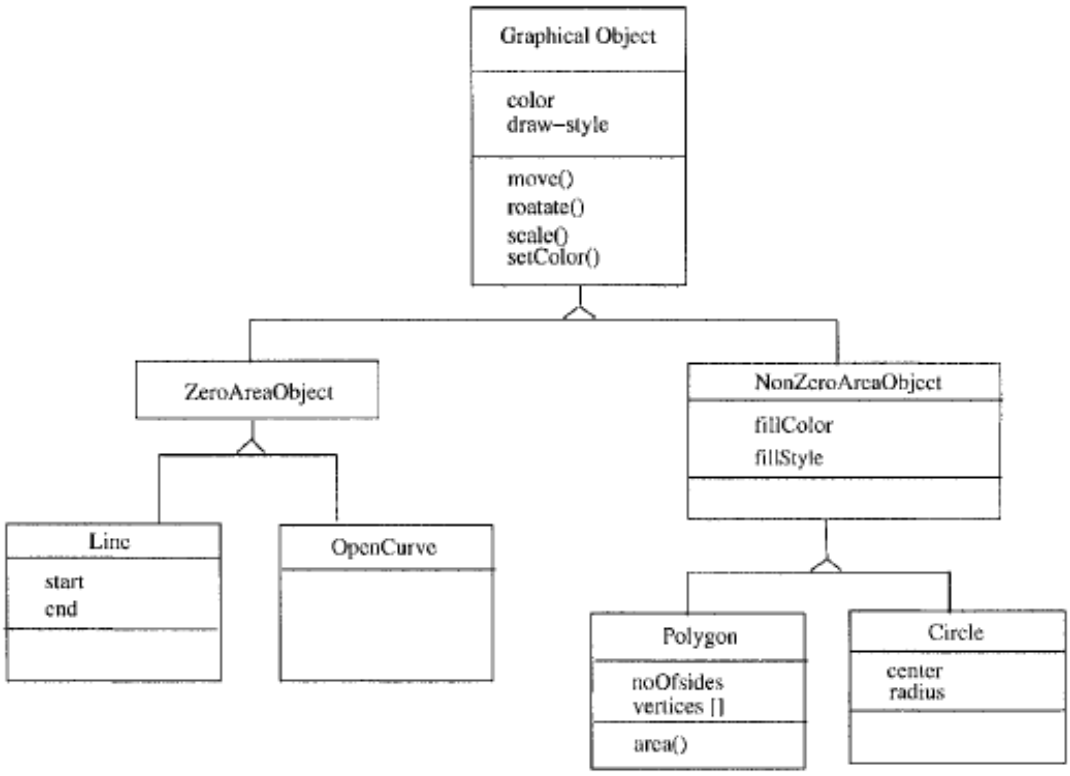
supported with object orientation. Inheritance is a relation between classes that allows for definition and implementation of one class based on the definition of existing classes.
When a class B inherits from another class A, B is referred to as the subclass or the derived class and A is referred to as the superclass or the base class. In general, a subclass B will have two parts: a derived part and an incremental part. The derived part is the part inherited from A and the incremental part is the new code and definitions that have been specifically added for B. This is shown in the figure below. Objects of type B have the derived part as well as the incremental part. Hence, by defining only the incremental part and inheriting the derived part from an existing class, we can define objects that contain both. Inheritance is often called an "is-a" relation, implying that an object of type B is also an instance of type A. That is, an instance of a subclass, though more than an instance of the superclass, is also an instance of the superclass. In general, an inherited feature of A may be redefined in various forms in B. This redefinition may change the visibility of the operation (e.g., a public operation of A may be made private in B), changed (e.g., by defining a different sequence of instructions for this operation), renamed, voided, and so on. The inheritance relation between classes forms a hierarchy. As inheritance represents an "is-a" relation, it is important that the hierarchy representa structure present in the application domain and is not created simply to reuse some parts of an existing class. That is, the hierarchy should be such that an object of a class is also an object of all its super classes in the problem domain.



Y – Derived class

The power of inheritance lies in the fact that all common features of the subclasses can be accumulated in the superclass. In other words, a feature is placed in the higher level of abstractions. Once this is done, such features can be inherited from the parent class and used in the subclass directly. This implies that if there are many abstract class definitions available, when a new class is needed, it is possible that the new class is a specialization of one or more of the existing classes. In that case, the existing class can be tailored through inheritance to define the new class. Inheritance promotes reuse by defining the common operations of the subclasses in a superclass. However, inheritance makes the subclasses dependent on the superclass, and a change in the superclass will directly affect the subclasses that inherit from it. As classes may change as design is refined, with each change in a class, its impact on the subclasses will also have to be analyzed. This also has an impact on the testing of classes. Consider a graphics package that has the class *GraphicalObject* representing all graphical objects. A graphical object can have a zero area or a non-zero area, giving two subclasses *ZeroAreaObject* and *NonZeroAreaObject*. ***Line*** and ***Curve*** are two specific object classes of the first category, and ***Polygon*** and

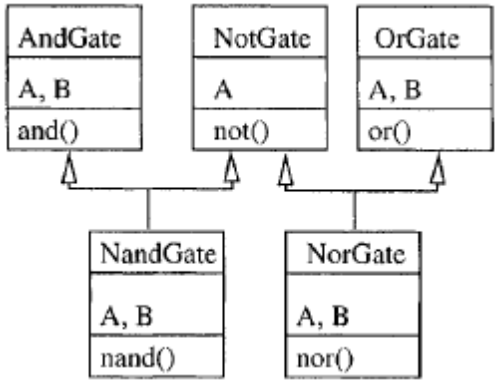
Circle are two specific object classes of the latter category. This hierarchy of classes is shown below.



