Software Design 5

Software design is more creative process than analysis because it deals with the development of the actual mechanics for a new workable system. While analysing, it is possible to produce the correct model of an existing system. However, there is, no such thing as correct design. Good design is always system dependent and what is good design for one system may be bad for another.

The design of the new system must be done in great detail as it will be the basis for future computer programming and system implementation. Design is a problem-solving activity and as such, very much a matter of trial and error. The designer, together with users, has to search for a solution using his/her professional wisdom until there is an agreement that a satisfactory solution has been found.

For small projects (such as student's projects), one can sit with the specifications and simply write a program. For larger projects, it is necessary to bridge the gap between specifications and the coding with something more concrete. This bridge is the software design.

5.1 WHAT IS DESIGN?

Design is the highly significant phase in the software development where the designer plans "how" a software system should be produced in order to make it functional, reliable and reasonably easy to understand, modify and maintain. A software requirements specifications (SRS) document tells us "what" a system does, and becomes input to the design process, which tells us "how" a software system works. Designing software systems means determining how requirements are realized and result is a software design document (SDD). Thus, the purpose of design phase is to produce a solution to a problem given in SRS document.

A framework of the design is given in Fig. 5.1. It starts with initial requirements and ends up with the final design. Here, data is gathered on user requirements and analysed accordingly. A high level design is prepared after answering questions of requirements. Moreover, design is validated against requirements on regular basis. Design is refined in every cycle and finally it is documented to produce software design document. Fig. 5.1 shows the design framework.

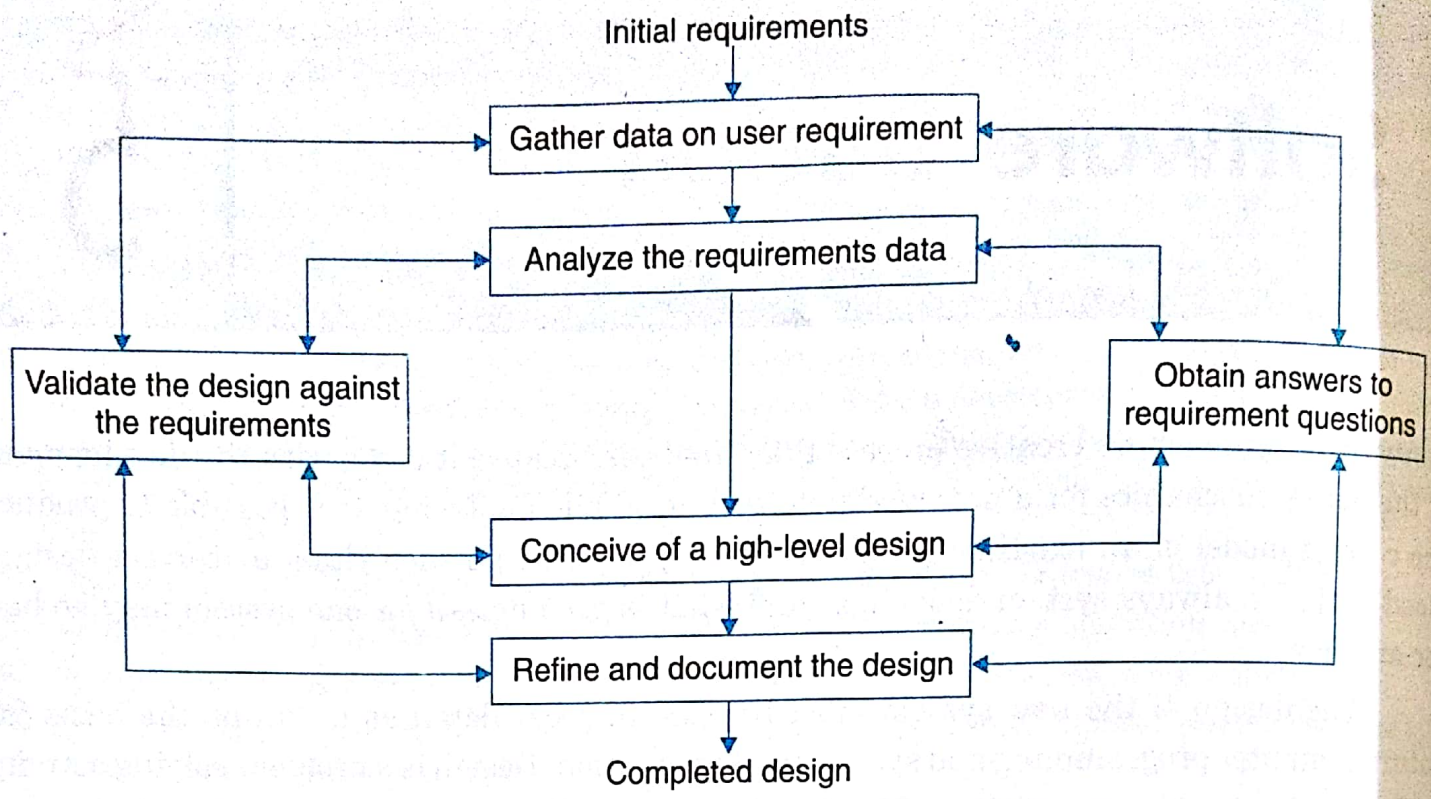


Fig. 5.1: Design framework

5.1.1 Conceptual and Technical Designs

The process of software design involves the transformation of ideas into detailed implementation descriptions, with the goal of satisfying the software requirements. To transform requirements into a working system, designers must satisfy both customers and the system builders (coding persons). The customers understand what the system is to do. At the same time, the system builders must understand how the system is to work. For this reason, design is really a two part, iterative process. First, we produce conceptual design that tells the customer exactly what the system will do. Once the customer approves the conceptual design, we translate the conceptual design into a much more detailed document, the technical design, that allows system builders to understand the actual hardware and software needed to solve the customer's problem. This two part design process is shown in Fig. 5.2.

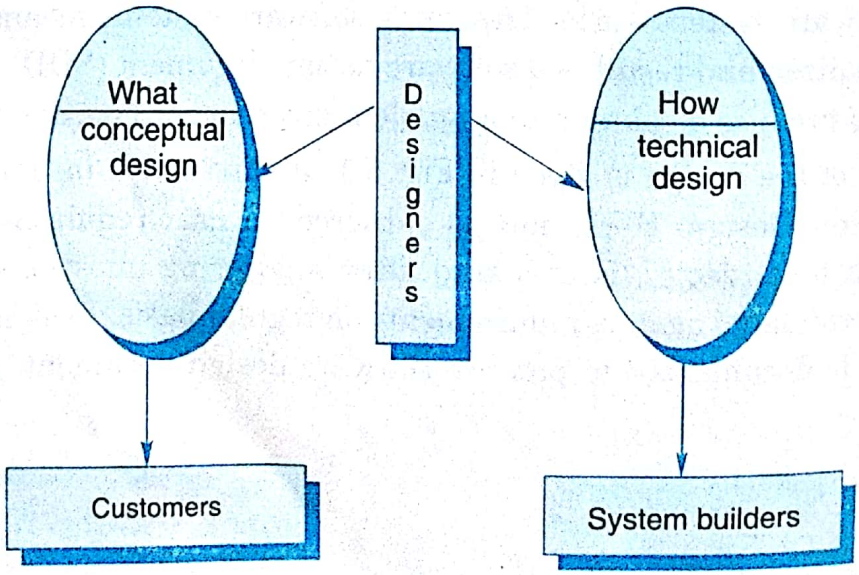


Fig. 5.2: A two-part design process

The two design documents describe the same system, but in different ways because of the different audiences for the documents. The conceptual design answers the following questions [PFLE98].

- Where will the data come from?
- What will happen to the data in the system?
- How will the system look to users?
- What choices will be offered to users?
- What is the timing of events?
- How will the reports and screens look like?

The conceptual design describes the system in language understandable to the customer. It does not contain any technical jargons and is independent of implementation.

By contrast, the technical design describes the hardware configuration, the software needs, the communications interfaces, the input and output of the system, the network architecture, and anything else that translates the requirements into a solution to the customer's problem.

Sometimes customers are very sophisticated and they can understand the "what" and "how" together. This can happen when customers are themselves software developers and may not require conceptual design. In such a cases comprehensive design document may be produced.

5.1.2 Objectives of Design

The specification (*i.e.* the "outside" view) of a program should obviously be as free as possible of aspects imposed by "how" the program will work (*i.e.* the "inside" view). It is seldom a document from which coding can directly be done. So design fills the gap between specifications and coding; taking the specifications, deciding how the program will be organized, and the methods it will use, in sufficient detail as to be directly codeable.

If the specification calls for a large or complex program (or both), then the design is quite likely to work down through a number of levels. At each level, breaking the implementation problem into a combination of smaller and simpler problems. Filling a large gap will involve a number of stepping-stones! The wider the gap, the larger the number of stepping-stones. The design needs to be

- Correct and complete
- Understandable
- At the right level
- Maintainable, and to facilitate maintenance of the produced code.

Software designers do not arrive at a finished design document immediately but develop the design iteratively through a number of different phases. The design process involves adding details as the design is developed with constant backtracking to correct earlier, less formal, designs. The starting point is an informal design which is refined by adding information to make it consistent and complete and this is shown in Fig. 5.3 [SOMM2K].

