

# Software Design

# *Software Design*

---

- ❖ More creative than analysis
- ❖ Problem solving activity

## WHAT IS DESIGN

‘HOW’



Software design document (SDD)

# Software Design

---

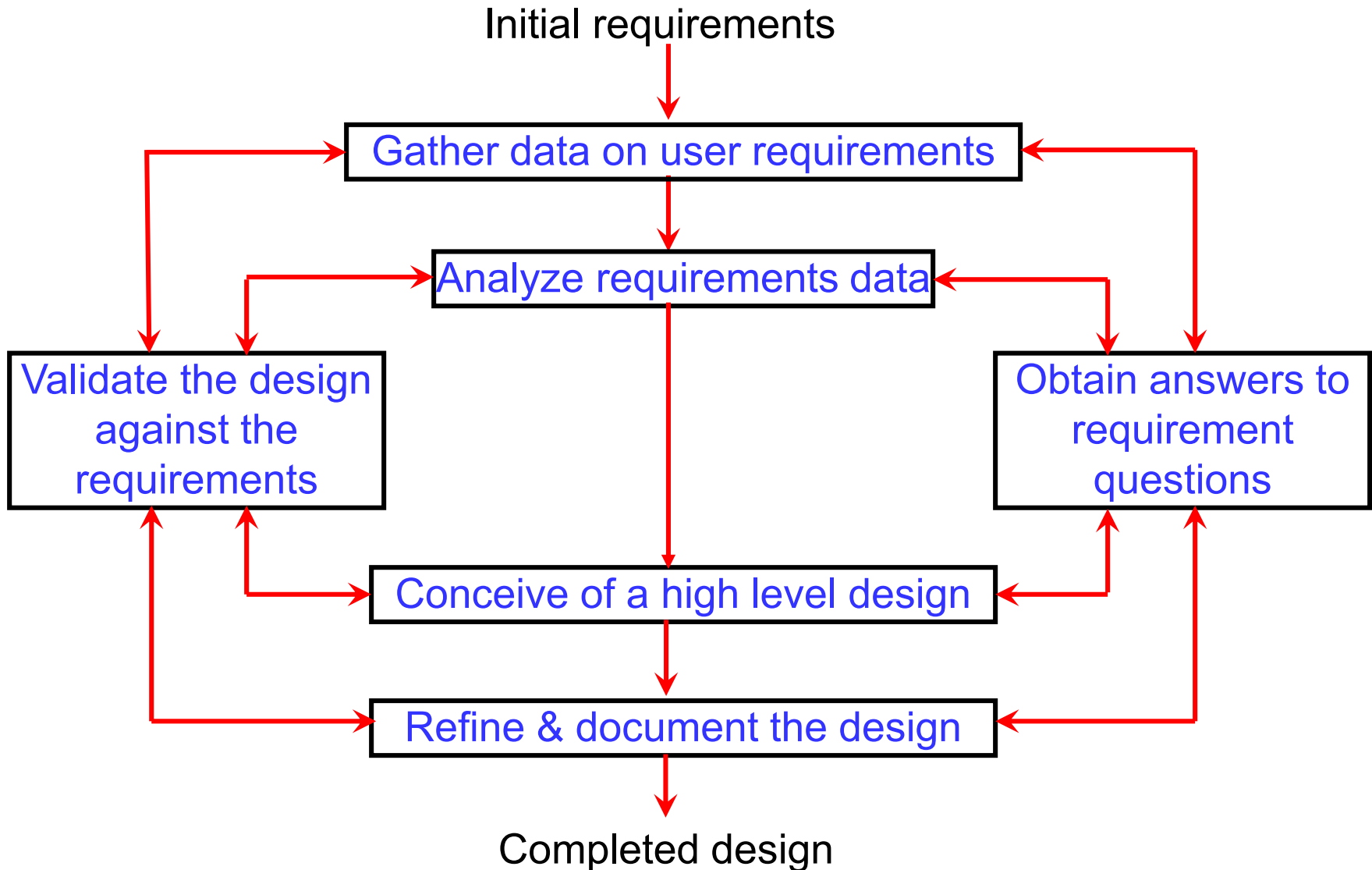
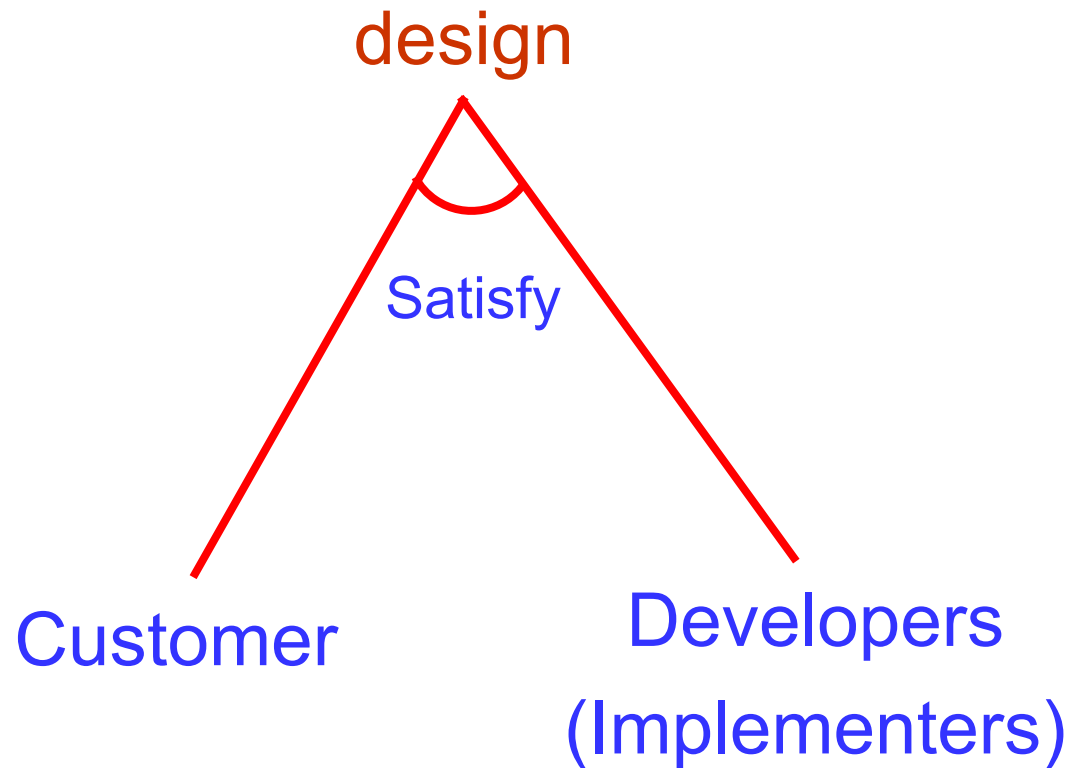