the Graphical Object has attributes of color and drawstyle(which represents the style of drawing the figure)—both of which eachgraphical object has. It has many operations defined on it—***move(), rotate(), scale()***, etc.—the ones that are needed for every object by the graphics package.Note, however, that even though operations like rotate() and scale() aredefined for an object, they are totally conceptual in that their exact specificationdepends on the nature of the object (e.g., rotate() on a circle has todo different things than rotate() on a line). Hence, these operations have tobe defined for each object. In C++, such operations that are declared in asuperclass and redefined in a subclass are declared as virtual in the superclass.If an operation specified in a class is always redefined in its subclass,then the operation can be defined as *pure virtual* (in C++, this is done byequating it to 0), implying that the operation has no body. The implicationof existence of these operations is that no objects of this class can be created,as some of the operations declared in the class are not defined and hencecannot be performed. Such a class is sometimes called an *abstract base class.*

Inheritance can be broadly classified as being of two types: strict inheritance and non-strict inheritance. In *strict inheritance* a subclass takes all thefeatures from the parent class and adds additional features to specialize it.That is, all data members and operations available in the base class are also available in the derived class. This form supports the "is-a" relation and is the easiest form of inheritance. *Nonstrict inheritance* occurs when the subclass does not have all the features of the parent class or some features have been redefined. This form of inheritance has consequences in the dynamic behavior and complicates testing. A class hierarchy need not be a simple tree structure. It may be a graph,which implies that a class may inherit from multiple classes. This typeof

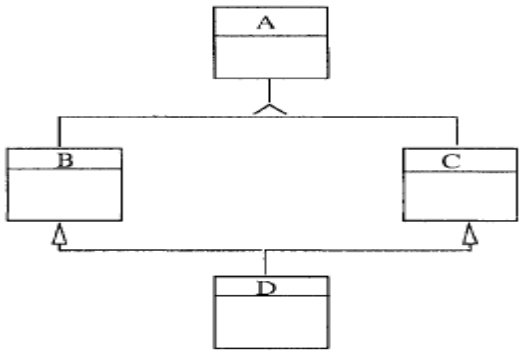
inheritance, when a subclass inherits from many superclasses, is called **multiple inheritance**. Consider part of the class hierarchy of logic gates fora system for simulating digital logic of circuits as shown in figure below. Inthis example, there are separate classes to represent And gates, Nor gates, and Or gates. The class for representing Nand gates inherits from both theclass for And gates and the class for Not gates. That is, all the definitions (instances and operations) that have been declared as public (or protected) in the classes *NotGate* and *AndGate* are available for use to the class *NandGate*. Similarly, the class *NorGate* inherits from the *OrGate* and *NotGate*. Like inregular inheritance, a subclass can redefine any feature if it desires. Multiple inheritance brings in some new issues. First, some features of two-parent classes may have the same name. So, for example, there may bean operation ***O()*** in class A and class B. If a class C inherits from class Aand class B, then when ***O()*** is invoked from an object of class C, if ***O()*** isnot defined locally within C, it is not clear from where the definition of ***O()*** should be taken—from class A or from class B. This ambiguity does not ariseif there is no multiple inheritance; the operation of the closest ancestor inwhich ***O()*** is defined is executed. Different language mechanisms or rules can be used to resolve this ambiguity. In C++, when such an ambiguity arises, the programmer has to resolve it by explicitly specifying the superclass fromwhich the definition of the feature is to be taken. Multiple inheritance also brings in the possibility of repeated inheritance, where a class inherits more than once from the same class.



For example, consider the situation shown above where classes B and C inherit from class A and class D inherits from both B and C. A situation like this means that effectively class D is inheriting twice from A- once through B and once through C. This form of inheritance is even more complex, as features of A may have been renamed in B and C, and can lead to run-time errors. Due to the complexity that comes with multiple inheritance and its variations and the possibility of confusion that comes with them, it is generally advisable to avoid their usage.

Inheritance brings in *polymorphism*, a general concept widely used in type theory that deals with the ability of an object to be of different types. In OOD, polymorphism comes in the form that a reference in an OO program can refer to objects of different types at different times. Here we are not talking about "type coercion," which is allowed in languages like C; these are features that can be avoided if desired. In object-oriented systems, with inheritance, polymorphism cannot be avoided—it must be supported. The reason is the "is-a" relation supported by inheritance—an object *x* declared to be of class B is also an object of any class A that is the superclass of B. Hence, anywhere an instance of A is expected, *x* can be used. With polymorphism, an entity has a static type and a dynamic type. The static type of an object is the type of which the object is declared in the program text, and it remains unchanged. The dynamic type of an entity, on the other

hand, can change from time to time and is known only at reference time. Once an entity is declared, at compile time the set of types that this entity belongs to can be determined from the inheritance hierarchy that has been defined. The dynamic type of the object will be one of this set, but the actual dynamic type will be defined at the time of reference of the object. In the preceding example, the static type of x is B. Initially, its dynamic type is also B. Suppose an object *y* is declared of type A, and in some sequence of instructions there is an instruction *x:= y*. Due to the "isa" relation between A and B, this is a valid statement. After this statement is executed, the dynamic type of *x* will change to A (though its static type remains B). This type of polymorphism is called *object polymorphism*, in which wherever an object of a superclass can be used, objects of subclasses can be used.



