

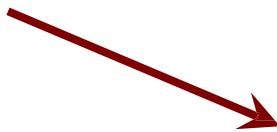
Software Design



-
- ❖ More creative than analysis
 - ❖ Problem solving activity

WHAT IS DESIGN

‘HOW’



Software design document (SDD)

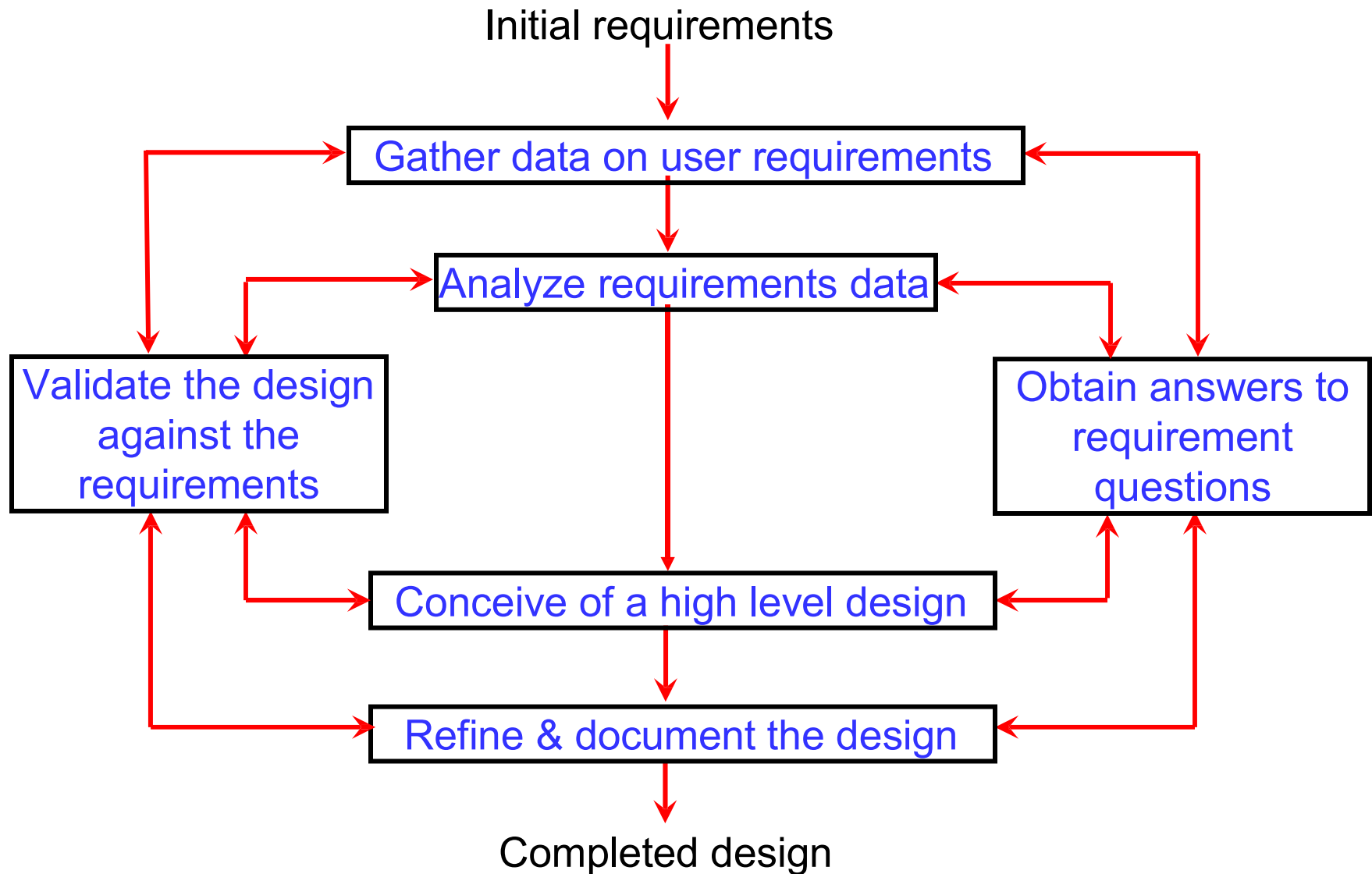


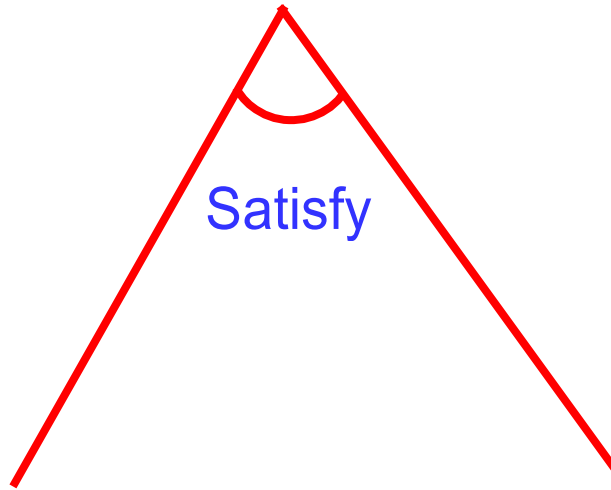
Fig. 1 : Design framework

design

Satisfy

Customer

Developers
(Implementers)