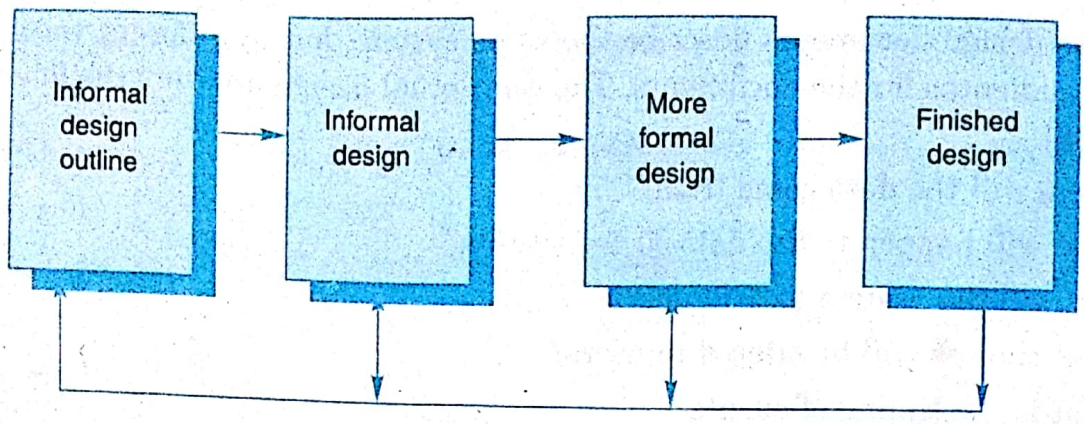


Fig. 5.3: The transformation of an informal design to a detailed design

5.1.3 Why Design is Important?

A good design is the key to successful product. Almost 2000 years ago Roman Architect Vitruvious recorded the following attributes of a good design:

- Durability
- Utility and
- Charm

A well-designed system is easy to implement, understandable and reliable and allows for smooth evolution. Without design, we risk building an unstable system:

- One that will fail when small changes are made
- One that will be difficult to maintain
- One whose quality cannot be assessed until late in the software process.

Therefore, software design should contain a sufficiently complete, accurate and precise solution to a problem in order to ensure its quality implementation.

There are three characteristics that serve as a guide for the evolution of a good design.

- The design must implement all of the explicit requirements contained in the analysis model and it must accommodate all of the implicit requirements desired by the customer.
- The design must be readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional and behavioural domain from an implementation perspective.

5.2 MODULARITY

There are many definitions of the term "module". They range from "a module is a FORTRAN subroutine" to "a module is an Ada package" to "procedures and functions of PASCAL and C", to "C++ / Java Classes", to "Java packages" to "a module is a work assignment for an individual programmer" [FAIR2K]. All of these definitions are correct. A modular system consist of well defined, manageable units with well defined interfaces among the units. Desirable properties of a modular system include:

- Each module is a well defined subsystem that is potentially useful in other applications
- Each module has a single, well defined purpose
- Modules can be separately compiled and stored in a library
- Modules can use other modules
- Modules should be easier to use than to build
- Modules should be simpler from the outside than from the inside

Modularity is the single attribute of software that allows a program to be intellectually manageable [MYER78]. It enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the software product.

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. Here, one important question arises is to what extent we shall modularize. As the number of modules grows, the effort associated with integrating the module also grows. Fig. 5.4 establishes the relationship between cost/effort and number of modules in a software.

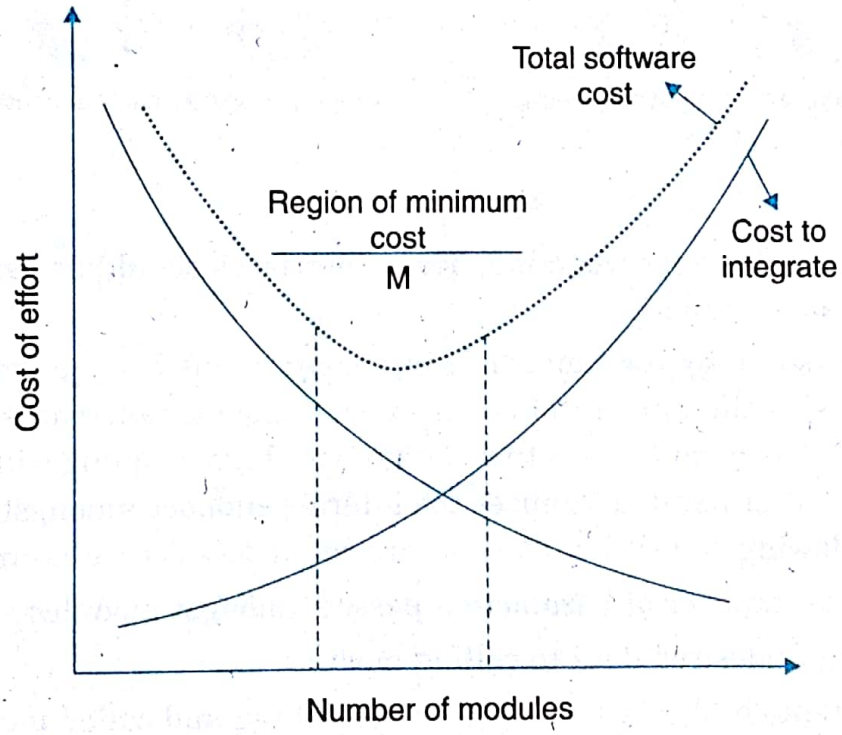


Fig. 5.4: Modularity and software cost

It can be observed that a software system cannot be made modular by simply chopping it into a set of modules. Each module needs to support a well defined abstraction and should have a clear interface through which it can interact with other modules. Thus, it is felt that under modularity and over modularity in a software should be avoided.

5.2.1 Module Coupling

Coupling is the measure of the degree of interdependence between modules. Two modules with high coupling are strongly interconnected and thus, dependent on each other. Two modules with low coupling are not dependent on one another. "Loosely coupled" systems are made up of

modules which are relatively independent. "Highly coupled" systems share a great deal of dependence between modules. For example, if modules make use of shared global variables. 'Uncoupled' modules have no interconnections at all; they are completely independent as shown in Fig. 5.5.

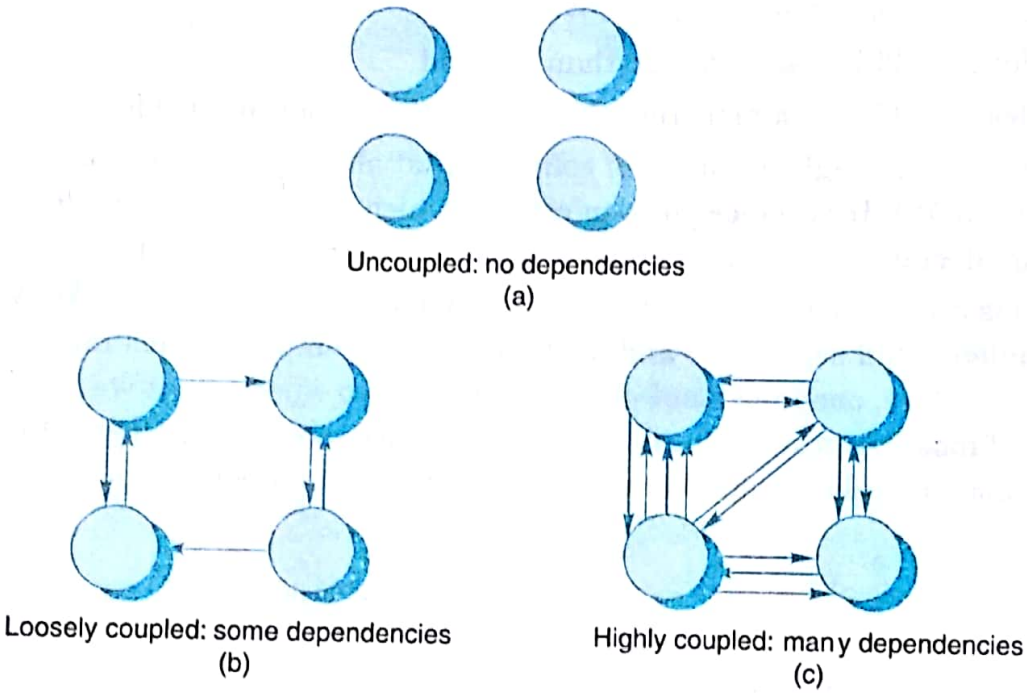


Fig. 5.5: Module coupling

A good design will have low coupling. Thus, interfaces should be carefully specified in order to keep low value of coupling.

Coupling is measured by the number of interconnections between modules. For example, coupling increases as the number of calls between modules increases, or the amount of shared data increases. The hypothesis is that design with high coupling will have more errors. Loose coupling, on the other hand, minimizes the interdependence amongst modules. This can be achieved in the following ways:

- Controlling the number of parameters passed amongst modules
- Avoid passing undesired data to calling module
- Maintain parent/child relationship between calling and called modules
- Pass data, not the control information

Fig. 5.6 demonstrates two alternative design for editing a student record in a "Student Information System".

The first design demonstrates tight coupling wherein unnecessary information as student name, student address, course is passed to the calling module. Passing superfluous information unnecessary increases the overhead, reducing the system performance/efficiency.