This type of polymorphism requires *dynamic binding* of operations, which brings in *feature polymorphism.* Dynamic binding means that the code associated with a given procedure call is not known until the moment of the call. Suppose x is a polymorphic reference whose static type is B but whose dynamic type could be either A or B. Suppose that an operation O is defined in the class A, which is redefined in the class B. Now when the operation O is invoked on x, it is not known statically what code will be executed. That is, the code to be executed for the statement ***x.O*** is decided at run time, depending on the dynamic type of *x*—if the dynamic type is A, the code for the operation O in class *A* will be executed; if the dynamic type is B, the code for operation O in class B will be executed. This dynamic binding can be used quite effectively during application development to reduce the size of the code. For e.g., take the case of the graphical object hierarchy discussed earlier. In an application, suppose the elements of a figure are stored in an array A (of ***GraphicalObject*** type). Suppose element 1 of this array is a line, element2 is a circle, and so on. Now if we want to rotate each object in the figure,we simply loop over the array performing A[i].rotate(). For each A[i],the appropriate rotate function will be executed. That is, which function ***A[i].rotate()*** refers to is decided at run time, depending on the dynamic type of object A[i]. This feature polymorphism, which is essentially overloading of the feature (i.e., a feature can mean different things in different contexts and its exact meaning is determined only at run time) causes no problem in strict inheritance because all features of a superclass are available in the subclasses. But in nonstrict inheritance, it can cause problems, because a child may lose a feature. Because the binding of the feature is determined at run time, this can cause a run-time error as a situation may arise where the object is bound to the superclass in which the feature is not present.

Design Concepts

In an OO system, the basic module is a class, and during design the key activity is to identify and specify the modules that should be there in the system being built. The three concepts in an OO system are cohesion, coupling, and open-closed principle. Our goal is

to create a design in which the modules are lowin coupling, high in cohesion, and which satisfy the open-closed principle.

Coupling

As mentioned earlier, coupling is an inter-module concept which captures the strength of interconnection between modules. The more tightly coupled the modules are, the more dependent they are on each other, and the more difficult it is to understand and modify them. Low coupling is desirable for making the system more understandable and modifiable. The degree of coupling between a module and another module depends on how much information is needed about the other module for understandingand modifying this module, and how complex and explicit this information is. Low coupling occurs when this information is as little as possible, as simple as possible, and is easily visible or identifiable. This concept has been discussed for systems with functional modules. Although the concept remains the same, its manifestation in OO systems is somewhat different as objects are semantically richer than functions. In OO systems, three different types of coupling exist between modules.

• *Interaction coupling*
• *Component coupling*
• *Inheritance coupling*

Interaction coupling occurs due to methods of a class invoking methods of other classes. Note that as we are looking at coupling between classes we focus on interaction between classes, and not within a class. In many ways, this situation is similar to a function calling another function and hence this coupling is similar to coupling between functional modules. Like with functions, the worst form of coupling here is if methods directly access internal parts of other methods. (This type of interaction is disallowed in many languages but is allowed where concepts like friend classes, which allow a friend to delve into the internals of a class, exist.) Interaction coupling reduces, though is still very high, if methods of aclass interact with methods in another class by directly manipulating instance variables or attributes of objects of the other class. This form of interaction is also bad as one has to understand the code of other classes to understand what changes they are making to the class. It also violates the encapsulation principle of OO. This form of interaction is worse if variables are used to communicate temporary data, that is, the variables are used not to hold the state of the object but to pass state of the computation from one object to another. If this temp-value holder variable happens to be in the super class, then the coupling worsens since the variable is visible to all subclasses. Coupling is least (like in coupling with functional modules) if methods communicate directly through parameters. Within this category, coupling is lower if only data is passed, but is higher if control information is passed since the invoked method impacts the execution sequence in the calling method. Also, coupling is higher if the amount of data being passed is more. So, if whole data structures are passed when only some parts are needed, coupling is being unnecessarily increased. Similarly, if an object is passed to a method when only some of its component objects (or objects the passed object refers to) are used within the method, coupling increases unnecessarily. The least coupling situation therefore is when communication is with parameters only,with only necessary variables being passed, and these parameters only pass data.

Component coupling refers to the interaction between two classes where a class has variables of the other class. Three clear situations exist when this can happen. A class C can be component coupled with another class C, if C has an instance variable of type C, or C has a method whose parameter is of type C, or if C has a method which has a local variable of type C (which can then be passed as parameter to some method it invokes.) Note that when C is component coupled with C, it has the potential of being component coupled with all subclasses of C as at runtime an object of any subclass may actually be used. It should be

clear that whenever there is component coupling, there is likely to be interaction coupling. Component coupling is considered to be weakest (i.e., most desired) if in a class C, the variables of class C are either in the signatures of the methods of C, or some attributes are of type C. If interaction is through local variables, then this interaction is not visible from outside, and therefore increases coupling.