Conceptual Design and Technical Design

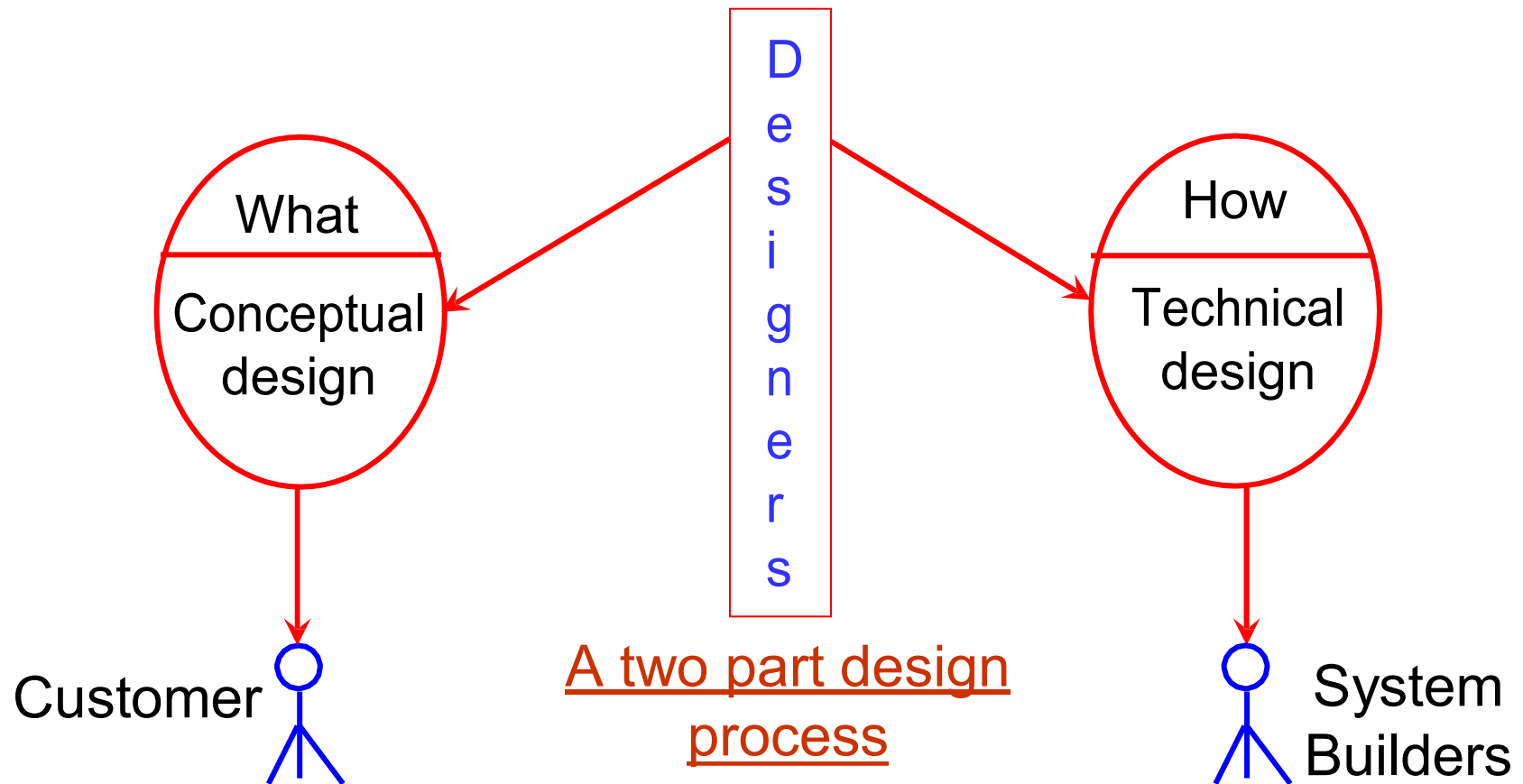


Fig. 2 : A two part design process

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?

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.