Fig. 1 : Design framework

# *Software Design*

---



# Software Design

## Conceptual Design and Technical Design

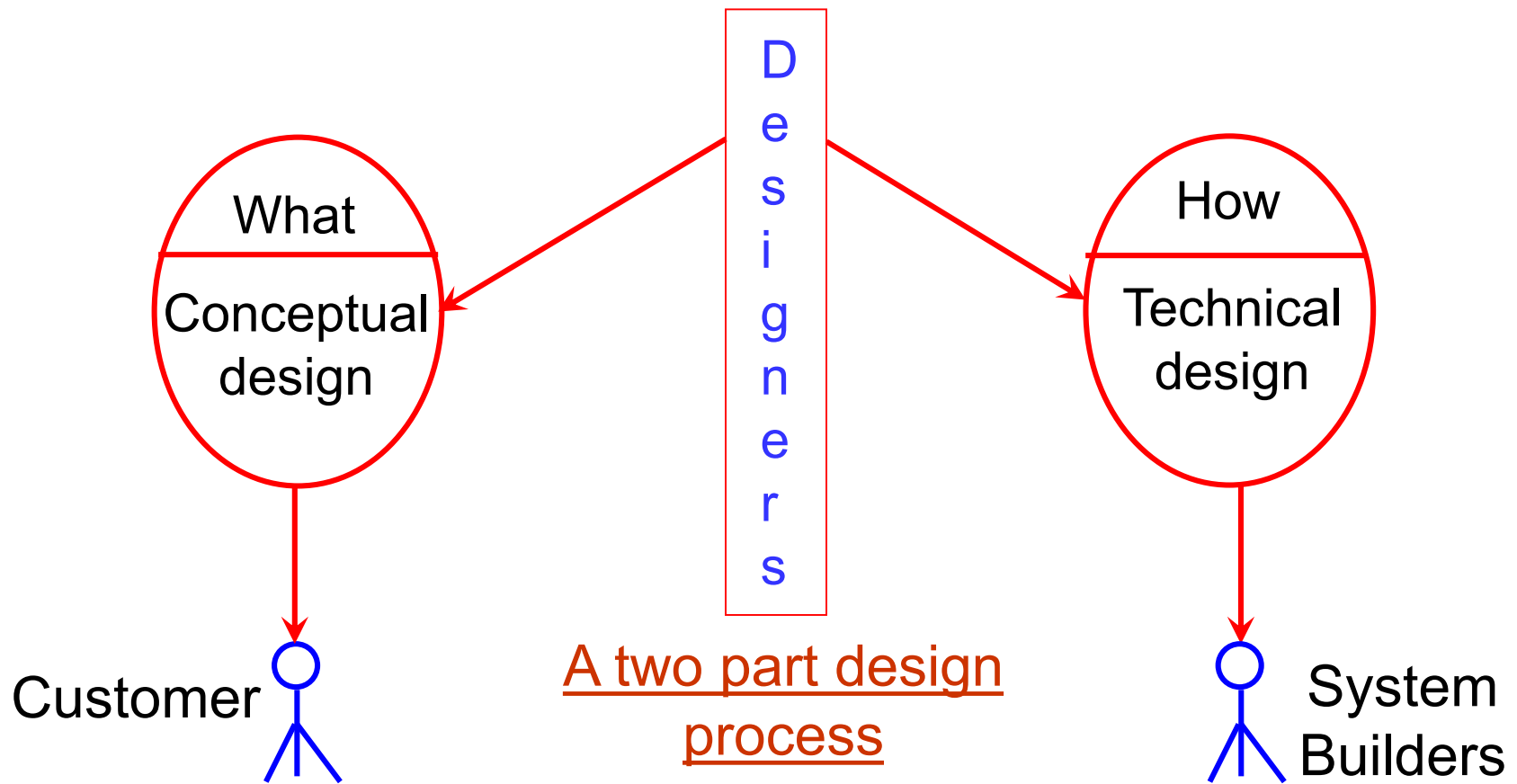


Fig. 2 : A two part design process

# *Software Design*

---

## Conceptual design answers :

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

# *Software Design*

---

Technical design describes :

- ❖ Hardware configuration
- ❖ Software needs
- ❖ Communication interfaces
- ❖ I/O of the system
- ❖ Software architecture
- ❖ Network architecture
- ❖ Any other thing that translates the requirements in to a solution to the customer's problem.

# *Software Design*

---

The design needs to be

- Correct & complete
- Understandable
- At the right level
- Maintainable



# *Software Design*

---

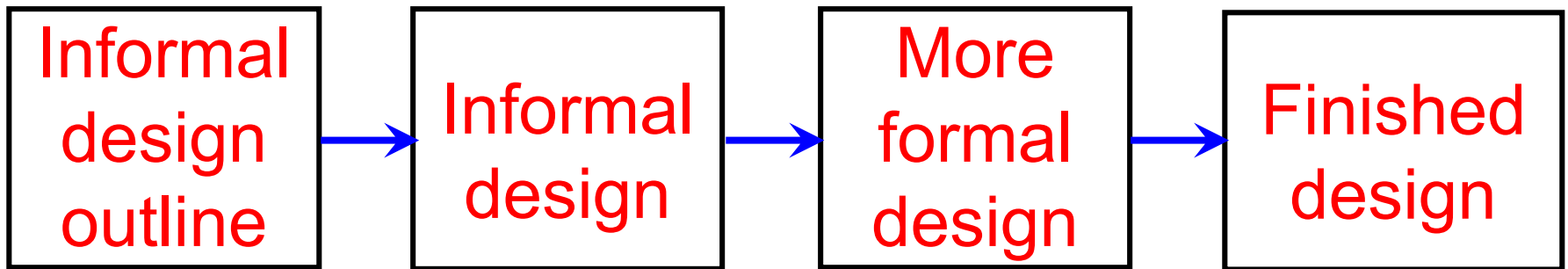


Fig. 3 : The transformation of an informal design to a detailed design.

# *Software Design*

---

## **MODULARITY**

There are many definitions of the term module. Range is from :

- i. Fortran subroutine
- ii. Ada package
- iii. Procedures & functions of PASCAL & C
- iv. C++ / Java classes
- v. Java packages
- vi. Work assignment for an individual programmer

# *Software Design*

---

All these definitions are correct. A modular system consist of well defined manageable units with well defined interfaces among the units.

# *Software Design*

---

## Properties :

- i. Well defined subsystem
- ii. Well defined purpose
- iii. Can be separately compiled and stored in a library.
- iv. Module can use other modules
- v. Module should be easier to use than to build
- vi. Simpler from outside than from the inside.

# *Software Design*

---

Modularity is the single attribute of software that allows a program to be intellectually manageable.

It enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of software product.

# Software Design

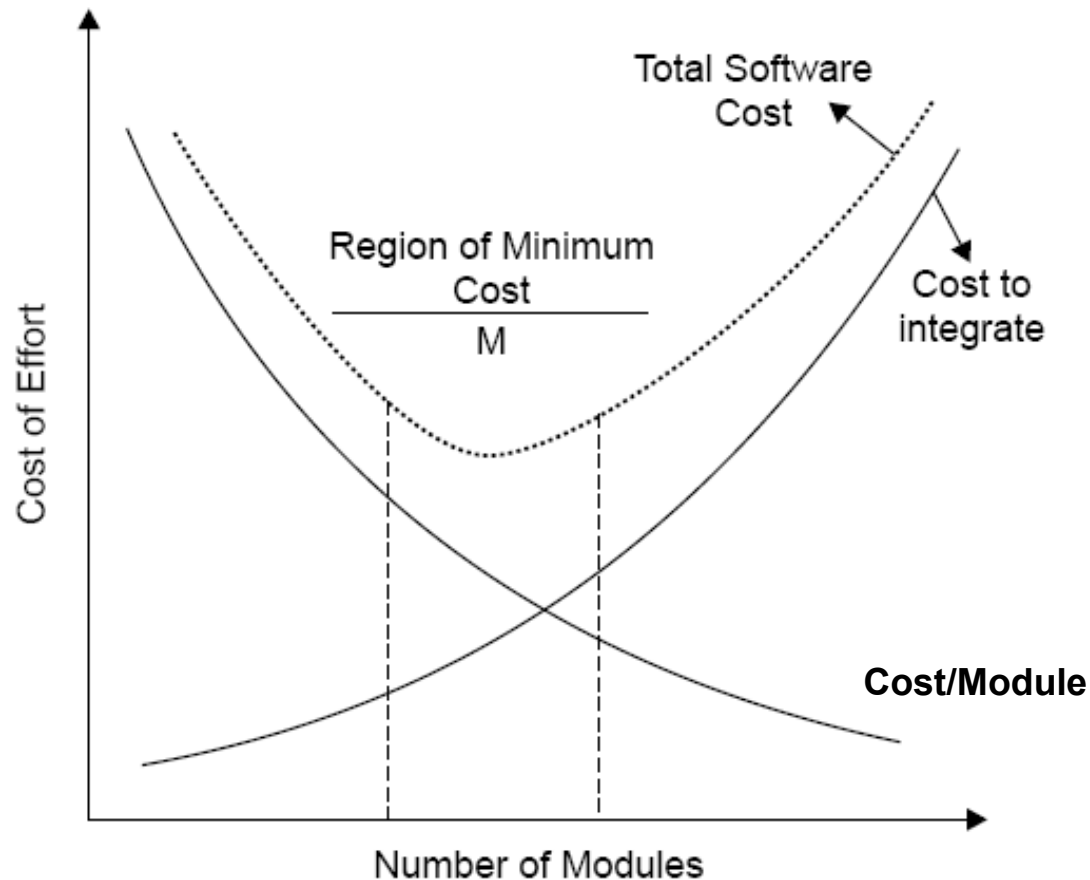


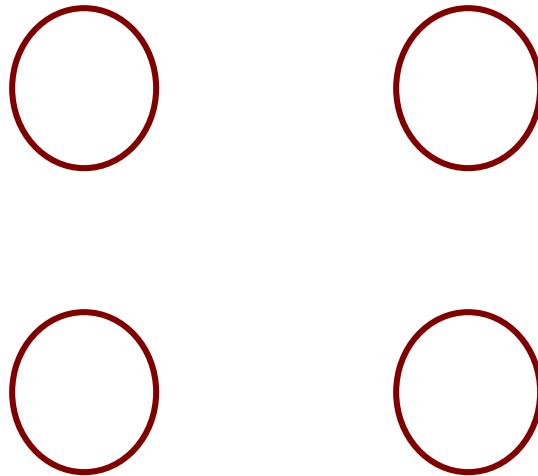
Fig. 4 : Modularity and software cost

# *Software Design*

---

## Module Coupling

Coupling is the measure of the degree of interdependence between modules.