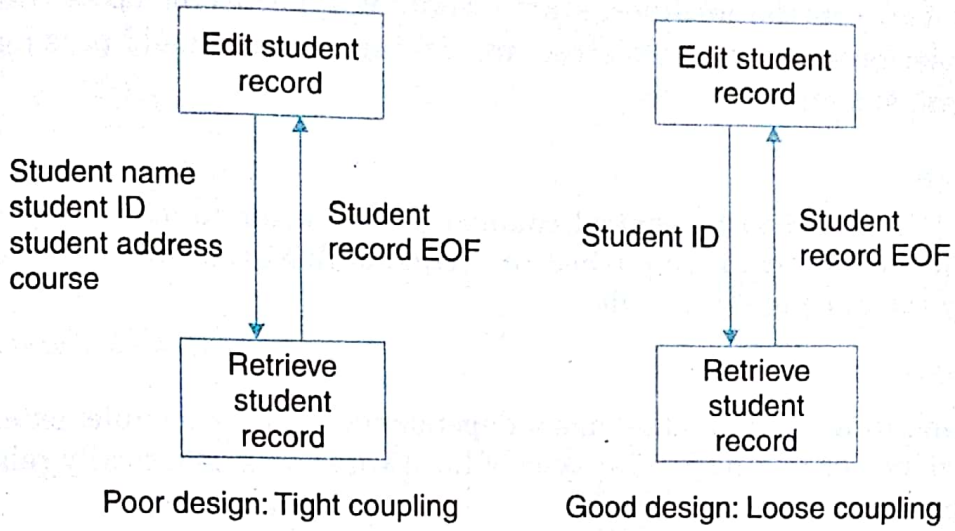


Fig. 5.6: Example of coupling

Types of coupling

Different types of coupling are content, common, external, control, stamp and data. The strength of coupling from lowest coupling (best) to highest coupling (worst) is given in Fig. 5.7.

Data coupling	Best
Stamp coupling	
Control coupling	
External coupling	
Common coupling	
Content coupling	(Worst)

Fig. 5.7: The types of module coupling

Given two procedures A and B, we can identify a number of ways in which they can be coupled.

Data coupling

The dependency between module A and B is said to be data coupled if their dependency is based on the fact they communicate by only passing of data. Other than communicating through data, the two modules are independent. A good strategy is to ensure that no module communication contains "tramp data". In Fig. 5.6 above students name, address, course are examples of tramp data that are unnecessarily communicated between modules. By ensuring that modules communicate only necessary data, module dependency is minimized.

Stamp coupling

Stamp coupling occurs between module A and B when complete data structure is passed from one module to another. Since not all data making up the structure are usually necessary in

communication between the modules, stamp coupling typically involves tramp data. If one procedure only needs a part of a data structure, calling module should pass just that part, not the complete data structure.

Control coupling

Module A and B are said to be control coupled if they communicate by passing of control information. This is usually accomplished by means of flags that are set by one module and reacted upon by the dependent module.

External coupling

A form of coupling in which a module has a dependency to other module, external to the software being developed or to a particular type of hardware. This is basically related to the communication to external tools and devices.

Common coupling

With common coupling, module A and module B have shared data. Global data areas are commonly found in programming languages. Making a change to the common data means tracing back to all the modules which access that data to evaluate the effect of change. With common coupling, it can be difficult to determine which module is responsible for having set a variable to a particular value. Fig. 5.8 shows how common coupling works [PFLE98]

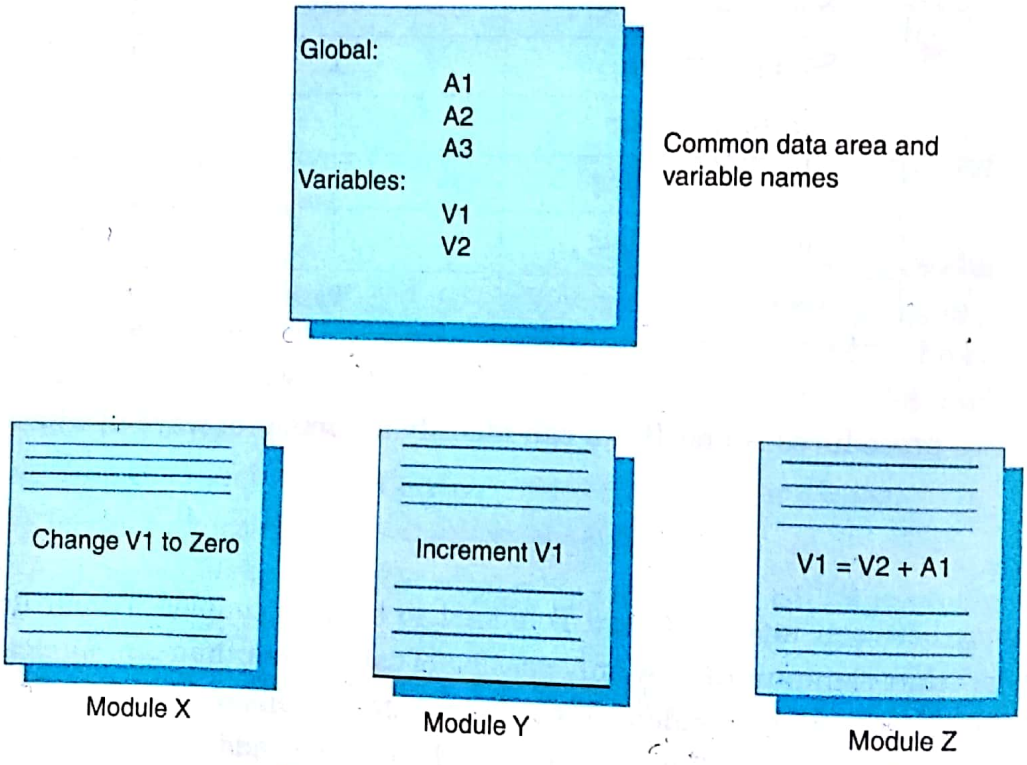


Fig. 5.8: Example of common coupling

Content coupling

Content coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another. In Fig. 5.9, module B branches into D, even though D is supposed to be under the control of C.

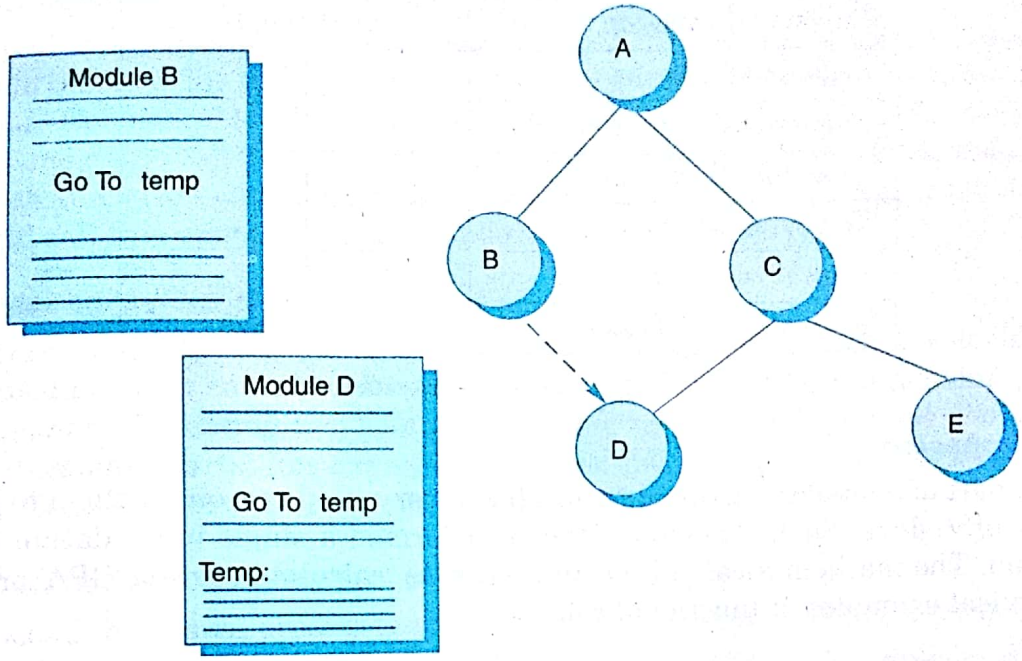


Fig. 5.9: Example of content coupling

5.2.2 Module Cohesion

Cohesion is a measure of the degree to which the elements of a module are functionally related. A strongly cohesive module implements functionality that is related to one feature of the solution and requires little or no interaction with other modules. This is shown in Fig. 5.10. Cohesion may be viewed as a glue that keeps the module together. It is a measure of the mutual officity of the components of a module.

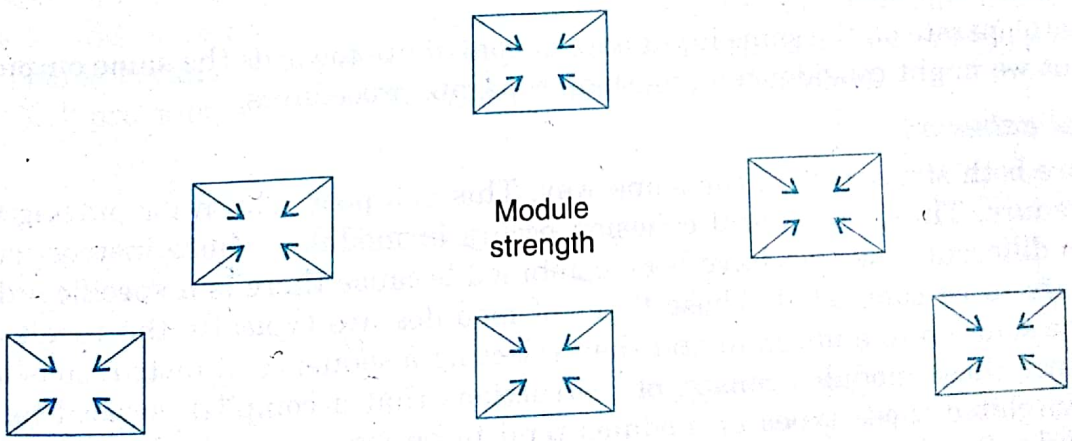


Fig. 5.10: Cohesion = Strength of relations within modules

Thus, we want to maximize the interaction within a module. Hence, an important design objective is to maximize the module cohesion and minimize the module coupling.