The design needs to be

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

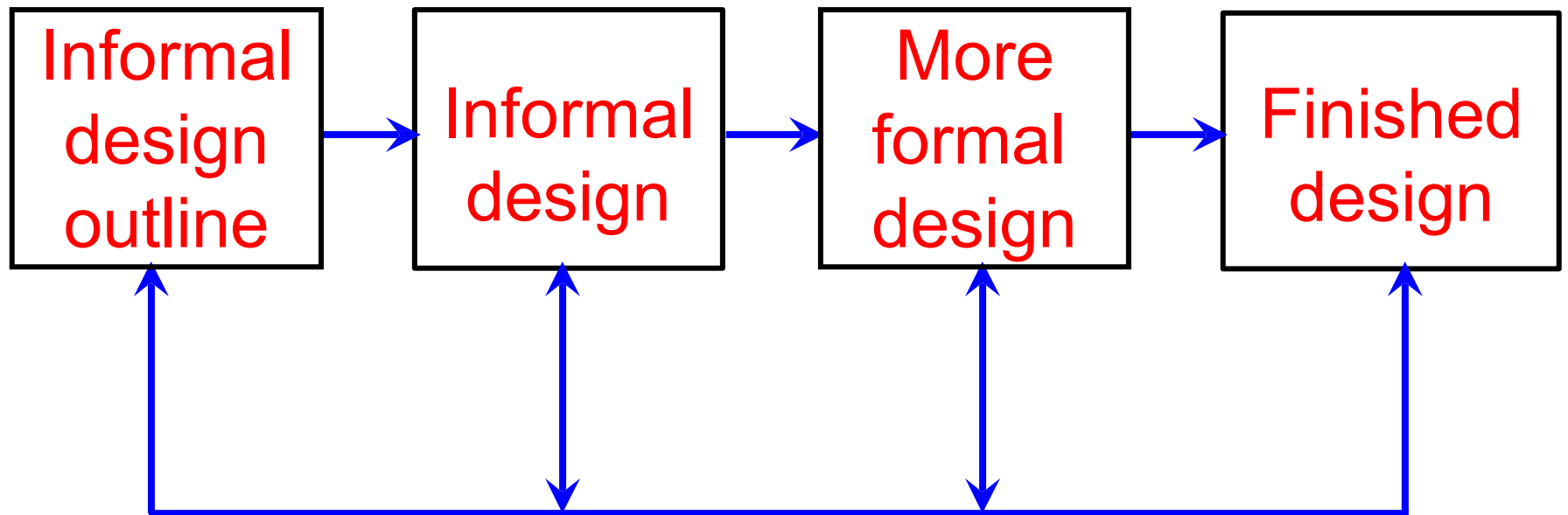


Fig. 3 : The transformation of an informal design to a detailed design.

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

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

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.

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.

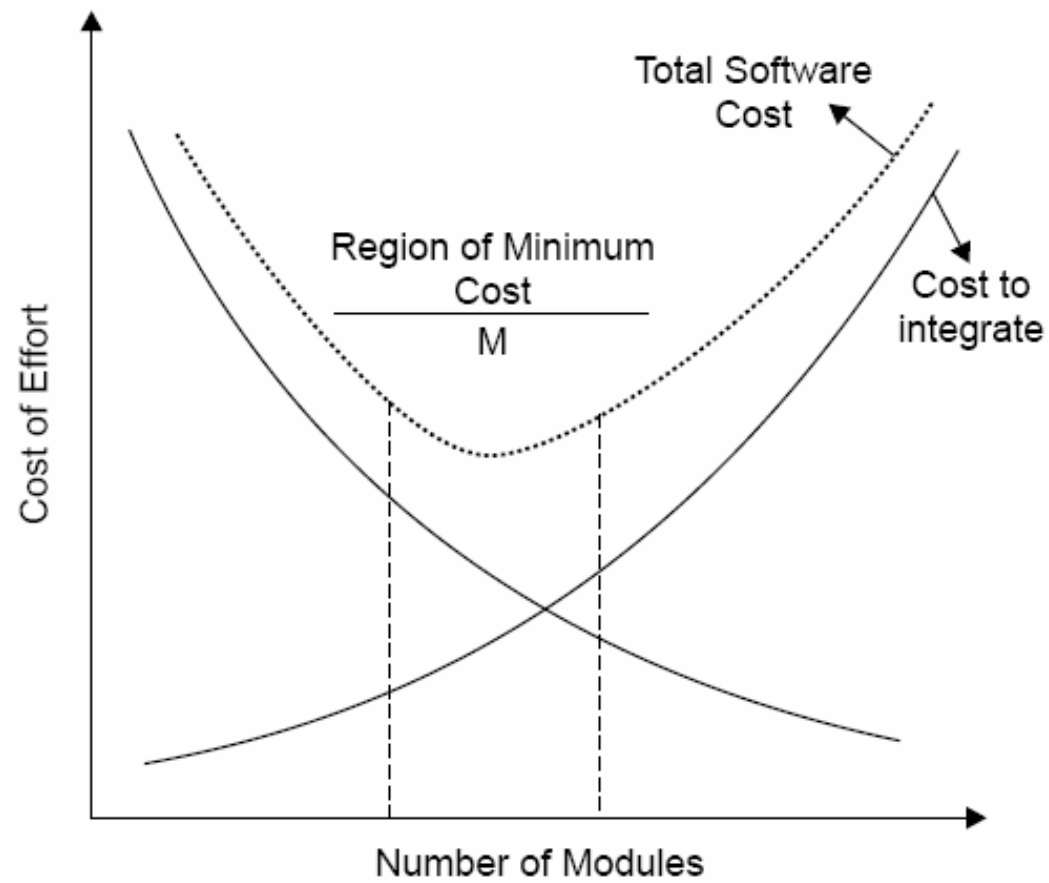
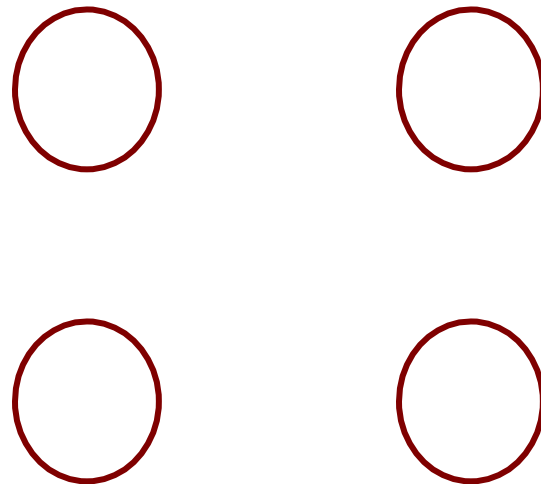