Inheritance coupling is due to the inheritance relationship between classes.Two classes are considered inheritance coupled if one class is a direct or indirect subclass of the other. If inheritance adds coupling, why not do away with inheritance altogether. The reason is that inheritance may reduce the overall coupling in the system. Let us consider two situations. If a class A is coupled with another class B, and if B is a hierarchy with B and B as two subclasses, then if a method m is factored out of B and B and put in the super class B, the coupling reduces as A is now only coupled with B, whereas earlier it was coupled with both B and B. Similarly, if B is a class hierarchy which supports specialization-generalization relationship, then if new subclasses are added to B, no changes need to be made to a class A which calls methods in B. That is, for changing B's hierarchy, A need not be disturbed. Without this hierarchy, changes in B would most likely result in changes in A. Within inheritance coupling there are some situations that are worse than others. The worst form is when a subclass B modifies the signature ofa method in B (or deletes the method). This situation can easily lead to a run-time error, besides violating the true spirit of the is-a relationship. If the signature is preserved but the implementation of a method is changed, that also violates the is-a relationship, though may not lead to a run-time error, and should be avoided. The least coupling scenario is when a subclass only adds instance variables and methods but does not modify any inherited ones.

Cohesion

Whereas coupling is an inter-module concept, cohesion is an intra-module concept. It focuses on why elements of a module are together in the same module. The objective here is to have elements that are tightly related to belong to the same module. This will make the modules easier to understand, and as they capture clear concepts and abstractions, easier to modify. Generally, higher cohesion will lead to lower coupling as many elements that need to interact a lot will reside together in strongly coupled modules, lessening the need for interaction with other modules. On the other hand, modules that have low cohesion will often need to interact with other modules to perform their task. Clearly, for making a system more understandable and modifiable, we would like it to consist of modules that are highly cohesive. In other words, the goal is to have a high degree of cohesion in the modulesin the system. Cohesion in OO systems also has three aspects:

- Method cohesion
- Class cohesion
- Inheritance cohesion

Method cohesion is same as cohesion in functional modules. It focuses on why the different code elements of a method are together within the method. The highest form of cohesion is if each method implements a clearly defined function, and all statements in the method contribute to implementing this function. In general, with functionally cohesive methods, what the method does can be stated easily with a simple statement. That is, in a short and simple statement of the type "this method does...," we can express the functionality of the method.

Class cohesion focuses on why different attributes and methods are together in this class. The goal is to have a class that implements a single concept or abstraction with all elements contributing towards supporting this concept. In general, whenever there are multiple concepts encapsulated within a class, the cohesion of the class is not as high as it could be, and a designer should try to change the design to have each class encapsulate a single

concept. One symptom of the situation where a class has multiple abstractions is that the set of methods can be partitioned into two (or more) groups, each accessing a distinct subset of the attributes. That is, the set of methods and attributes can be partitioned into separate groups, each encapsulating a different concept. Clearly, in such a situation, by having separate classes encapsulating separate concepts, we can have modules with improved cohesion. In many situations, even though two (or more) concepts may be encapsulated within a class, there are some methods that access attributes of both the encapsulated concepts. This happens, when the class represents different entities which have a relationship between them. For cohesion, itis best to represent them as two separate classes with relationship among them. That is, we should have multiple classes, with some methods in these classes accessing objects of the other class. In a way, this improvement in cohesion results in an increased coupling. However, for modifiability and understandability, it is better if each class encapsulates a single concept.

Inheritance cohesion focuses on why classes are together in a hierarchy.The two main reasons for inheritance are to model generalization-specialization relationship, and for code reuse. Cohesion is considered high if the hierarchy supports generalization-specialization of some concept (which is likely to naturally lead to reuse of some code). It is considered lower if the hierarchy is primarily for sharing code with weak conceptual relationship between superclass and subclasses. In other words, it is desired that in an OO system the class hierarchies should be such that they support clearly identified generalization-specialization relationship.

The Open-Closed Principle

This is a design concept which came into existence in the OO context. Like with cohesion and coupling, the basic goal here is again to promote building of systems that are easily modifiable, as modification and change happen frequently and a design that cannot easily accommodate change will result in systems that will die fast and will not be able easily adapt to the changing world.

The basic principle, here is that Software entities should be open for extension, but closed for modification. A module being "open for extension" means that its behaviour can be extended to accommodate new demands placed on this module due to changes in requirements and system functionality. The modules being "closed for modification" means that the existing source code of the module is not changed when making enhancements. Then how does one make enhancements to a module without changing the existing source code? This principle restricts the changes to modules to extension only, i.e., it allows addition of code, but disallows changing of existing code. If this can be done, clearly, the value is tremendous. Code changes involve heavy risk and to ensure that a change has not "broken" things that were working often requires a lot of regression testing. This risk can be minimized if no changes are made to existing code. But if changes are not made, how will enhancements be made? This principle says that enhancements should be made by adding new code, rather than altering old code.

There is another side benefit of this. Programmers typically prefer writing new code rather than modifying old code. But the reality is that systems that are being built today are being built on top of existing software. If this principle is satisfied, then we can expand existing systems by mostly adding new code to old systems, and minimizing the need for changing code. This principle can be satisfied in OO designs by properly using inheritance and polymorphism. Inheritance allows creating new classes that will extend the behaviour of existing classes without changing the original class. And it is this property that can be used to support this principle. As an example consider an application in which a client object (of type Client) interacts with a printer object (of class Printer 1) and invokes the necessary methods for completing its printing needs. The class diagram for this is shown below.