Types of cohesion

There are seven types or levels of cohesion and are shown in Fig. 5.11. Given a procedure that carries out operations X and Y, we can describe various forms of cohesion between X and Y.

Functional cohesion	Best (high)
Sequential cohesion	
Communicational cohesion	
Procedural cohesion	
Temporal cohesion	
Logical cohesion	
Coincidental cohesion	Worst (low)

Fig. 5.11: Types of module cohesion

Functional cohesion

X and Y are part of a single functional task. This is very good reason for them to be contained in the same procedure. Such a module often transformed a single input datum into a single output datum. The mathematical subroutines such as 'calculate current GPA' or 'cumulative GPA' are typical examples of functional cohesion.

Sequential cohesion

X outputs some data which forms the input to Y. This is the reason for them to be contained in the same procedure.

For example, addition of marks of individual subjects into a specific format is used to calculate the GPA as input for preparing the result of the students.

A component is made of parts that need to communicate/exchange data form one source for different functional purposes. They are together in a component for communicational convenience. For example calculate current and cumulative GPA uses the "student grade record" as input.

Communicational cohesion

X and Y both operate on the same input data or contribute towards the same output data. This is okay, but we might consider making them separate procedures.

Procedural cohesion

X and Y are both structured in the same way. This is a poor reason for putting them in the same procedure. Thus, procedural cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed. These types of modules are typically the result of first flow charting the solution to a program and then selecting a sequence of instructions to serve as a module. Since these modules consist of instructions that accomplish several tasks that are virtually unrelated these types of modules tend to be less maintainable. For example, if a report module of an examination system includes the following "calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA" is a case of procedural cohesion.

Temporal cohesion

X and Y both must perform around the same time. So, module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time-span. The set of functions responsible for initialization, start up activities such as setting program counters or control flags associated with programs exhibit temporal cohesion. This is not a good reason to put them in same procedure.

Logical cohesion

X and Y perform logically similar operations. Therefore, logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions. Considerable duplication can exist in the logical strength level. For example, more than one data item in an input transaction may be a date. Separate code would be written to check that each such date is a valid date. A better way to construct a DATECHECK module and call this module whenever a date check is necessary.

Coincidental cohesion

X and Y here no conceptual relationship other than shared code. Hence, coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another. That is, instead of creating two components, each of one part, only one component is made with two unrelated parts. For example, check validity and print is a single component with two parts. Coincidental cohesion is to be avoided as far as possible.

5.2.3 Relationship between Cohesion and Coupling

The essence of the design process is that the system is decomposed into parts to facilitate the capability of understanding and modifying a system. Projects rarely gets into trouble because of massive requirement changes. These changes can be properly recognized and properly reviewed.

If the software is not properly modularized, a host of seemingly trivial enhancement or changes will result into death of the project. Therefore, a good software design professes clean decomposition of a problem into modules and the arrangement of these modules in a neat hierarchy. Therefore, a software engineer must design the modules with goal of high cohesion and low coupling.

A good example of a system that has high cohesion and low coupling is the 'plug and play' feature of the computer system. Various slots in the mother board of the system simply facilitate to add or remove the various services/functionalities without affecting the entire system. This is because the add on components provide the services in highly cohesive manner. Fig. 5.12 provides a graphical review of cohesion and coupling.

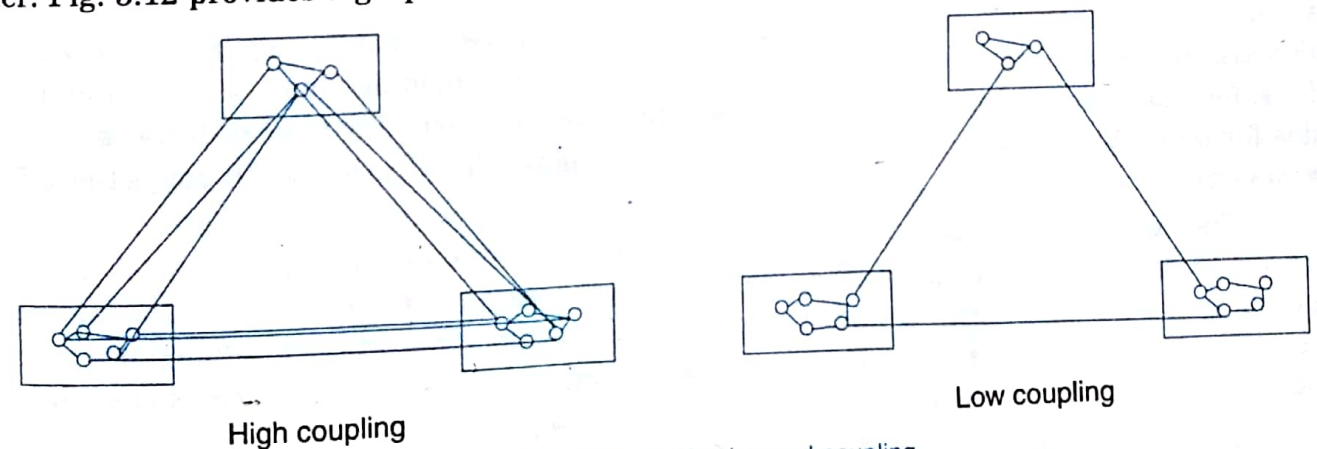


Fig. 5.12: View of cohesion and coupling

Module design with high cohesion and low coupling characterizes a module as black box when the entire structure of the system is described. Each module can be dealt separately when the module functionality is described.

5.3 STRATEGY OF DESIGN

A good system design strategy is to organize the program modules in such a way that are easy to develop and later to, change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program. It is important for two reasons:

- First, even pre-existing code, if any, needs to be understood, organized and pieced together.
- Second, it is still common for the project team to have to write some code and produce original programs that support the application logic of the system.

In early days, if any design was done, it was just "writing down the flowchart in words". Many people feel that flowcharts are too detailed, so leading to the detail often being decided too early, and too far from the specifications. Hence, there is a sudden jump from specifications to flow chart that leads to the cause of many errors. Flowcharts are at a low level. As a result, errors in flow-charts could only be found by coding them, seeing that the code ran wrongly, diagnosing that the error is in the flow chart, then diagnosing where in the flowchart, then fixing it, modifying code and recording. Repeated surgery on the flow chart sometimes lead to the final flow chart where further errors can not be fixed and the project may fail.

So writers of large and complex software now seldom use flow charts for design. The result is that we have designed other notations for expressing designs, and they are at a "higher level" than flow charts. This helps us to minimize the length of jumps from specifications to design and design to code. These notations usually permit multiple levels of design, and many small jumps in place of one or two massive jumps.

There are many strategies or techniques for performing system design. They include bottom up approach, top down approach, and hybrid approach.

5.3.1 Bottom-Up Design

A common approach is to identify modules that are required by many programs. These modules are collected together in the form of a "library". These modules may be for match functions, for input-output functions, for graphical functions etc. We may have collections of modules for result preparation system like "maintain student detail", "maintain subject details", "marks entry" etc.

This approach lead to a style of design where we decide how to combine these modules to provide larger ones; to combine those to provide even larger ones, and so on, till we arrive at one big module which is the whole of the desired program. The set of these modules form a hierarchy as shown in Fig. 5.13. This is a cross-linked tree structure in which each module is subordinate to those in which it is used.

Since the design progressed from bottom layer upwards, the method is called bottom-up design. The main argument for this design is that if we start coding a module soon after its design, the chances of recoding is high; but the coded module can be tested and design can be validated sooner than a module whose sub modules have not yet been designed.

This method has one terrible weakness; we need to use a lot of intuition to decide exactly what functionality a module should provide.

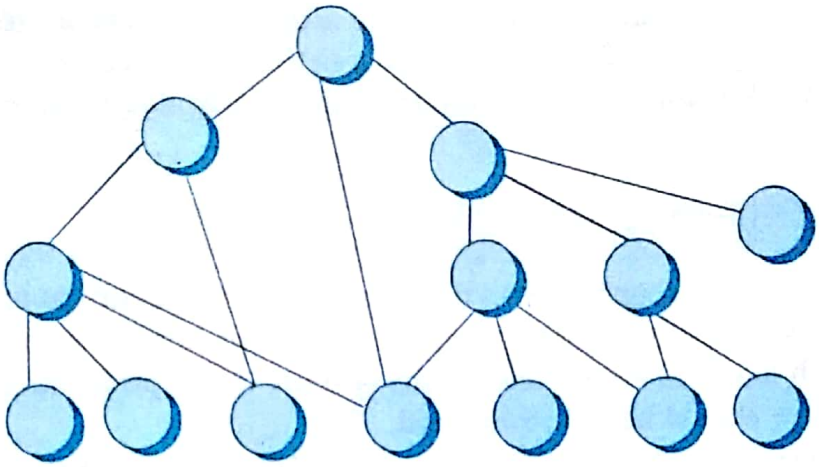


Fig. 5.13: Bottom-up tree structure

If we get it wrong, then at a higher level, we will find that it is not as per requirements; then we have to redesign at a lower level. If a system is to be built from an existing system, this approach is more suitable, as it starts from some existing modules.