Fig. 4 : Modularity and software cost

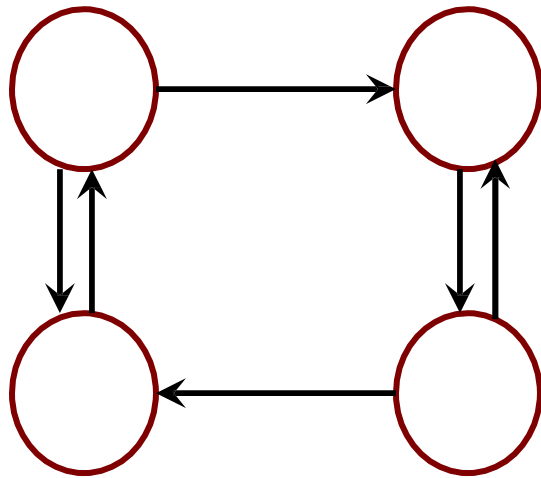
Module Coupling

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



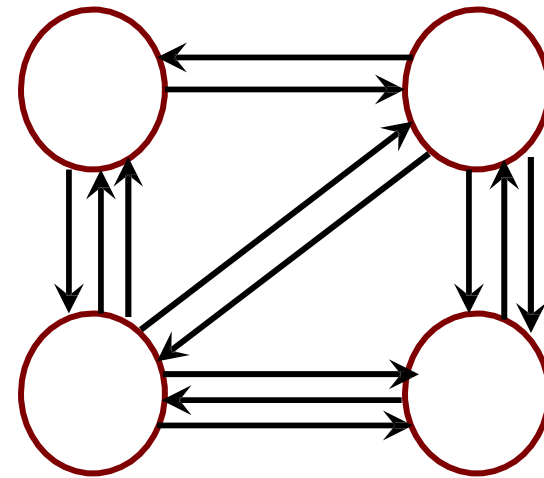
(Uncoupled : no dependencies)

(a)



Loosely coupled:
some dependencies

(B)



Highly coupled:
many dependencies

(C)

Fig. 5 : Module coupling

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.

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

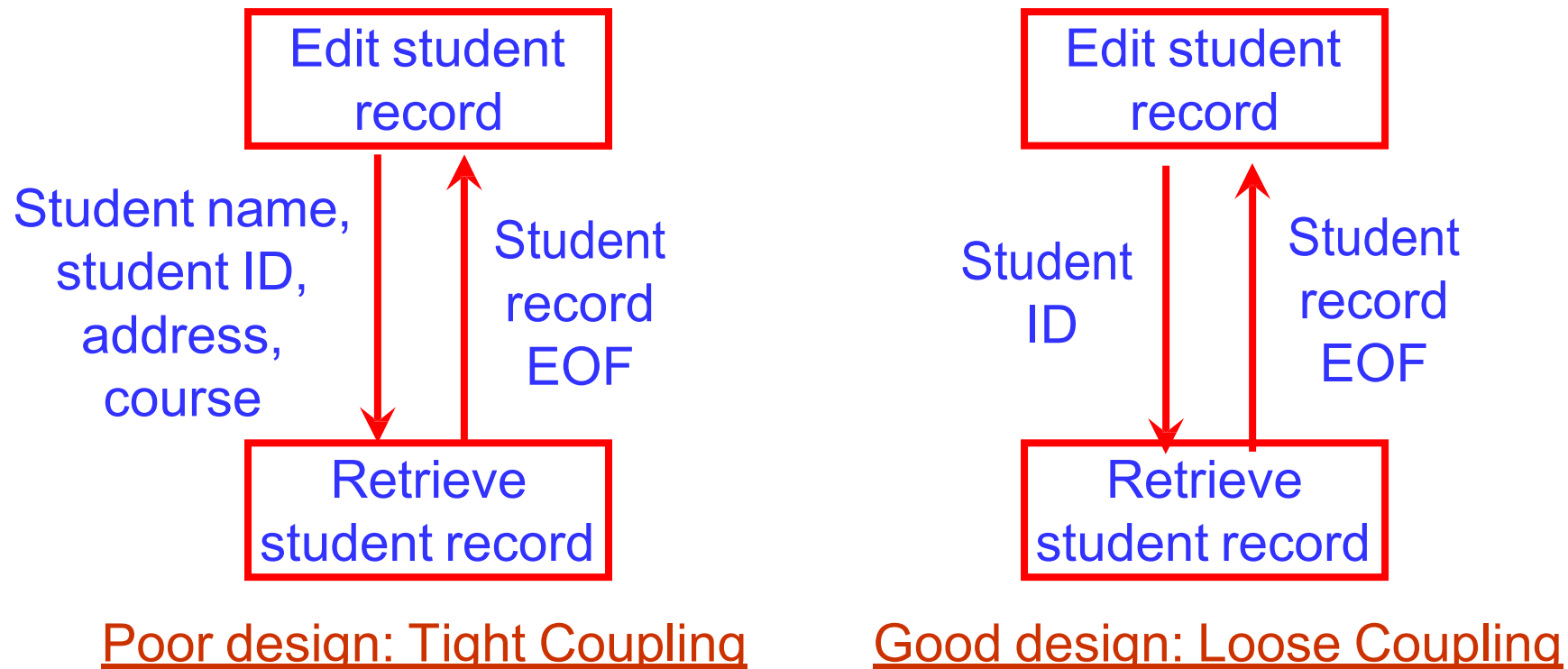


Fig. 6 : Example of coupling

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

Fig. 7 : The types of module coupling

Given two procedures A & B, we can identify 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.