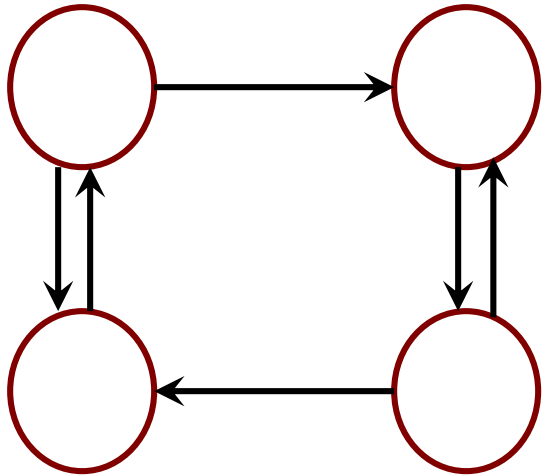


(Uncoupled : no dependencies)

(a)

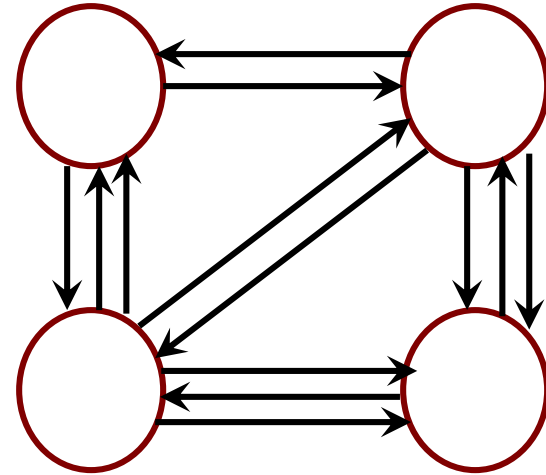
# Software Design

---



Loosely coupled:  
some dependencies

(B)



Highly coupled:  
many dependencies

(C)

Fig. 5 : Module coupling



# *Software Design*

---

This can be achieved as:

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

# Software Design

Consider the example of editing a student record in a 'student information system'.

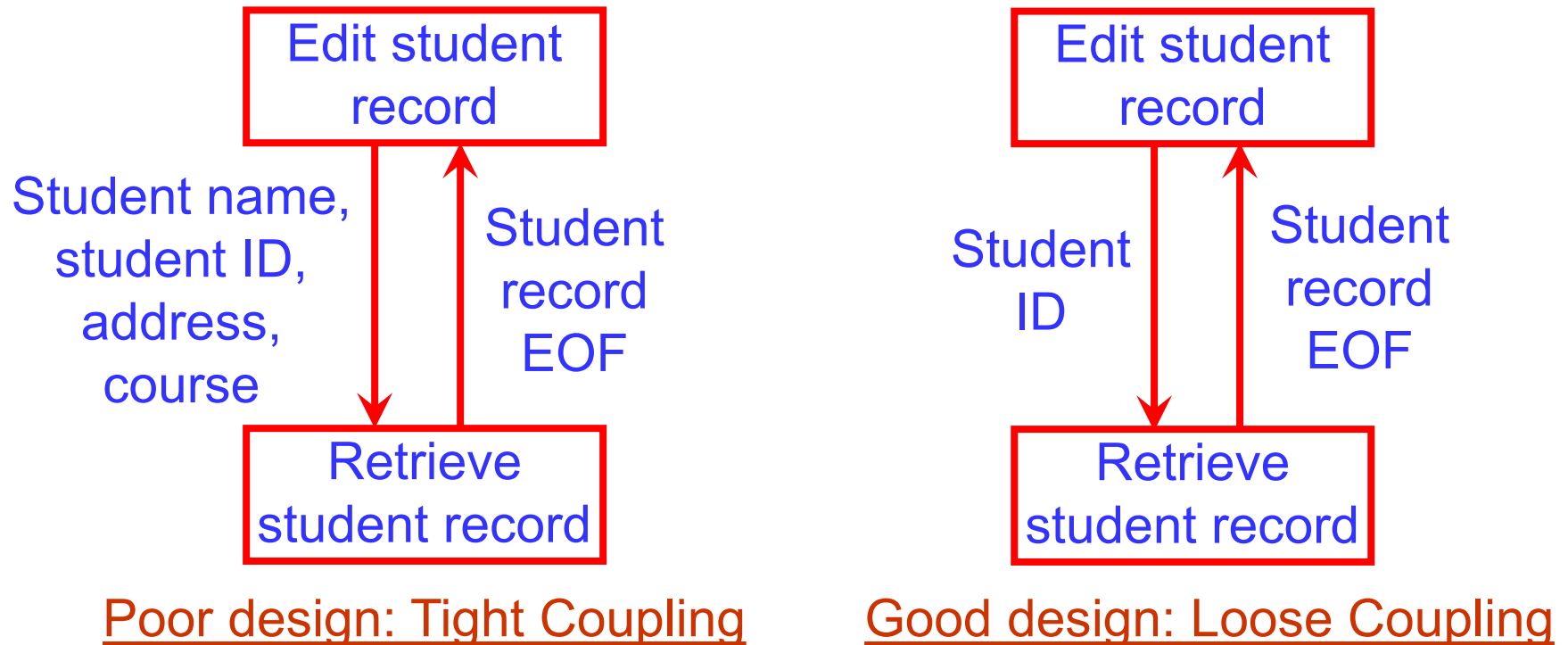


Fig. 6 : Example of coupling

# Software Design

---

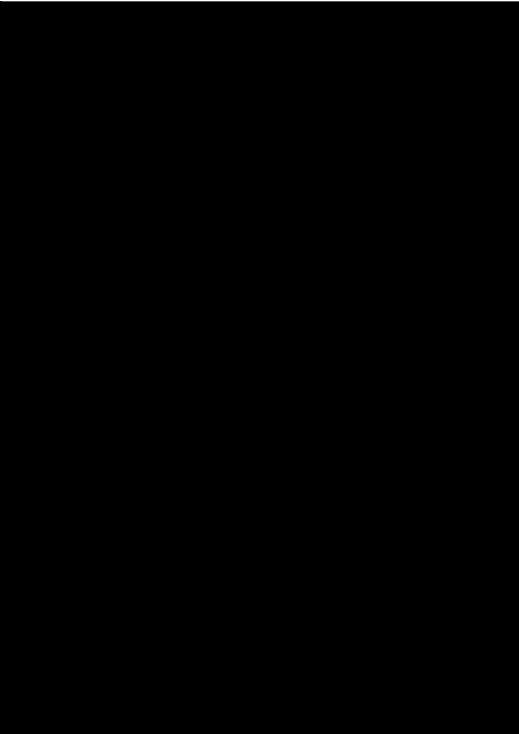
Data coupling	
Stamp coupling	
Control coupling	
External coupling	
Common coupling	
Content coupling	

Fig. 7 : The types of module coupling

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

# *Software Design*

---

## **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.

## **Stamp coupling**

Stamp coupling occurs between module A and B when complete data structure is passed from one module to another.

# *Software Design*

---

## **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.

# *Software Design*

---

## **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.

# *Software Design*

---

## **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 changes.

# *Software Design*

---

## **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. 9, module B branches into D, even though D is supposed to be under the control of C.