5.3.2 Top-Down Design

The essential idea of top-down design is that the specification is viewed as describing a black box for the program? The designer should decide how the internals of the black box is constructed from smaller black boxes; and that those inner black boxes be specified. This process is then repeated for those inner boxes, and so on till the black boxes can be coded directly.

A top down design approach starts by identifying the major modules of the system, decomposing them into their lower level modules and iterating until the desired level of detail is achieved. This is stepwise refinement; starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly. Most design methodologies are based on this approach and this is suitable, if the specifications are clear and development is from the scratch. If coding of a part starts soon after it's design, nothing can be tested until all its subordinate modules are coded.

5.3.3 Hybrid Design

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 notion of 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 [JALO98].

For top-down approach to be effective, some bottom-up (mostly in the lowest design levels) approach is essential for the following reasons:

- To permit common sub modules
- Near the bottom of the hierarchy, where the intuition is simpler, and the need for bottom-up testing is greater, because there are more numbers of modules at low levels than at high levels.
- In the use of pre-written library modules, in particular, reuse of modules.

Hybrid approach has really become popular after the acceptance of reusability of modules. Standard libraries, microsoft foundation classes (MFCs), object oriented concepts are the steps in this direction. We may soon have internationally acceptable standards for reusability.

5.4 FUNCTION ORIENTED DESIGN

The design activity begins when the SRS document for the software to be developed is available. 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, their specifications of these modules, and how the modules should be interconnected.

Function oriented design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function. Thus, system is designed from a functional viewpoint.

One of the best-known advocates of this method is Niklaus Wirth, the creator of PASCAL and a number of other languages. His special variety is called stepwise refinement, and it is a top down design method. We start with a high level description of what the program does. Then, in each step, we take one part of our high level description and refine it, i.e. specify in somewhat greater detail what that particular part does.

This method works fine for small programs. For large programs its value is more questionable. The main problem is that it is not easy to know what a large program does. For instance, what does UNIX do? Or an airline reservation system? Or a scheme interpreter? The answer is that it depends on what the user types at the terminal. Still, one can usually come up with some kind of high-level function. The risk is that this function is a highly artificial description of reality.

Consider the example of scheme interpreter. Top-level function may look like:

```
While (not finished)
{
        Read an expression from the terminal;
        Evaluate the expression;
        Print the value;
}
```

We thus get a fairly natural division of our interpreter into a "read" module, an "evaluate" module and a "print" module. Now we consider the "print" module and is given below:

```
        Print (expression exp)
{
        Switch (exp → type)
        Case integer: /*print an integer*/
        Case real:    /*print a real*/
        Case list:    /*print a list*/
        :::
}
```

The other modules are structured in a similar way. We notice that the different kinds of objects that are to be manipulated by the scheme interpreter (integer, real, etc.) need to be known by every module. Thus, if we need to add a type, every module needs to be altered. Needless to say, we would like to avoid that.

We continue the refinement of each module until we reach the statement level of our programming language. At that point, we can describe the structure of our program as a tree of refinements as in design top-down structure as shown in Fig. 5.14.

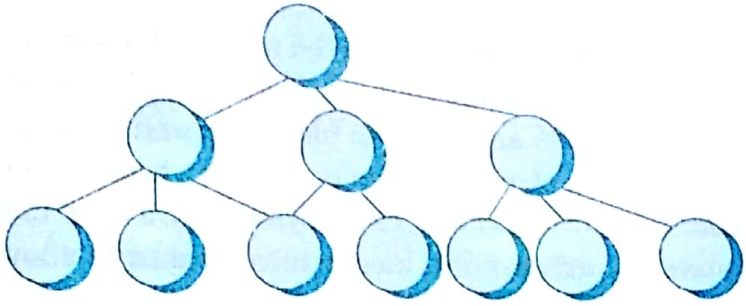


Fig. 5.14: Top-down structure

Unfortunately, if a program is created top-down, the modules become very specialized. As one can easily see in top down design structure, each module is used by at most one other module, its parent. For a module to be reusable, however, we must require that several other modules as in design-reusable structure as shown Fig. 5.15.

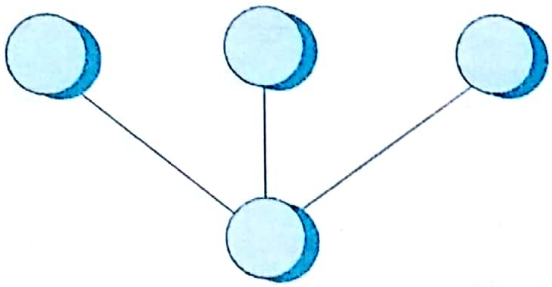


Fig. 5.15: Design reusable structure

It is, of course, not necessary to create a program top-down, even though its structure is function-oriented. However, if we want to delay the decision of what the system is supposed to do as long as possible, a better choice is to structure the program around the data rather than around the actions taken by the program.

5.4.1 Design Notations

During the design phase there are two things of interest: the design of the system, and the process of designing itself. It is for the latter that principles and methods are needed.

Design notations are largely meant to be used during the process of design and are used to represent design or design decisions. For a function oriented design, the design can be represented graphically or mathematically by the following:

- Data flow diagrams
- Data Dictionaries
- Structure Charts
- Pseudocode

The first two techniques have been discussed in chapter 3 and other two are discussed in this section.

Structure chart

The structure chart is one of the most commonly used method for system design. It partition a system into black boxes. A black box means that functionality is known to the user without the knowledge of internal design. Inputs are given to black box and appropriate outputs are generated by the black box. This concept reduces the complexity because details are hidden from those whose have no need or desire to know. Thus, systems are easy to construct and easy to maintain. Here, black boxes are arranged in hierarchical format as shown in Fig. 5.16.

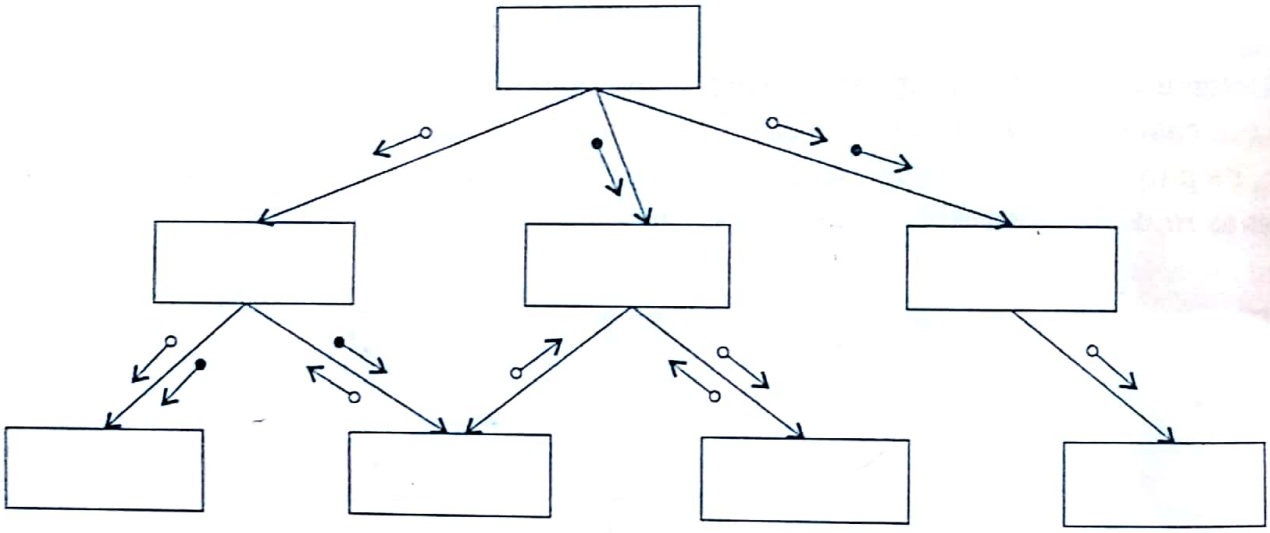


Fig. 5.16: Hierarchical format of a structure chart

In a structure chart, each program module is represented by a rectangular box. Modules at the top level call the modules at the lower level. The connection between modules are represented by lines between the rectangular boxes. The components are generally read from top to bottom, left to right. Modules are numbered in hieratical numbering scheme.