Stamp coupling

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

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.

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.

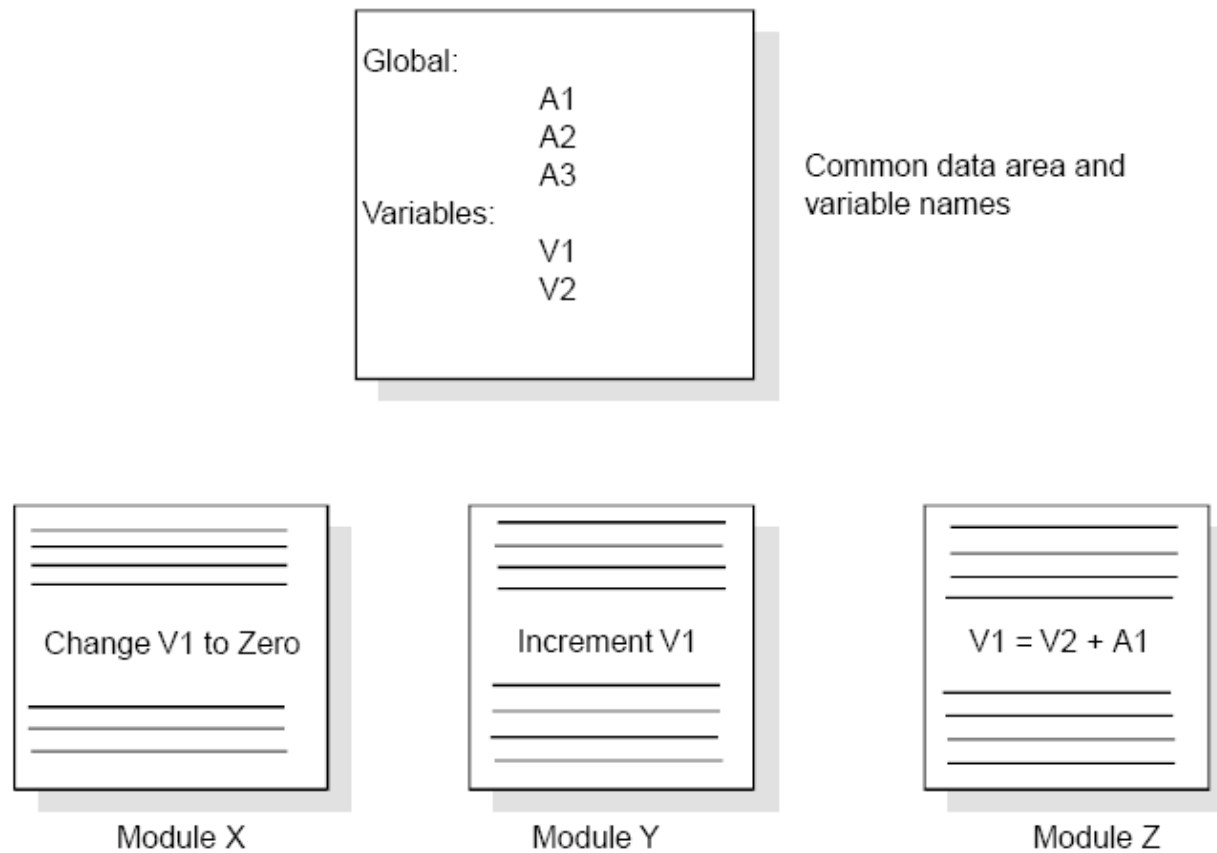


Fig. 8 : Example of common coupling

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

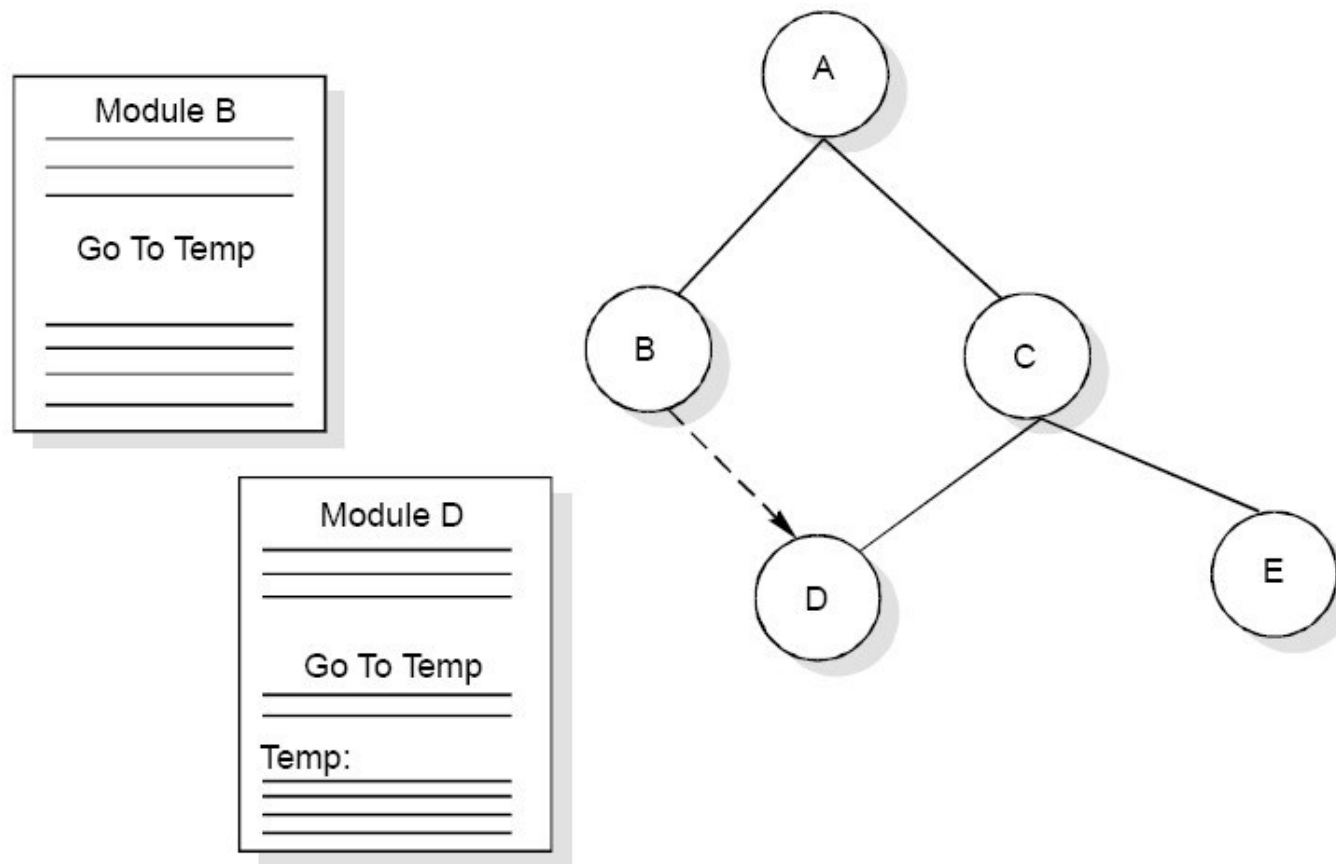


Fig. 9 : Example of content coupling

Module Cohesion

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