# Software Design

---

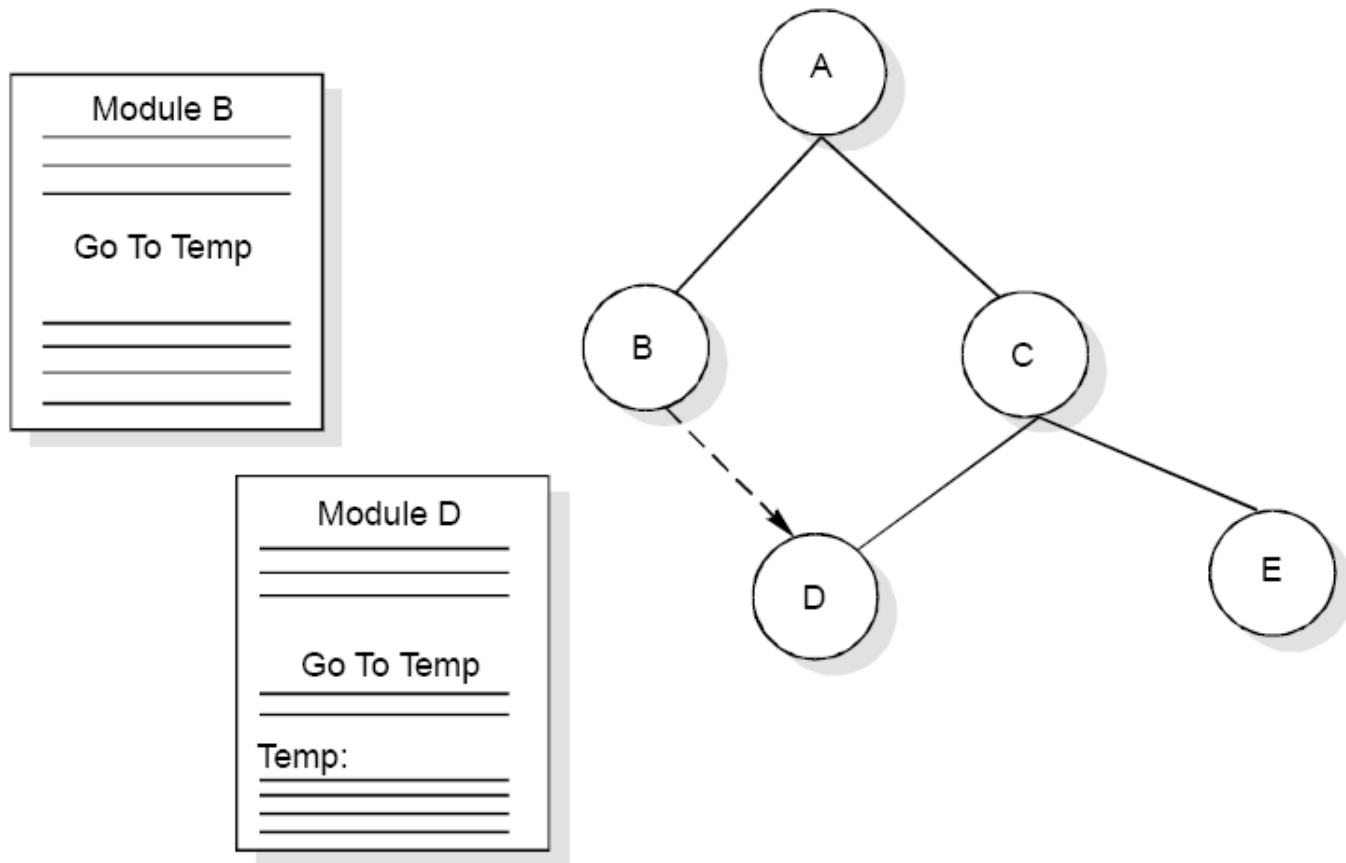


Fig. 9 : Example of content coupling

# Software Design

---


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

Fig. 7 : The types of module coupling

# Software Design

## Module Cohesion

Cohesion is a measure of the degree to which the elements of a module are functionally related.

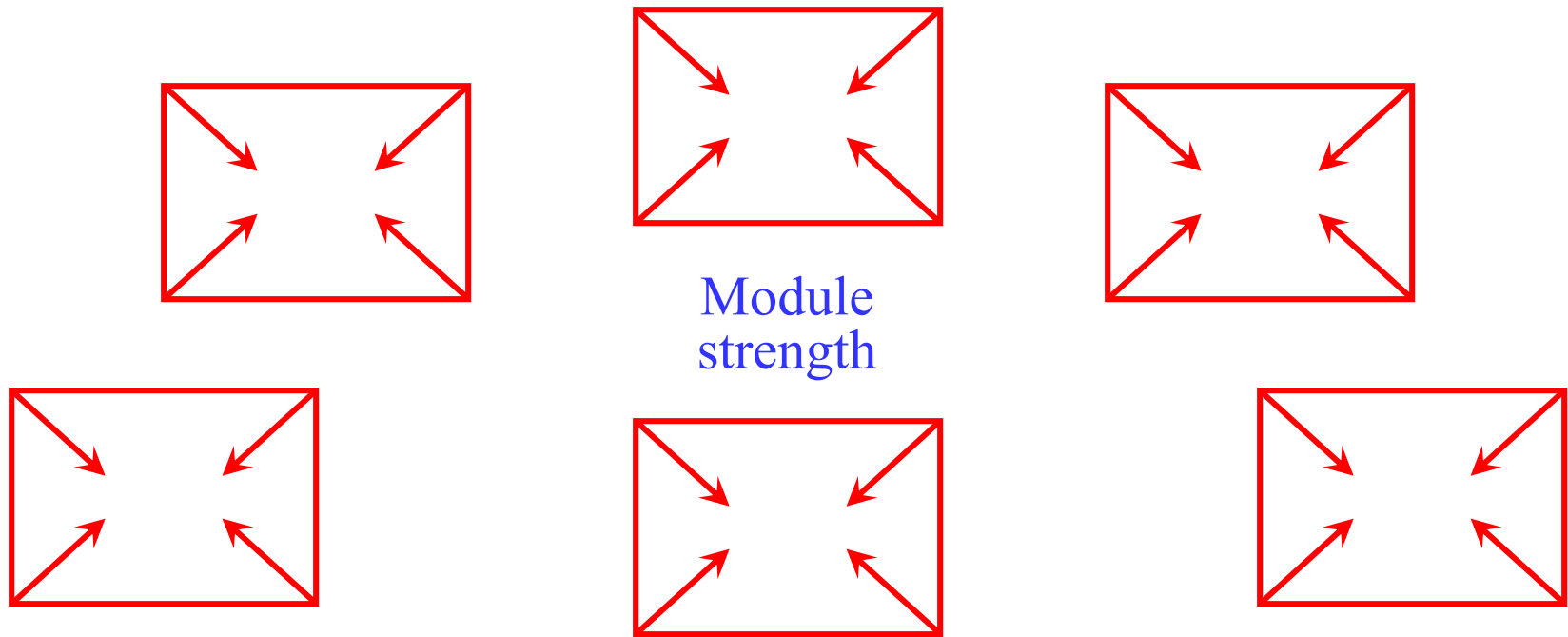


Fig. 10 : Cohesion=Strength of relations within modules

# *Software Design*

---

## **Types of cohesion**

- Functional cohesion
- Sequential cohesion
- Procedural cohesion
- Communicational cohesion
- Temporal cohesion
- Logical cohesion
- Coincident cohesion

# Software Design

---

Functional Cohesion	
Sequential Cohesion	
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	

Fig. 11 : Types of module cohesion

# *Software Design*

---

## **Functional Cohesion**

- A and B are part of a single functional task. This is very good reason for them to be contained in the same procedure.

## **Sequential Cohesion**

- Element A outputs some data which forms the input to B. This is the reason for them to be contained in the same procedure.

## **Communicational Cohesion**

- Elements are together because they operate on the same input or output data.

# *Software Design*

---

## **Procedural Cohesion**

- 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.

## **Temporal Cohesion**

- 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.

# *Software Design*

---

## **Logical Cohesion**

- Logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions.

## **Coincidental Cohesion**

- Coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another.



# Software Design

---


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

Fig. 11 : Types of module cohesion

# Software Design

## Relationship between Cohesion & Coupling

If the software is not properly modularized, a host of seemingly trivial enhancement or changes will result into death of the project. Therefore, a software engineer must design the modules with goal of high cohesion and low coupling.