When a module calls another, it views the called module as a black box, passing parameters needed for the called module's functionality and receiving answers. Control data passed between modules on a structure chart are represented by labelled directed arrow with filled in circle and data is depicted with an open circle. When a module is used by many other modules, it is put into the library of modules. The diamond symbol is used to represent the fact that one module out of several modules connected with the diamond symbol is used depending on the outcome of the condition attached to the diamond symbol. A loop around the control flow arrows denotes that the respective modules are used repeatedly and is called repetition symbol. Fig. 5.17 shows the notations used in structure chart:

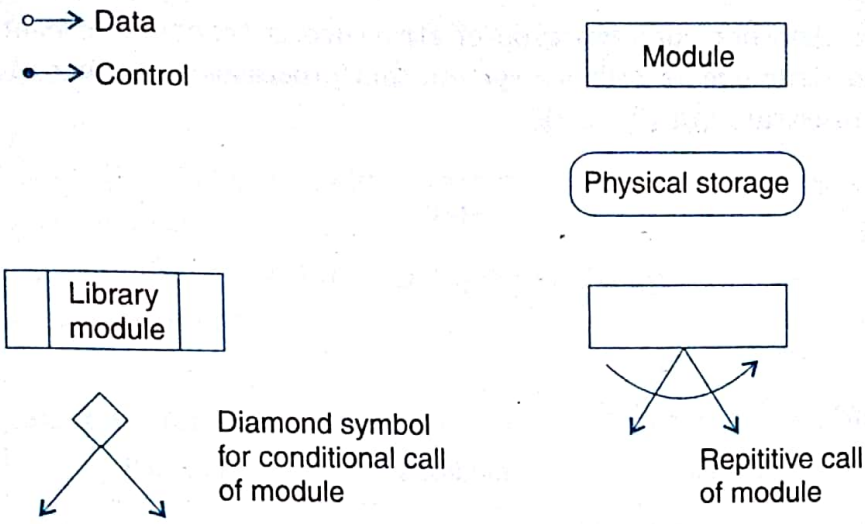


Fig. 5.17: Structure chart notations

A structure chart for "update file" is given in Fig. 5.18.

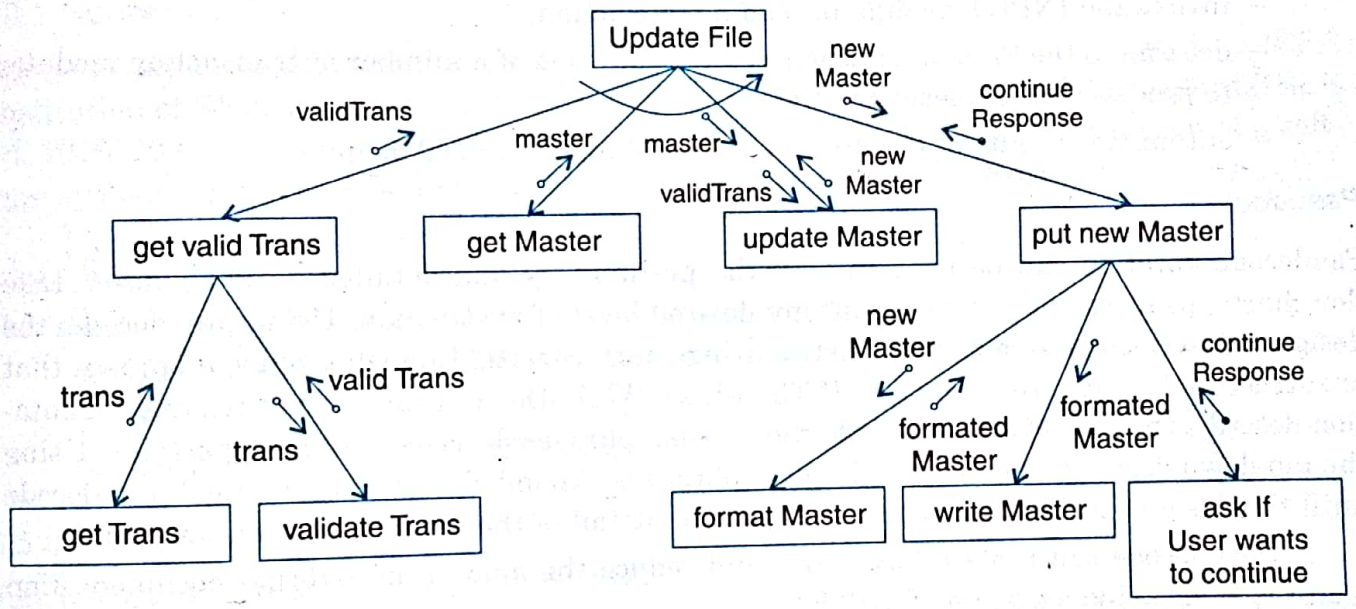


Fig. 5.18: Update file

The 'Update file' calls 'get Valid Trans' to get input parameter Valid Trans. Module 'get valid tans' calls 'get trans' module to read the trans from the input device and pass it back to 'get Valid Trans'. 'Get Valid Trans' then calls validate Trans 'module' to validate the transaction which is subsequently passed to the 'update file' module. 'Update file' module then calls the get master and passes the master and 'valid trans' information to 'update master'. 'Update' master module passes the new master data to 'update file'.

'Update file' then involves 'put new master' by passing 'new master' data to it. Put new Master involves format master 'write Master' and ask if user wants to continue Module 'Continue Resource control data' is passed to 'update file' to decide user wants to continue by repeating the above procedure.

This type of structure chart is often called as transform-centred structures. Transform-centered structure clout receive an input which is transformed by a sequence of operations, with each operation being carried out by one module. Fig. 5.19 above is an example of transform-

centred structures. Another common type of structure is transaction centred structure. A transaction centred structure describes a system that processes a number of different types of transactions. It is illustrated in Fig. 5.19.

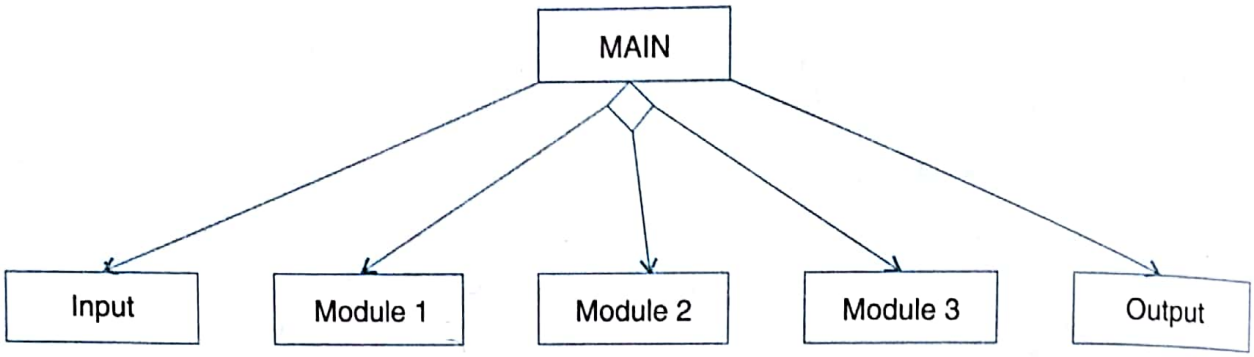


Fig. 5.19: Transaction-centered structure

In the above figure the MAIN module controls the system operation its function is to:

- invoke the INPUT module to read a transaction;
- determine the kind of transaction and select one of a number of transaction modules to process that transaction, and
- output the results of the processing by calling OUTPUT module.

Pseudocode

Psudocode notation can be used in both the preliminary and detailed design phases. Like flowcharts, psudocode can be used at any desired level of abstraction. Using pseudocode, the designer describes system characteristics using short, concise, English language phrases that are structured by key words such as If-Then-Else, While-Do, and End. Keywords and indentation describe the flow of control, while the English phrases describe processing actions. Using the top-down design strategy, each English phrase is expanded into more detailed pseudocode until the design specification reaches the level of detail of the implementation language.

Pseudocode can replace flowcharts and reduce the amount of external documentation required to describe a system [FAIR2K].

5.4.2 Functional Procedure Layers

- Functions are built in layers, Additional notation is used to specify details.
- Level 0
  - Function or procedure name
  - Relationship to other system components (e.g., part of which system, called by which routines, etc.)
  - Brief description of the function purpose.
  - Author, date.
- Level 1
  - Function parameters (problem variables, types, purpose, etc.)
  - Global variables (problem variable, type, purpose, sharing information)
  - Routines called by the function.

- Side effects.
- Input/Output Assertions.

- Level 2
  - Local data structures (variable etc.)
  - Timing constraints
  - Exception handling (conditions, responses, events)
  - Any other limitations.
- Level 3
  - Body (structured chart, English pseudo code, decision tables, flow charts, etc.)

5.5 IEEE RECOMMENDED PRACTICE FOR SOFTWARE DESIGN DESCRIPTIONS (IEEE STD 1016–1998)

5.5.1 Scope

This is a recommended practice for describing software designs. This is designed by IEEE (Institution of Electricals and Electronics Engineers) and is known as IEEE Standard (IEEE std. 1016–1998) for software design description (SDD). An SDD is a representation of a software system that is used as a medium for communicating software design information.

5.5.2 References

This standard shall be used in conjunction with the following publications.

(*i*) IEEE std 830–1998, IEEE recommended practice for software requirements specifications

(*ii*) IEEE std 610.12–1990, IEEE glossary of software engineering terminology.

5.5.3 Definitions

Few important definitions are given below:

(*i*) **Design entity:** An element (component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

(*ii*) **Design View:** A subset of design entity attribute information that is specifically suited to the needs of a software project activity.

(*iii*) **Entity attribute:** A named property or characteristics of a design entity. It provides a statement of fact about the entity.

(*iv*) **Software design description (SDD):** A representation of a software system created to facilitate analysis, planning, implementation and decision making. A blue print or model of the software system. The SDD is used as the primary medium for communicating software design information.