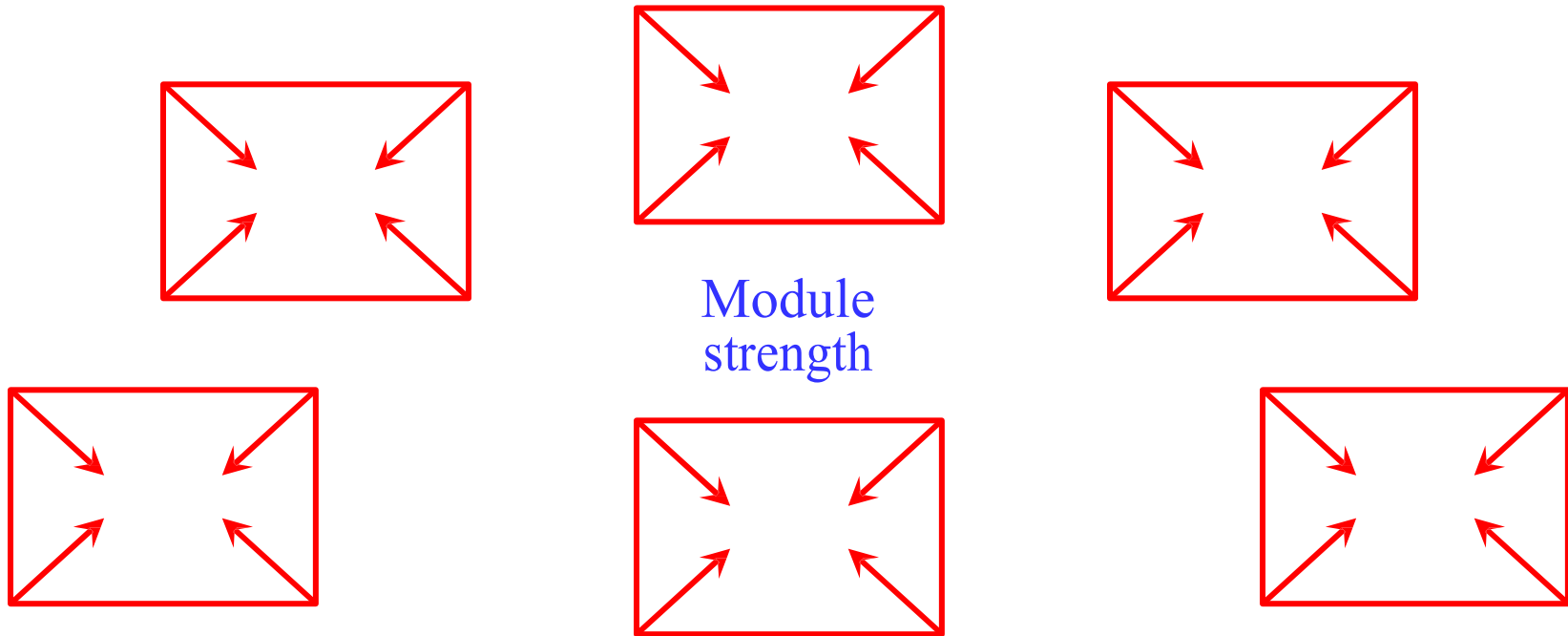


Fig. 10 : Cohesion=Strength of relations within modules

Types of cohesion

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


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

Fig. 11 : Types of module cohesion

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

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

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.