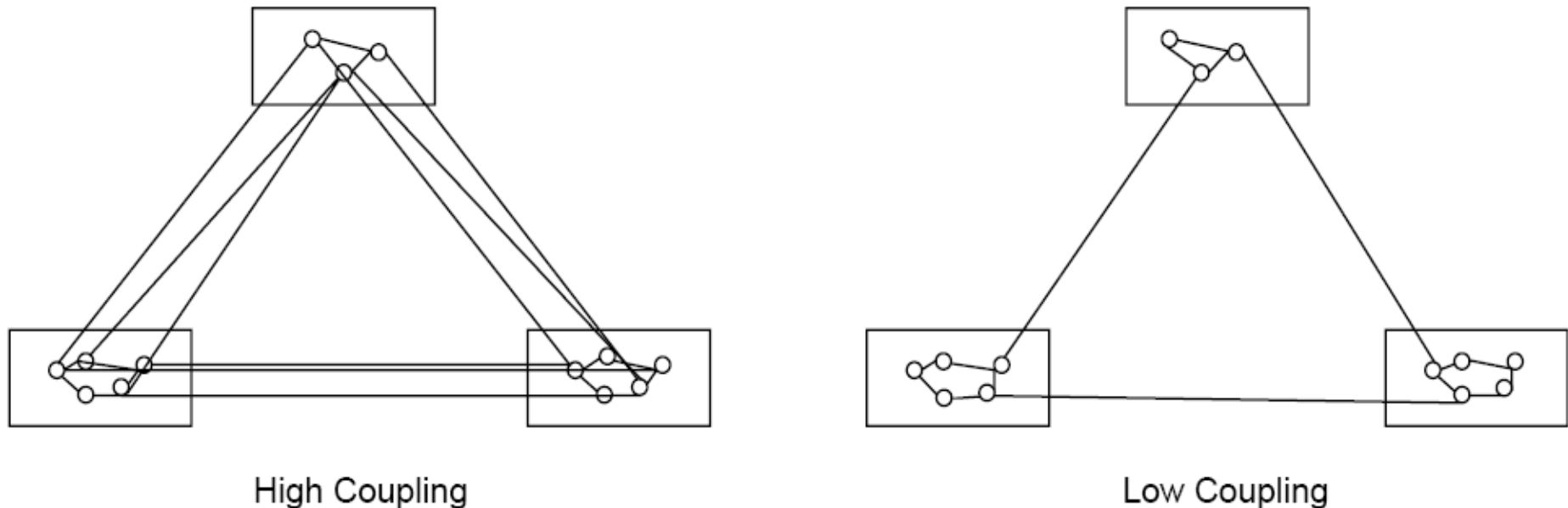


Fig. 12 : View of cohesion and coupling

# *Software Design*

---

## **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.

# *Software Design*

---

## **Top-Down Design**

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.

# Software Design

---

## Bottom-Up Design

These modules are collected together in the form of a “library”.

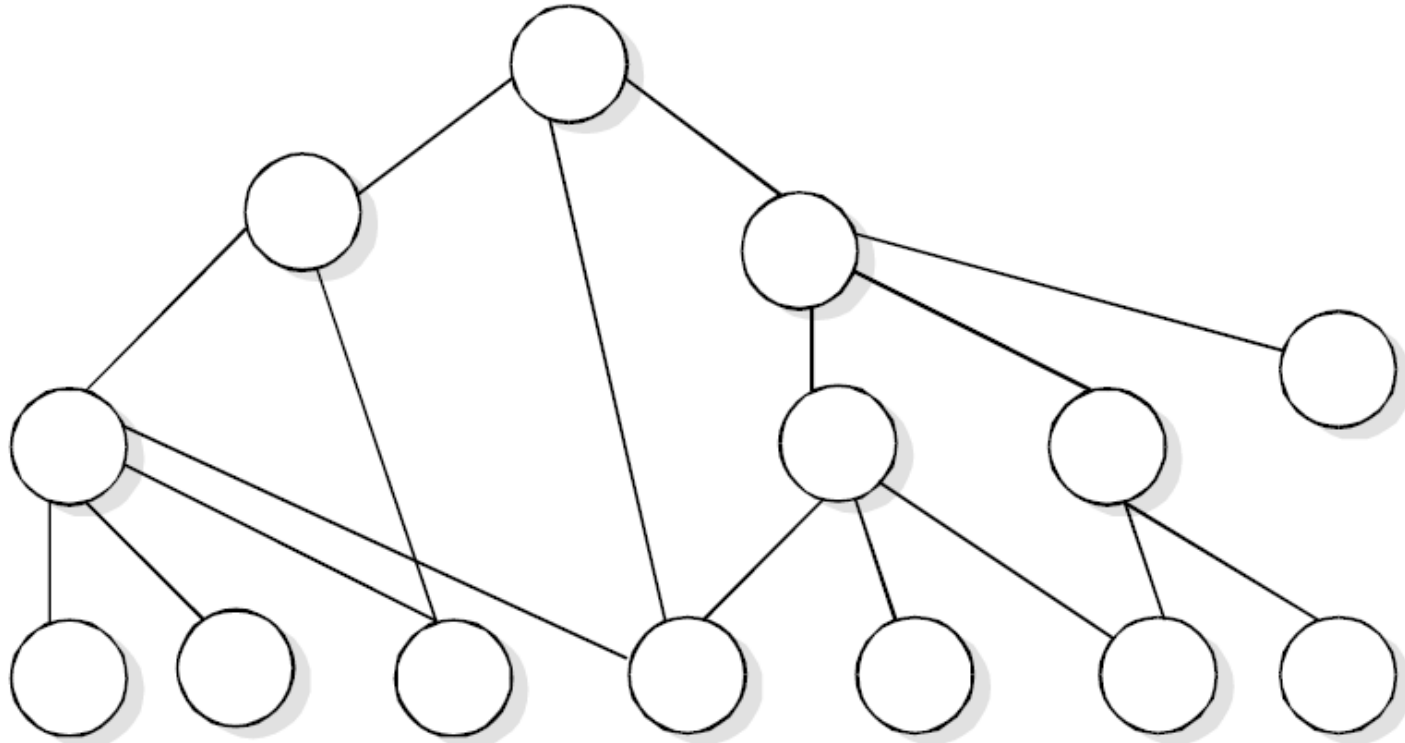


Fig. 13 : Bottom-up tree structure

# *Software Design*

---

## **Hybrid Design**

For top-down approach to be effective, some bottom-up 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 number of modules at low levels than high levels.
- In the use of pre-written library modules, in particular, reuse of modules.

# *Software Design*

---

## **FUNCTION ORIENTED DESIGN**

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.