5.5.4 Purpose of an SDD

The SDD shows how the software system will be structured to satisfy the requirements identified in the SRS. It is basically the translation of requirements into a description of the software

Table 5.2: Design views

Design view	*Scope*	*Entity attribute*	*Example representation*
Decompostion description	Partition of the system into design entities.	Identification, type purpose, function, subordinate	Hierarchical decomposition diagram, natural language
Dependency description	Description of relationships among entities of system resources	Identification, type, purpose, dependencies, resources	Structure chart, data flow diagrams, transaction diagrams
Interface description	List of everything a designer, developer, tester needs to know to use design entities that make up the system	Identification, function, interfaces	Interface files, parameter tables
Detail description	Description of the internal design details of an entity	Identification, processing, data	Flow charts, PDL etc.

5.6 OBJECT ORIENTED DESIGN

"Object Oriented" has clearly become the buzzword of choice in the industry. Almost everyone talks about it. Almost everyone claims to be doing it, and almost everyone says it is better than traditional function oriented design. Object oriented design is the result of focusing attention not on the function performed by the program, but instead on the data that are to be manipulated by the program. Thus, it is orthogonal to function oriented design.

Object Oriented Design begins with an examination of the real world "things" that are part of the problem to be solved. These things (which we will call objects) are characterized individually in terms of their attributes (transient state information) and behaviour (functional process information). Each object maintains its own state, and offers a set of services to other objects. Shared data areas are eliminated and objects communicate by message passing (e.g. parameters). Objects are independent entities that may readily be changed because all state and representation information is held within the object itself. Objects may be distributed and may execute either sequentially or in parallel [BOOC03].

5.6.1 Basic Concepts

Object Oriented Design is not dependent on any specific implementation language. Problems are modelled using objects. Objects have:

- Behaviour (they do things)
- State (which changes when they do things)

For example, a car is an object. It has state: whether its engine is running; and it has a behaviour: starting the car, which changes its state from "engine not running" to "engine running".

The various terms related to object oriented design are Objects, Classes, Abstraction, Inheritance and Polymorphism.

(*i*) **Objects:** The word "Object" is used very frequently and conveys different meaning in different circumstances. Here, meaning is an entity able to save a state (information) and which offers a number of operations (behaviour) to either examine or affect this state. Hence, an object is characterised by number of operations and a state which remembers the effect of these operations [JACO 98].

All objects have unique identification and are distinguishable. Two bananas may be of same colour, shape and texture but still are different. Each has an identity. There may be four dogs of same colour, breed and size but all are distinguishable. The term identity means that objects are distinguished by their inherent existence and not by descriptive properties [JOSHO 3].

As discussed above, object is an entity, which has a state (whose representation is hidden from other objects) and a defined set of operations, which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects, which request these services when some computation is required. In principle, objects communicate by passing messages to each other and these messages initiate object operations.

(*ii*) **Messages:** Conceptually, objects communicate by message passing. Messages consist of the identity of the target object, the name of the requested operation and any other operation needed to perform the function. In some distributed systems, object communications are implemented directly as text messages which objects exchange. The receiving object parses the message, identifies the service and the associated data and carries out the requested service. Messages are often implemented as procedure or function calls (name = procedure name, information = parameter list).

(*iii*) **Abstraction:** In object oriented design, complexity is managed using abstraction. *Abstraction is the elimination of the irrelevant and the amplification of the essentials.*

We can teach someone to drive any car using an abstraction. We amplify the essentials: we teach about the ignition and steering wheel, and we eliminate the details, such as details of the particular engine in this car or the way fuel is pumped to the engine where a spark ignites it, it explodes pushing down a piston and driving a crankshaft [FIEL01].

Problems usually have levels of abstraction. We can see a car at a high level of abstraction for driving, but mechanics need to work at a lower level of abstraction. They do care about details and need to know about batteries and engines.

However, there are abstractions at the mechanic's level too. The mechanic might test or charge a battery without caring that inside the battery there is a complex chemical reaction going on. A battery designer would care about these details but would not care about, say, the electronics that goes into the car's stereo.

We have seen that we can look at the details such as the battery or stereo design, and they are separated in manageable chunks, and by using abstraction and ignoring the details, we can also look at the whole car as a manageable chunk.

(*iv*) **Class:** In any system, there shall be number of objects. Some of the objects may have common characteristics and we can group the objects according to these characteristics. This type of grouping in known as a class. Hence, a class is a set of objects that share a common structure and a common behaviour.

We may have a class "car" with objects like Indica, Santro, Maruti, Indigo. These objects are related due to some common characteristics to constitute a class named "car". A class may be object defined as [JACO98]:

"A class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structures.

In object oriented system, each object belongs to a class. An objects, that belongs to a certain class is called an instance of that class. We often use object and instance as synonyms. Hence, an instance is an object created from a class. The class describes the structure (behaviour and information) of the instance, while the current state of the instance is defined by the operations performed on the instance.

We may define a class "car" and each object that represents a car becomes an instance of this class. In this class "car", Indica, Santro, Maruti, Indigo are instances of this class as shown in Fig. 5.20.

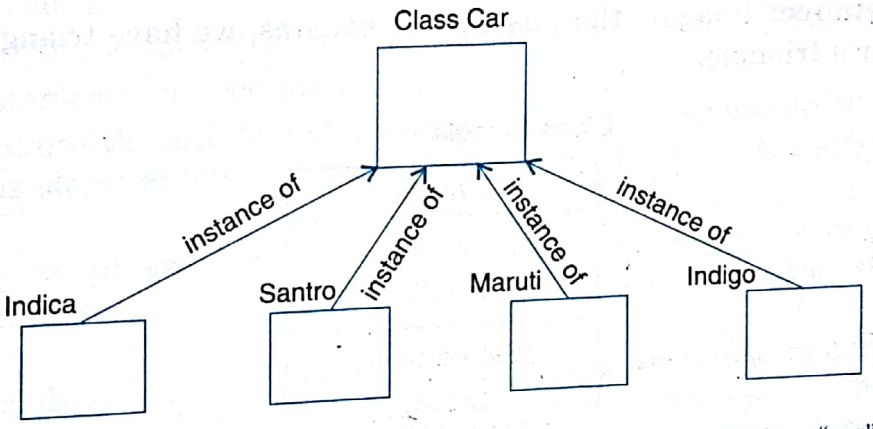


Fig. 5.20: Indica, Santro, Maruti, Indigo are all instances of the class "car"

We may have different types of classes depending upon common characteristics. The class is the static description and the object is an instance in runtime of that class.

Imagine a picture made up of squares. Each square is an object. It has a state: its colour and position, and behaviour. We can, amongst other things, change its colour and draw it. Each square is different but has much in common with other squares. So, we abstract out the commonalities: they share the same behaviour and have same sort of attributes. We have ignored same sort of attributes. We have ignored the particular values of the attributes.

Classes are useful because they act as a blue print for objects. If we want a new square we may use the square class and simply fill in the particular details (*i.e.*, colour and position). Fig. 5.21 shows how can we represent the square class.

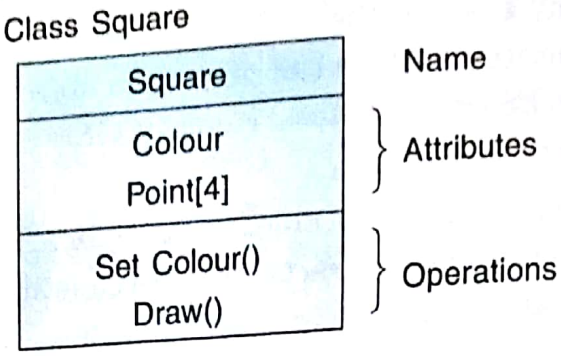


Fig. 5.21: The square class

(*v*) **Attributes:** An attribute is a data value held by the objects in a class. The square class has two attributes: a colour and array of points. Each attribute has a value for each object instance. For example, colour may be different in different objects and "array of points" size may also be different in different squares. The attributes are shown as second part of the class as shown in Fig. 5.21.

(*vi*) **Operations:** An operation is a function or transformation that may be applied to or by objects in a class. In the square class, we have two operations: set colour() and draw(). All objects in a class share the same operations. Each operation has a target object as an implicit argument. The behaviour of the operation depends on the class of its target. An object "knows" its class, and hence the right implementation of the operation [JOSHO3]. Operations are shown in the third part of the class as indicated in Fig. 5.21.

(*vii*) **Inheritance:** Imagine that, as well as squares, we have triangle class. Fig. 5.22 shows the class for a triangle.

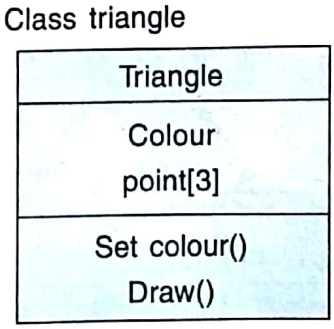


Fig. 5.22: The triangle class

Now, comparing Fig. 5.21 and Fig. 5.22, we can see that there is some difference between triangle and squares classes. Triangles have three vertices ; squares have four. Also, the way that these shapes are drawn is different. However, there are some similarities. For example, we can set the colour of both and both can be drawn (even if the way they are drawn is different). It would be nice if we could abstract; eliminate the details of each shape and amplify the fact that both can have their colours set and can be drawn. For example, at a high level of abstraction, we might want to think of a picture as made up of shapes and to draw the picture, we draw each shape in turn. We want to eliminate the irrelevant details: we do not care that one shape is a square and the other is a triangle as long as both can draw themselves.

To do this, we consider the important parts out of these classes in to a new class called Shape. Fig. 5.23 shows the results.