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.

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.

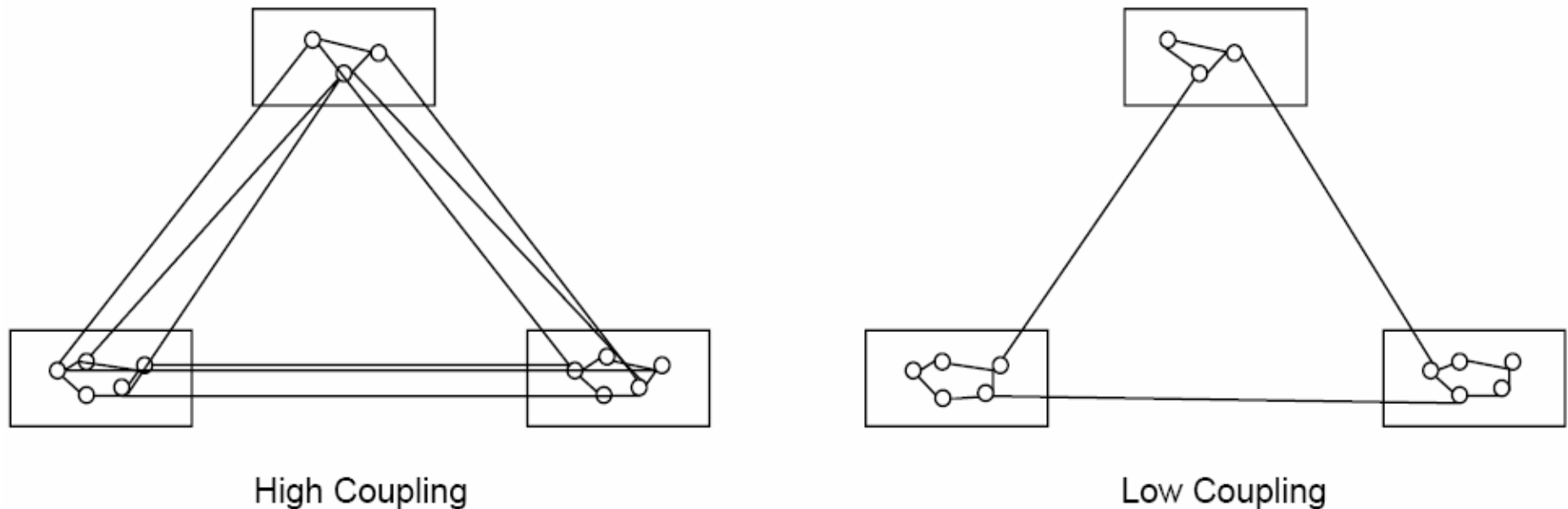


Fig. 12 : View of cohesion and coupling

STRATEGY OF DESIGN

A good system design strategy is to organize the program modules in such a way that are easy to develop and latter 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.