# Software Design

---

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  $\rightarrow$  type)  
    Case integer: /*print an integer*/  
    Case real:   /*print a real*/  
    Case list:   /*print a list*/  
    ...  
}
```



# Software Design

---

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 refinement as in design top-down structure as shown in fig. 14.

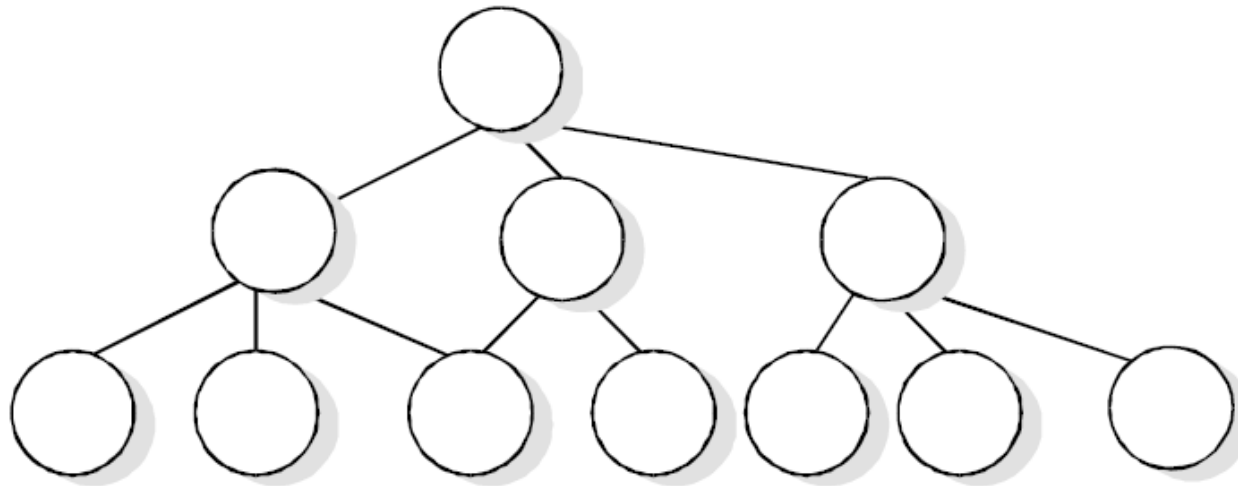


Fig. 14 : Top-down structure

# *Software Design*

---

## **Object 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 and behavior.

# *Software Design*

---

## ➤ Basic Concepts

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

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

# *Software Design*

---

The various terms related to object design are:

## 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 (behavior) to either examine or affect this state. An object is characterized by number of operations and a state which remembers the effect of these operations.

# *Software Design*

---

## ii. Messages

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. Messages are often implemented as procedure or function calls.

## iii. Abstraction

In object oriented design, complexity is managed using abstraction. Abstraction is the elimination of the irrelevant and the amplification of the essentials.

# *Software Design*

---

## 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 is known as a class. Hence, a class is a set of objects that share a common structure and a common behavior.

We may define a class “car” and each object that represent 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. 20.

Classes are useful because they act as a blueprint 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. 21 shows how can we represent the square class.

# Software Design

---

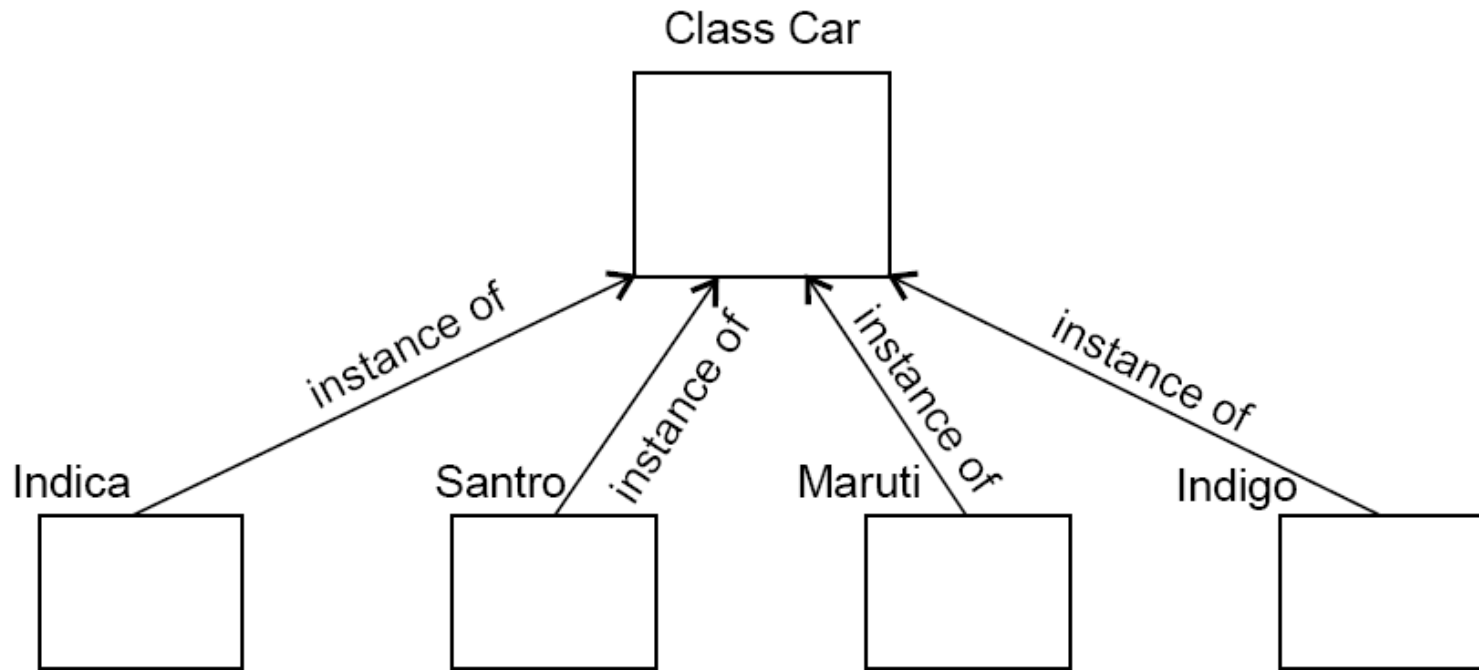


Fig.20: Indica, Santro, Maruti, Indigo are all instances of the class “car”