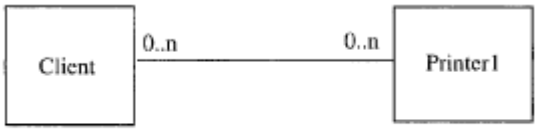


Figure : Example without using subtyping.

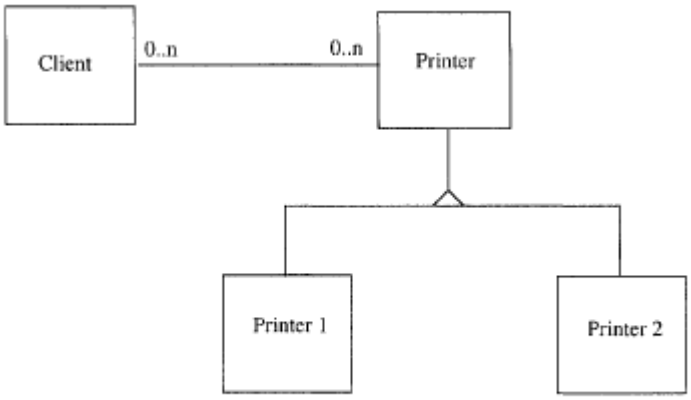


Figure : Example using subtyping.

In this design, the client directly calls the methods on the printer object for printing something. Now suppose the system has to be enhanced to allow another printer to be used by the client. Under this design, to implement this change, a new class Printer2 will have to be created and the code of the client class will have to be changed to allow using object of Printer2 type as well. This design does not support the open-closed principle as the Client class is not closed against change.The design for this system, however, can be done in another manner that supports the open-closed principle. In this design, instead of directly implementing the Printer 1 class, we create an abstract class Printer that defines the interface of a printer and specifies all the methods a printer object should support. Printer 1 is implemented as a specialization of this class. In this design, when Printer2 is to be added, it is added as another subclass of type Printer. The client does not need to be aware of this subtype as it interacts with objects of type Printer. That is, the client only deals with a generic Printer, and its interaction is same whether the object is actually of type Printer 1 or Printer2. The class diagram for this is shown in the figure which uses subtyping. It is this inheritance property of OO that is leveraged to support the open-closed principle. The basic idea is to have a class encapsulate the abstraction of some concept. If this abstraction is to be extended, the extension is done by creating new subclasses of the abstraction, thereby keeping all the existing code unchanged. If inheritance hierarchies are built in this manner, they are said to satisfy the Liskov Substitution Principle. According to this principle, if aprogram is using object Ol of a (base) class C, that program should remain unchanged if O1 is replaced by an object O2 of a class C, where C is a subclass of C. If this principle is satisfied for class hierarchies, and hierarchies are used properly, then the open-closed principle can be supported. It should also be noted that recommendations for both inheritance coupling and inheritance cohesion support that this principle be followed in class hierarchies.

III. UML Diagrams

See notes for UML diagrams.

IV. Detailed Design

So far two different approaches for system design have been discussed. In system design we concentrate on the modules in a system and how they interact with each other. Once a module is precisely specified, the internal logic that will implement the given specifications can be decided, and is the focus of detailed design. Process Design Language (PDL) is one way in which the design can be communicated precisely and completely to whatever degree of detail desired by the designer. That is, it can be used to specify the system design and to extend it to include the logic design. PDL is particularly useful when using top-down refinement techniques to design a system or module.

PDL/ Structured English

One way to communicate a design is to specify it in a natural language, like English. This approach often leads to misunderstanding, and such imprecise communication is not particularly useful when converting the design into code. The other extreme is to communicate it precisely in a formal language, like a programming language. Such representations often have great detail, which is necessary for implementation but not important for communicating the design. These details are often a hindrance to easy communication of the basic design. Ideally we would like to express the design in a language that is as precise and unambiguous as possible without having too much detail and that can be easily converted into an implementation. This is what PDL attempts to do.

```
minmax(infile)

ARRAY a

      DO UNTIL end of input
            READ an item into a
      ENDDO
      max, min := first item of a
      DO FOR each item in a
            IF max < item THEN set max to item
            IF min > item THEN set min to item
      ENDDO
END
```

PDL has an overall outer syntax of a structured programming language and has a vocabulary of a natural language (English in our case). It can be thought of as "structured English." Because the structure of a design expressed in PDL is formal, using the formal language constructs, some amount of automated processing can be done on such designs. As an example, consider the problem of finding the minimum and maximum of a set of numbers in a file and outputting these numbers in PDL as shown in figure above. Notice that in the PDL program we have the entire logic of the procedure, but little about the details of implementation in a particular language. To implement this in a language, each of the PDL statements will have to be converted into programming language statements. With PDL, a design can be expressed in whatever level of detail that is suitable for the problem. One way to use PDL is to first generate a rough outline of the entire solution at a given level of detail. When the design is agreed on at this level, more detail can be

added. This allows a successive refinement approach, and can save considerable cost by detecting the design errors early during the design phase. It also aids design verification by phases, which helps in developing error-free designs. The structured outer syntax of PDL also encourages the use of structured language constructs while implementing the design.