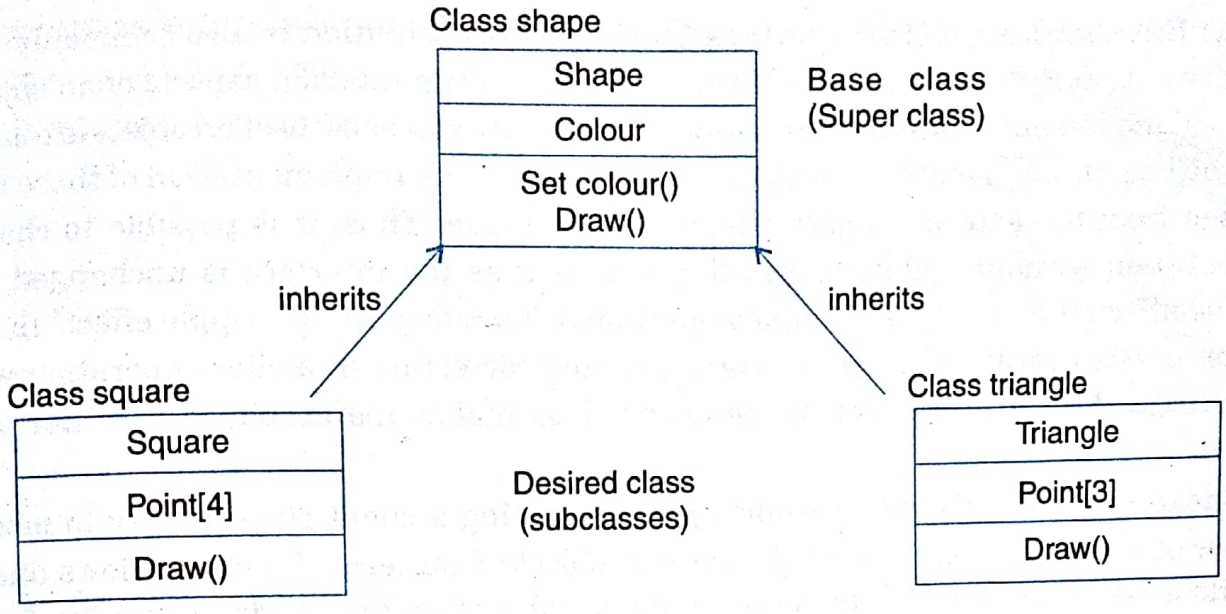


Fig. 5.23: Abstracting common features in a new class

This sort of abstraction is called inheritance. The low level classes (known as subclasses or derived classes) inherit state and behaviour from this high level class (known as a super class or base class).

Hence, a triangle object will have a colour, a setcolour() operation, three point and a draw() operation. We can inherit three sorts of things:

(*i*) State: colour

(*ii*) Operations: setcolour() should be able to set the colour for any shape.

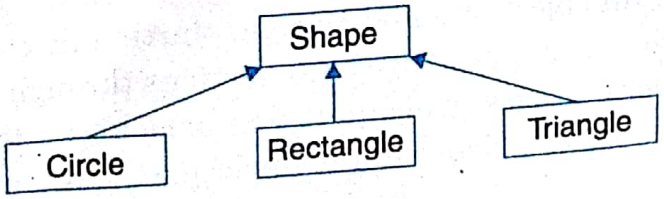
(*iii*) The interface of an operation.

Shape contains the interface of the draw() operation because draw() interface for triangle and square is identical but code for the operation remains in the subclasses because it is different.

(*viii*) **Polymorphism:** When we abstract just the interface of an operation and leave the implementation to subclasses it is called a polymorphic operation and process is called polymorphism.

So, we can abstract by pulling out important state, behaviour and interface into an new class. We can also abstract by combining object's inside a new object. In the car example, a car combines a battery, engine and other objects into a new object and provides a simple interface for driving that hides these details. There are different sorts of abstraction and finding the best ways to apply abstraction to a problem is what design is all about.

Another example may be for a base class *shape,* polymorphism enables the programmer to define different *area* methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the *area* method to it will return the correct results. Plolymorphism is considered to be a requirement of any true object-oriented programming language (OOPL).



(*ix*) **Encapsulation (Information Hiding):** Encapsulation is also commonly referred to as "Information Hiding. It consists of the separation of the external aspects of an object from the internal implementation details of the object. The external aspects of an object are accessible by other objects through methods of object, while the internal implementation of those methods are hidden from the external object sending the message. Thus, it is possible to change the implementation without updating the clients as long as the interface is unchanged. Clients will not be affected by changes in implementation, thus reducing the "ripple effect" in which a correction to one operation forces the corresponding correction in a client operation which in turn causes a change in a client of the client. This makes maintenance is easier and less expensive.

Encapsulation deals with permitting or restricting a client class' ability to modify the attributes or invoke the methods of the class or object of concern. If a class allows another to modify its attributes or invoke its methods, the attributes and methods are said to be part of the class' public interface. If a class doesn't allow another to modify its attributes or invoke its methods, those are part of the class' private implementation. Thus, Encapsulation protects (a) an object's internal state from being corrupted by its clients and (b) Client code from changes in the object's implementation.

A "Queue" provides a good example of this characteristic. A queue is an abstract concept that represents an ordered list of things. The implementation of a queue may by an array or it may be by a linked-list. If the implementation were known, a developer writing a client of the queue class may use this knowledge and directly access the internal storage mechanism. If the implementation changed, the client would then have to be modified also. This type of tight coupling between classes would cause a very brittle system and would increase maintenance costs as parts of the system were modified. Therefore, the levels of encapsulation that a language supports and how those mechanisms are used directly impacts the level of coupling between classes and it can significantly affect the cost of maintenance in an application.

(*x*) **Hierarchy:** Hierarchy involves organizing something according to some particular order or rank (*e.g.*, complexity, responsibility, etc.). It is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way. This hierarchy is implemented in software via a mechanism called "Inheritance". Just as a child inherits genes from its parent, a class can inherit attributes and behaviours from its parent. The parent class is commonly referred to as the supper-class and the child class as the sub-class. *Classes at the same level of the hierarchy should be at the same level of abstraction.*

Using a hierarchy to describe differences or variations of a particular concept provides for more descriptive and cohesive abstractions, as well as a better allocation of responsibility. In any one system, there may be multiple abstraction hierarchies (*e.g.*, in a financial application, you may have different types of customers and accounts). Generalization can be used to realize a hierarchy within an object-oriented system, starting at the most general of the abstractions, and then defining more specialized abstractions through sub-classing. Generalization can take place in several stages, which lets you model complex, multilevel inheritance hierarchies. General properties are placed in the upper part of the inheritance hierarchy, and

special properties lower down. In other words, you can use generalization to model specialization of a more general concept.

This relationship can be easily understood by the Fig. 5.24 below.

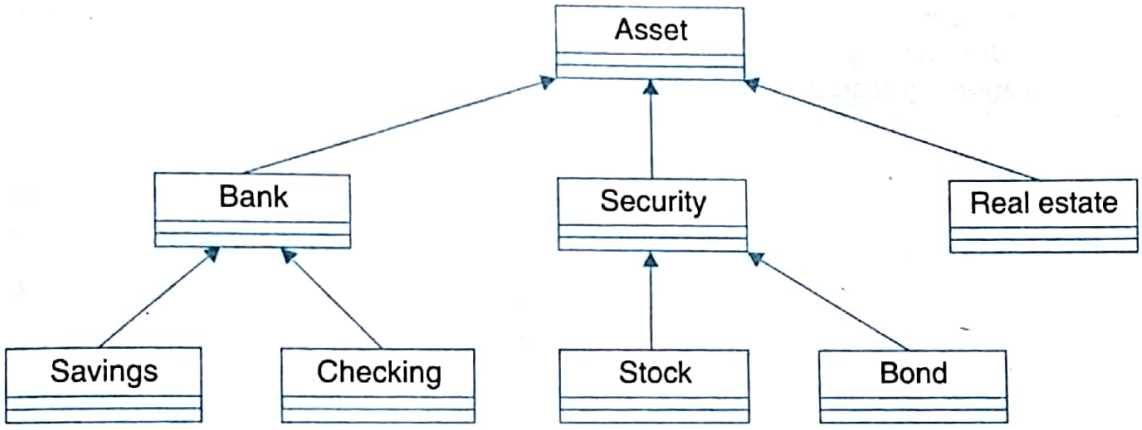


Fig. 5.24: Hierarcy

In summary, with object-oriented design, we use abstraction to break a problem into manageable chunks. We can comprehend the problem as a whole or study parts of the problem at lower level of abstraction.

If we work at an appropriate level of abstraction then any problems we find can be solved relatively easily. As an example of not working at the right level of abstraction, imagine that we tried to build a house by going out with some bricks and just building. The chances are we would finish, try to put the bath in and discover that the bathroom is too small. So, we have to knock down and rebuild a wall. Meaning, thereby, lot of work, assuming it could be done. If we had designed the house including, where the fittings were going. We would have noticed the problem and fixing it would have taken about 30 seconds with our eraser and pencil.

5.6.2 Steps to Analyze and Design Object Oriented System

There are various steps in the analysis and design of an object oriented system and are given in Fig. 5.25.

(*i*) **Create use case model:** First step is to identify the actors interacting with the system. We should then write the use case and draw the use case diagram.

(*ii*) **Draw activity diagram (If required):** Activity diagram illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. It is essentially like a flow chart. Fig. 5.26 shows the activity diagram processing an order to deliver some goods.