# Software Design

---

Class Square

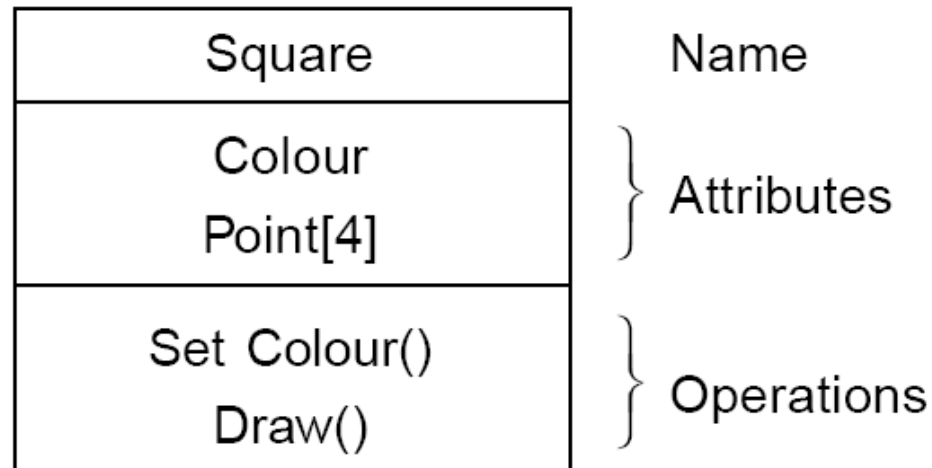


Fig. 21: The square class



# *Software Design*

---

## v. Attributes

An attributes is a data value held by the objects in a class. The square class has two attributes: a colour and array of points. Each attributes has a value for each object instance. The attributes are shown as second part of the class as shown in fig. 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. An object “knows” its class, and hence the right implementation of the operation. Operation are shown in the third part of the class as indicated in fig. 21.

# Software Design

---

## vii. Inheritance

Imagine that, as well as squares, we have triangle class. Fig. 22 shows the class for a triangle.

Class Triangle

Triangle
Colour Point[3]
Set Colour() Draw()

Fig. 22: The triangle class

# *Software Design*

---

Now, comparing fig. 21 and 22, we can see that there is some difference between triangle and squares classes.

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. 23 shows the results.

# Software Design

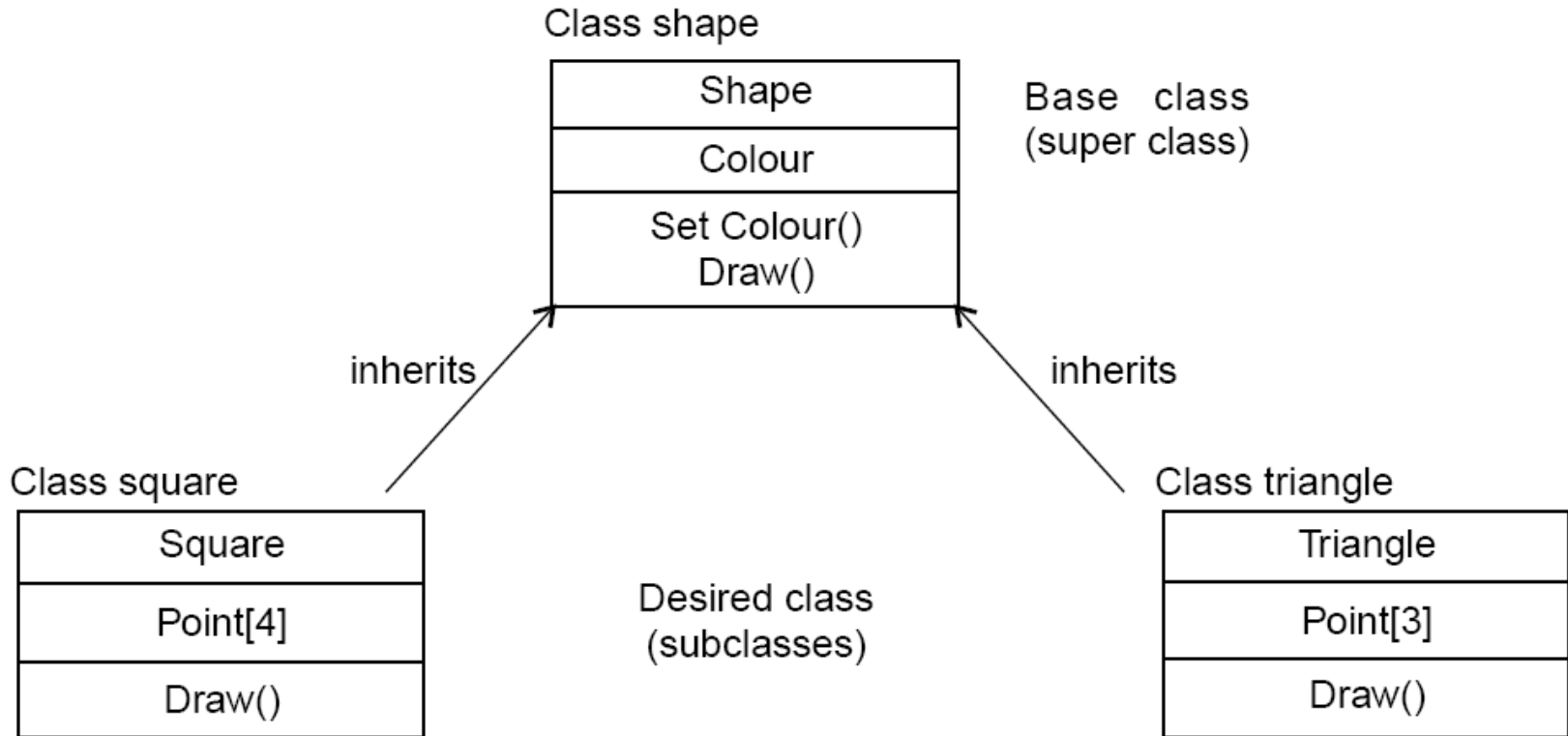


Fig. 23: Abstracting common features in a new class

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

# Software Design

---

## 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.

## 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.

## x. Hierarchy

Hierarchy involves organizing something according to some particular order or rank. It is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way.

# Software Design

---

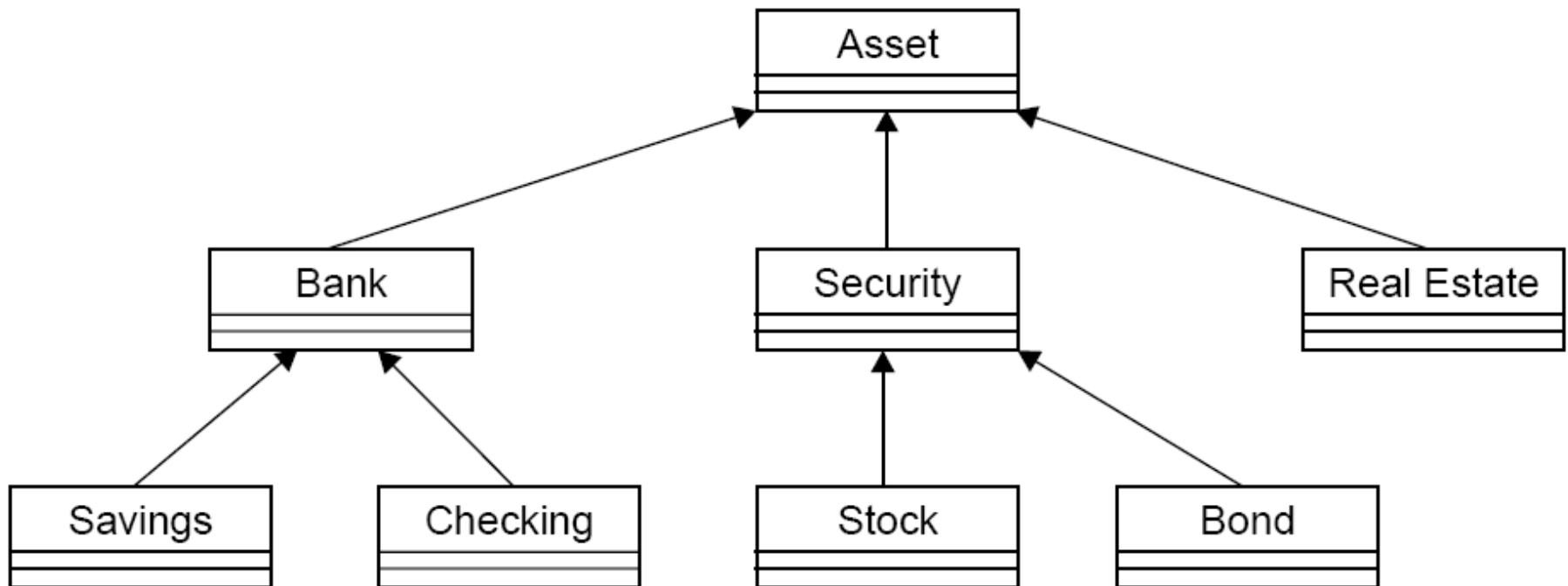


Fig. 24: Hierarchy

## **Design Notations**

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**

# Software Design

## Structure Chart

It partition a system into black boxes. A black box means that functionality is known to the user without the knowledge of internal design.