The basic constructs of PDL are similar to those of a structured language. The first is the IF construct. It is similar to the if-then-else construct of Pascal. However, the conditions and the statements to be executed need not be stated in a formal language. For a general selection, there is a CASE statement. Some examples of CASE statements are:

CASE OF transaction type
CASE OF operator type

The DO construct is used to indicate repetition. The construct is indicated by:

DO iteration criteria
    one or more statements
ENDDO

The iteration criteria can be chosen to suit the problem, and unlike a formal programming language, they need not be formally stated. Examples of valid uses are:

DO WHILE there are characters in input file
DO UNTIL the end of file is reached
DO FOR each item in the list EXCEPT when item is zero

A variety of data structures can be defined and used in PDL such as lists, tables, scalar, and integers. Variations of PDL, along with some automated support, are used extensively for communicating designs.

Logic/Algorithm Design

The basic goal in detailed design is to specify the logic for the different modules that have been specified during system design. Specifying the logic will require developing an algorithm that will implement the given specifications. Essentially, an algorithm is a sequence of steps that need to be performed to solve a given problem. There are a number of steps that one has to perform while developing an algorithm. The starting step in the design of algorithms is *statement of the problem*. The problem for which an algorithm is being devised has to be precisely and clearly stated and properly understood by the person responsible for *designing the algorithm*. The next step is the design of the algorithm. During this step the data structure and program structure are decided. Once the algorithm is designed, its correctness should be verified. The most common method for designing algorithms or the logic for a module is to use the *stepwise refinement* technique. The stepwise refinement technique breaks the logic design problem into a series of steps, so that the development can be done gradually. The process starts by converting the specifications of the module into an abstract description of an algorithm containing a few abstract statements. In each step, one or several statements in the algorithm developed so far are decomposed into more detailed instructions. The successive refinement terminates when all instructions are sufficiently precise that they can easily be converted into programming language statements. During refinement, both data and instructions have to be refined. The stepwise refinement technique is a top-down method for developing detailed design.

State Modeling of Classes

For object-oriented design, the approach just discussed for obtaining the detailed design may not be sufficient, as it focuses on specifying the logic or the algorithm for the modules identified in the (function-oriented) high-level design. But a class is not a functional abstraction and cannot be viewed as an algorithm. An object of a class has

some state and many operations on it. To better understand a class, the relationship between the state and various operations and the effect of interaction of various operations have to be understood. This can be viewed as one of the objectives of the detailed design activity for object-oriented development. A method to understand the behavior of a class is to view it as a finite state automata (FSA). An FSA consists of states and transitions between states, which take place when some events occur. When modeling an object, the state is the value of its attributes, and an event is the performing of an operation on the object. A state diagram relates events and states by showing how the state changes when an event is performed. A state diagram for an object will generally have an initial state, from which all states in the FSA are reachable (i.e., there is a path from the initial state to all other states).

A state diagram for an object does not represent all the actual states of the object, as there are many possible states. A state diagram attempts to represent only the logical states of the object. A logical state of an object is a combination of all those states from which the behavior of the object is similar for all possible events. The finite state modeling of objects is an aid to understand the effect of various operations defined on the class on the state of the object. A good understanding of this can aid in developing the logic for each of the operations.

Decision Trees & Decision Tables

One of the major challenges for a Systems Analyst is to effectively communicate the system requirements to a diverse audience. Quite often this entails taking facts harvested from stakeholders and presenting them in a readable form, with enough detail to facilitate testing and other downstream activities.

Supplemental specifications and use cases, along with their supporting artifacts (data definition, messages, UI prototype, etc.) can be an effective way to communicate the non-functional and functional requirements. Also, Structured English can be used to describe business rules. However, when business rules get complicated, Structured English begins to break down. In its place, the Analyst can describe complicated business logic by way of decision tables and/or decision trees.

Decision tables provide a useful way of viewing and managing large sets of similar business rules. Decision tables are composed of rows and columns. Each of the columns defines the conditions and actions of the rules. We can look at decision tables as a list of causes (conditions) and effects (actions) in a matrix, where each column represents a unique rule. The purpose is - as commonly used - to structure the logic for a specific specification feature.

A Decision Table can be constructed from the following steps:

1) Identify the Conditionals (Purchase Amount, Number of Items, etc.) and put each of them in a separate row in the leftmost column.
2) Identify the Actions (Shipping Charge) and put each of them in the leftmost column beneath the Conditionals.
3) Identify the Rules and match them to the Conditionals. Each rule is given a separate column.
4) If necessary, reorder the table so that the Conditionals with the fewest rules are above the Conditionals with more rules.
5) Fill in the Shipping Charges.
6) Label the table. A goof heuristic is to match the name of the table to the action (s)

An Example:

Consider the following example[1], which describes the calculation of shipping charges.
Acme Outdoor World has just announced that it will offer free shipping for all orders over $250. Shipping charges for all other orders will be prorated. For orders less than $250.00, the shipping charge will be calculated as follows:
If the number of items is 3 or less:

Delivery Day	Shipping Charge
Next Day	$35.00
2nd day	$15.00
Standard	$10.00

If the number of items is 4 or more:

Delivery Day	Shipping Charge, N = number of items
Next Day	N * $7.50
2nd day	N * $3.50
Standard	N * $2.50

For orders over $250.00, the shipping charge will be calculated as follows:
If the number of items is 3 or less:

Delivery Day	Shipping Charge
Next Day	$25.00
2nd day	$10.00
Standard	N * $1.50

If the number of items is 4 or more:

Delivery Day	Shipping Charge, N = number of items
Next Day	N * $6.00
2nd day	N *$2.50
Standard	FREE

Structured English Equivalent:

```
If purchase amt. > $250.00
        If number of items < 4 then
                If delivery date is next day then
                        Delivery charge = $25.00
                Else if delivery date is 2nd day then
                        Delivery charge is $10.00
                Else if delivery date is standard then
                        Delivery charge is $1.50 per item.
                End If
        Else
                If delivery date is next day then
                        Delivery charge is $6.00 per item
                Else if delivery date is 2nd day then
                        Delivery charge is $2.50 per item
                Else
                        Delivery is free of charge
                End If
        End If
Else
        If number of items < 4 then
                If delivery date is next day then
```

```
                Delivery charge = $35.00
            Else if delivery date is 2nd day then
                Delivery charge is $15.00
            Else if delivery date is standard then
                Delivery charge is $10.00
            End If
        Else
            If delivery date is next day then
                Delivery charge is $7.5 per item
            Else if delivery date is 2nd day then
                Delivery charge is $3.50 per item
            Else
                Delivery is $2.50 per item
            End If
End If
```

Decision Table Equivalent:

Purchase Amount	Over $250.00						Less Than $250.00					
Number of Items	3 or less			4 or more			3 or less			4 or more		
Delivery Day	Next	2nd day	Std.	Next	2nd day	Std.	Next	2nd day	Std.	Next	2nd day	Std.
Shipping Charge ($)	25	10	N * $1.50	N * $6.00	N * $2.50	FREE	35	15	10	N * $7.50	N * $3.50	N * $2.50

Table: Calculating Shipping Charges

Decision tables can be used to generate test cases. One simple logical way is to treat every rule as a test case, and cover them (at least once) in the test cases. In general, the condition values used in a unique way in each rule (causes) will serve as the different inputs for the test cases, and the values (effects) used in the Actions will serve as the expected results.

Regardless of the complexity of a decision or the sophistication of the technique used to analyze it, all decision makers are faced with alternatives and "states of nature." The following notation will be used in this module:

1) Terms:
   a) *Alternative*—a course of action or strategy that may be chosen by a decision maker (for example, not carrying an umbrella tomorrow).
   b) *State of nature*—an occurrence or a situation over which the decision maker has little or no control (for example, tomorrow's weather).
2) Symbols used in a decision tree:
   a) □ decision node from which one of several alternatives may be selected.
   b) ○ a state-of-nature node out of which one state of nature will occur.

To present a manager's decision alternatives, we can develop decision trees using the above symbols. When constructing a decision tree, we must be sure that all alternatives and states of nature are in their correct and logical places and that we include all possible alternatives and states of nature.

V. Verification

There are a few techniques available to verify that the detailed design is consistent with the system design. The focus of verification in the detailed design phase is on showing that the detailed design meets the specifications laid down in the system design. Validating that the system as designed is consistent with the requirements of the system is not stressed during detailed design. The three verification methods we consider are design walkthroughs, critical design review, and consistency checkers.

Design Walkthroughs

A design walkthrough is a manual method of verification. The definition and use of walkthroughs change from organization to organization. A design walkthrough is done in an informal meeting called by the designer or the leader of the designer's group. The walkthrough group is usually small and contains, along with the designer, the group leader and/or another designer of the group. The designer might just get together with a colleague for the walkthrough or the group leader might require the designer to have the walkthrough with him. In a walkthrough the designer explains the logic step by step, and the members of the group ask questions, point out possible errors or seek clarification. A beneficial side effect of walkthroughs is that in the process of articulating and explaining the design in detail, the designer himself can uncover some of the errors. Walkthroughs are essentially a form of peer review. Due to its informal nature, they are usually not as effective as the design review.

Critical Design Review

The purpose of critical design review is to ensure that the detailed design satisfies the specifications laid down during system design. The critical design review process is same as the inspections process in which a group of people get together to discuss the design with the aim of revealing design errors or undesirable properties. The review group includes, besides the author of the detailed design, a member of the system design team, the programmer responsible for ultimately coding the module(s) under review, and an independent software quality engineer. While doing design review it should be kept in mind that the aim is to uncover design errors, not try to fix them. Fixing is done later. The use of checklists, as with other reviews, is considered important for the success of the review. The checklist is a means of focusing the discussion or the "search" of errors. Checklists can be used by each member during private study of the design and during the review meeting. For best results, the checklist should be tailored to the project at hand, to uncover project specific errors.

A Sample Checklist:

- Does each of the modules in the system design exist in detailed design?
- Are there analyses to demonstrate that the performance requirements can be met?
- Are all the assumptions explicitly stated, and are they acceptable?
- Are all relevant aspects of system design reflected in detailed design?
- Have the exceptional conditions been handled?
- Are all the data formats consistent with the system design?
- Is the design structured, and does it conform to local standards?
- Are the sizes of data structures estimated? Are provisions made to guard against overflow?

- Is each statement specified in natural language easily code-able?
- Are the loop termination conditions properly specified?
- Are the conditions in the loops OK?
- Are the conditions in the if statements correct?
- Is the nesting proper?
- Is the module logic too complex?
- Are the modules highly cohesive?