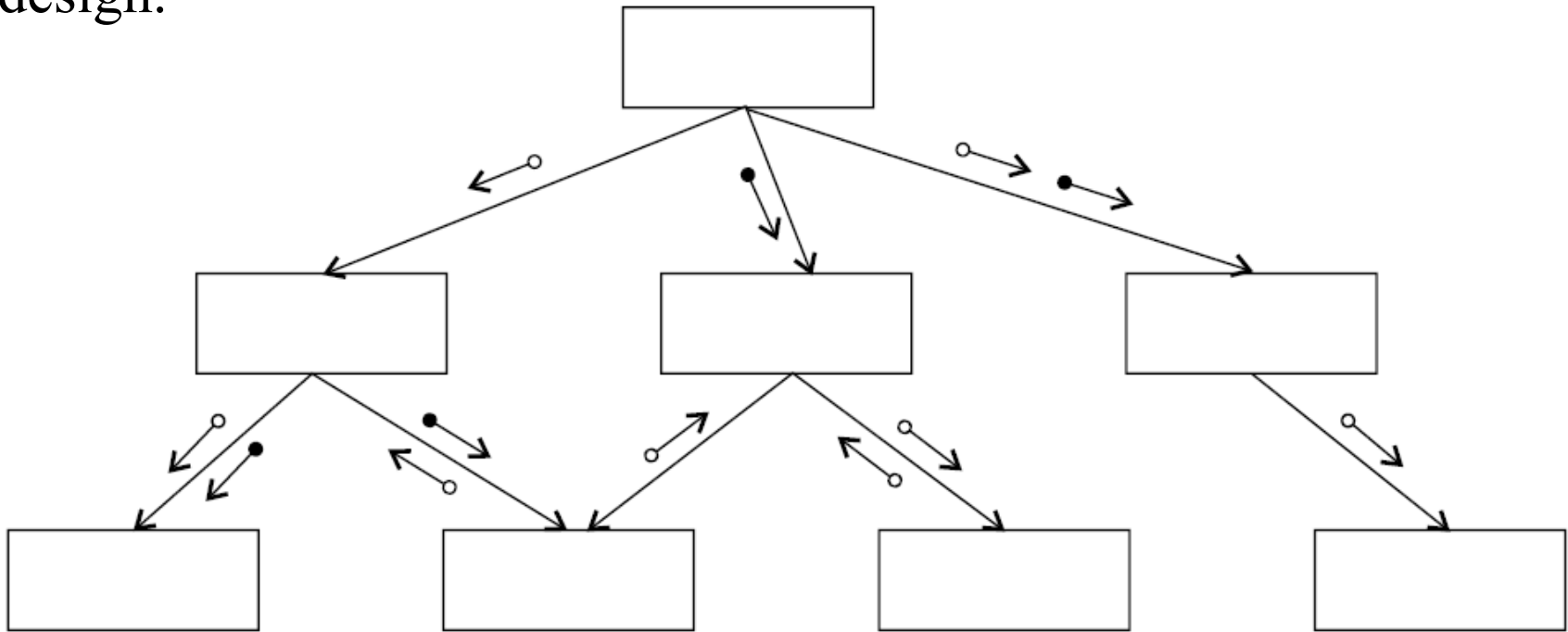


Fig. 16 : Hierarchical format of a structure chart



# Software Design

---

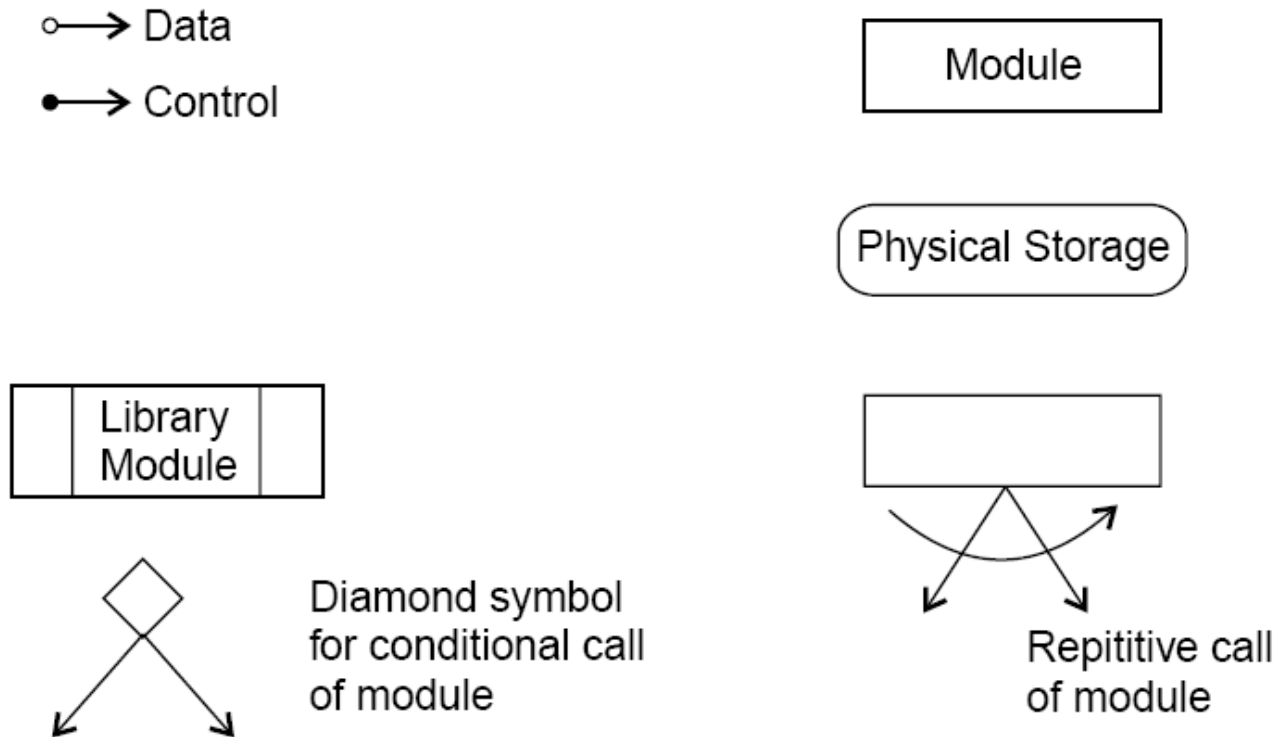


Fig. 17 : Structure chart notations

# Software Design

A structure chart for “update file” is given in fig. 18.

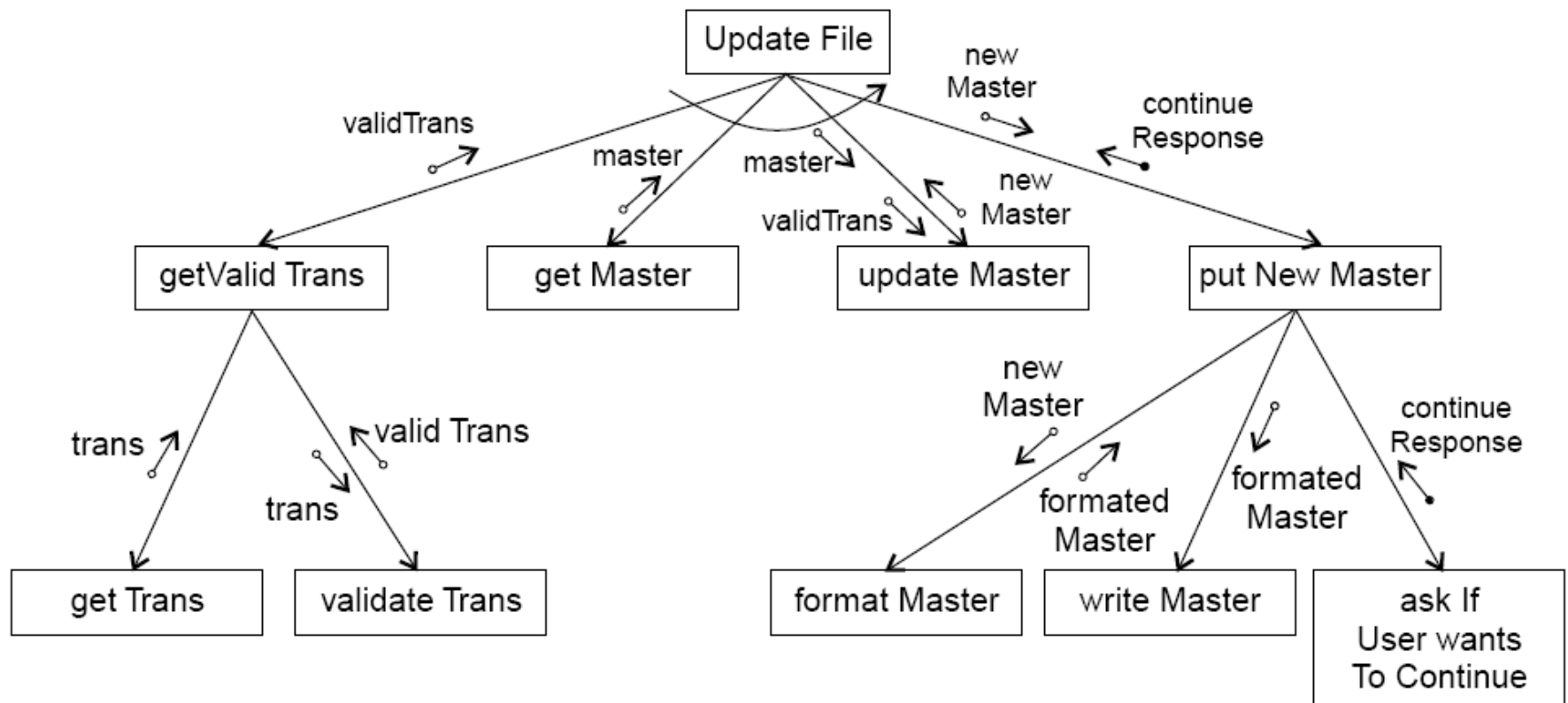


Fig. 18 : Update file

# Software Design

---

A transaction centered structure describes a system that processes a number of different types of transactions. It is illustrated in Fig.19.

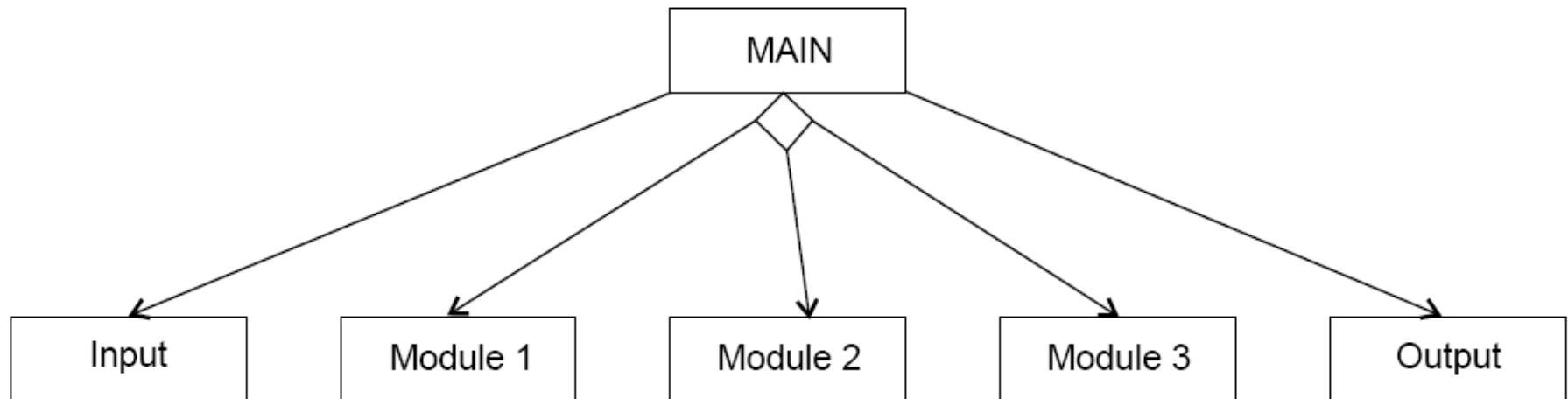


Fig. 19 : Transaction-centered structure

# Software Design

---

In the above figure the MAIN module controls the system operation its functions 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.

# *Software Design*

---

## **Pseudocode**

Pseudocode notation can be used in both the preliminary and detailed design phases.

Using pseudocode, the designer describes system characteristics using short, concise, English language phrases that are structured by key words such as It-Then-Else, While-Do, and End.

# *Software Design*

---

## Pseudocode

### **Do's :**

- . Use control structures
- . Use proper naming convention
- . Indentation and white spaces are the key
- . Keep it simple.
- . Keep it concise.

### **Don'ts :**

- . Don't make the pseudo code abstract.
- . Don't be too generalized.
- .

# *Software Design*

---

## Pseudocode Examples

```
Get first entry;  
Call this entry N;  
WHILE N is NOT the required entry  
DO Get next entry;  
    Call this entry N;  
ENDWHILE;
```

Enter value

if value greater than 10

say "Your number is greater than 10"

if value less than 10

say "Your number is less t

# *Software Design*

---

## ➤ 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 structure, software components, interfaces, and data necessary for the implementation phase. Hence, SDD becomes the blue print for the implementation activity.