Consistency Checkers

Consistency checkers are essentially compilers that take as input the design specified in a design language (PDL in our case). Clearly, they cannot produce executable code because the inner syntax of PDL allows natural language and many activities are specified in the natural language. However, the module interface specifications (which belong to outer syntax) are specified formally. A consistency checker can ensure that any modules invoked or used by a given module actually exist in the design and that the interface used by the caller is consistent with the interface definition of the called module. It can also check if the used global data items are indeed defined globally in the design. These tools can be used to compute the complexity of modules and other metrics, because these metrics are based on alternate and loop constructs, which have a formal syntax in PDL. The trade-off here is that the more formal the design language, the more checking can be done during design, but the cost is that the design language becomes less flexible and tends towards a programming language.

Metrics

Many of the metrics that are traditionally associated with code can be used effectively after detailed design. During detailed design ah the metrics covered during the system design are applicable and useful. With the logic of modules available after detailed design, it is meaningful to talk about the complexity of a module. Traditionally, complexity metrics are applied to code, but they can easily be applied to detailed design as well.

Cyclomatic Complexity

Based on the capability of the human mind and the experience of people, it is generally recognized that conditions and control statements add complexity to a program. Given two programs with the same size, the program with the larger number of decision statements is likely to be more complex.

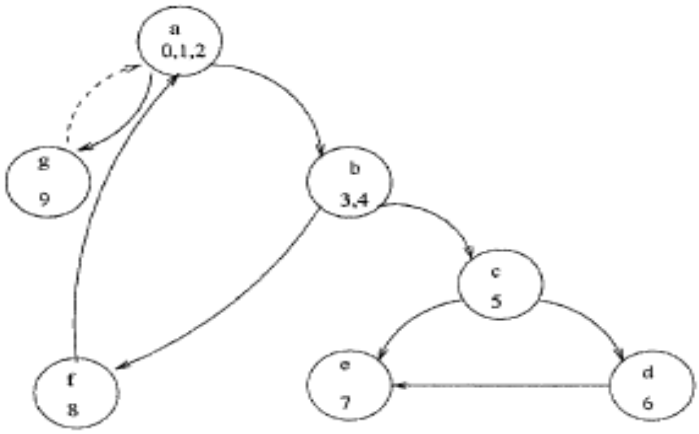
The simplest measure of complexity, then, is the number of constructs that represent branches in the control flow of the program, like if then else, while do, repeat until, and goto statements.

A more refined measure is the cyclomatic complexity measure proposed by McCabe, which is a graph-theoretic-based concept. For a graph G with n nodes, e edges, and p connected components, the cyclomatic number V (G) is defined as:

$$V(G) = e - n + p.$$

To use this to define the cyclomatic complexity of a module, the control flow graph G of the module is first drawn. To construct a control flow graph of a program module, break the module into blocks delimited by statements that affect the control flow, like **if**, **while**, **repeat**, and **goto**. These blocks form the nodes of the graph. If the control from a block **i** can branch to a block **j**, then draw an arc from node **i** to node **j** in the graph. The control flow of a program can be constructed mechanically. As an example, consider the C-like function for bubble sorting, given:

```
0. {
1.      i = 1;
2.      while (i <= n) {
3.           j = i;
4.           while (j <= i) {
5.                if (A[i] < A[j])
6.                     swap(A[i], A[j]);
7.                j = j + 1; }
8.      i = i + 1; }
9. }
```



The graph of a module has an entry node and an exit node, corresponding to the first and last blocks of statements (or we can create artificial nodes for simplicity, as in the example). In such graphs there will be a path from the entry node to any node and a path from any node to the exit node (assuming the program has no anomalies like unreachable code). For such a graph, the cyclomatic number can be 0 if the code is a linear sequence of statements without any control statement. If we draw an arc from the exit node to the entry node, the graph will be strongly connected because there is a path between any two nodes. The cyclomatic number of a graph for any program will then be nonzero, and it is desirable to have a nonzero complexity for a simple program without any conditions (after all, there is some complexity in such a program). Hence, for computing the cyclomatic complexity of a program, an arc is added from the exit node to the start node, which makes it a strongly connected graph. For a module, the cyclomatic complexity is defined to be the cyclomatic number of such a graph for the module. As it turns out the cyclomatic complexity of a module (or cyclomatic number of its graph) is equal to the maximum number of linearly independent circuits in the graph. A set of circuits is linearly independent if no circuit is totally contained in another circuit or is a combination of other circuits. So, for calculating the cyclomatic number of a module, we can draw the graph, make it connected by drawing an arc from the exit node to the entry node, and then either count the number of circuits or compute it by counting the number of edges and nodes. In the graph shown above, the cyclomatic complexity is

$$V(G) = 1\ 0 - 7 + 1 = 4.$$

The independent circuits are:

ckt 1: b c e b
ckt 2: b c d e b
ckt 3: a b f a
ckt 4: a g a

It can also be shown that the cyclomatic complexity of a module is the number of decisions in the module plus one, where a decision is effectively any conditional statement in the module. Hence, we can also compute the cyclomatic complexity simply

by counting the number of decisions in the module. For this example, as we can see, we get the same cyclomatic complexity for the module if we add 1 to the number of decisions in the module. (The module has three decisions: two in the two while statements and one in the if statement.)

The cyclomatic number is one quantitative measure of module complexity. It can be extended to compute the complexity of the whole program, though it is more suitable at the module level. McCabe proposed that the cyclomatic complexity of modules should, in general, be kept below 10. The cyclomatic number can also be used as a number of paths that should be tested during testing. Cyclomatic complexity is one of the most widely used complexity measures.