Bottom-Up Design

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

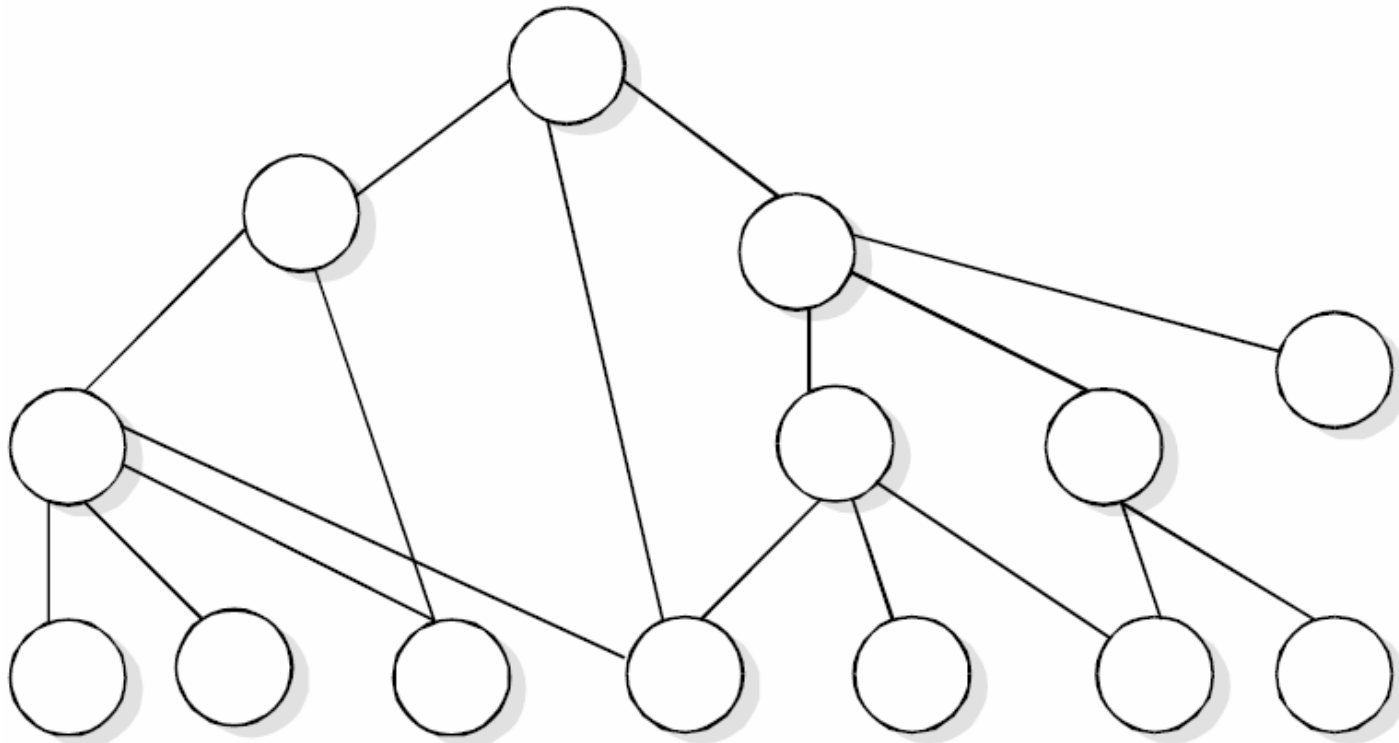


Fig. 13 : Bottom-up tree structure

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.

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.

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.

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*/  
    ...  
}
```

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.

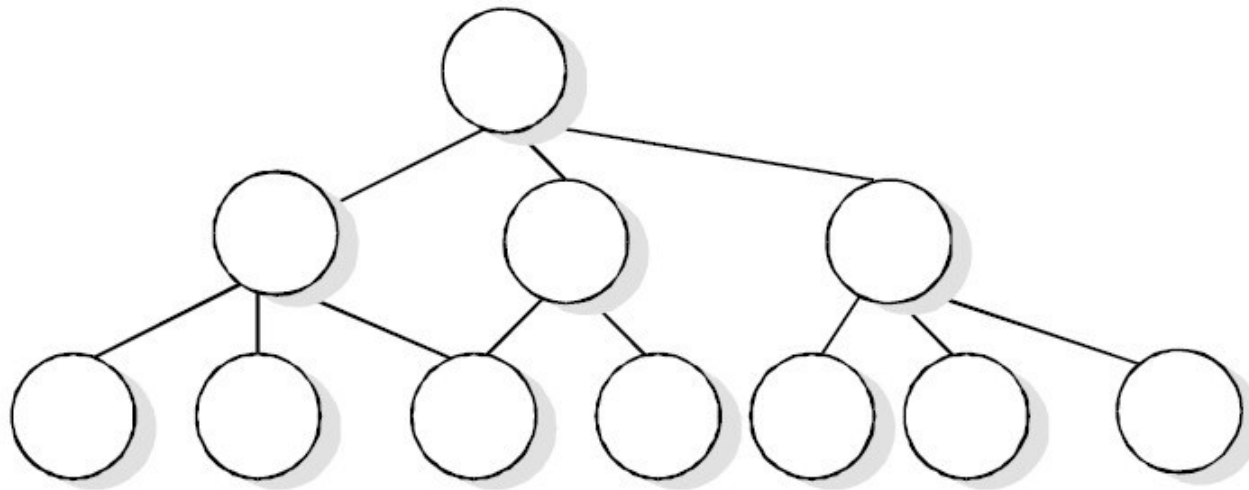


Fig. 14 : Top-down structure

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, however, we must require that several other modules as in design reusable structure as shown in fig. 15.

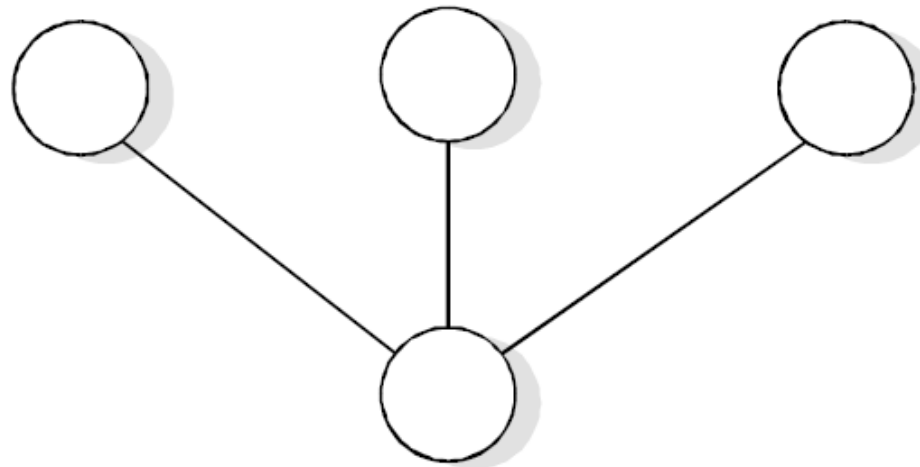


Fig. 15 : Design reusable structure

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

Structure Chart

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

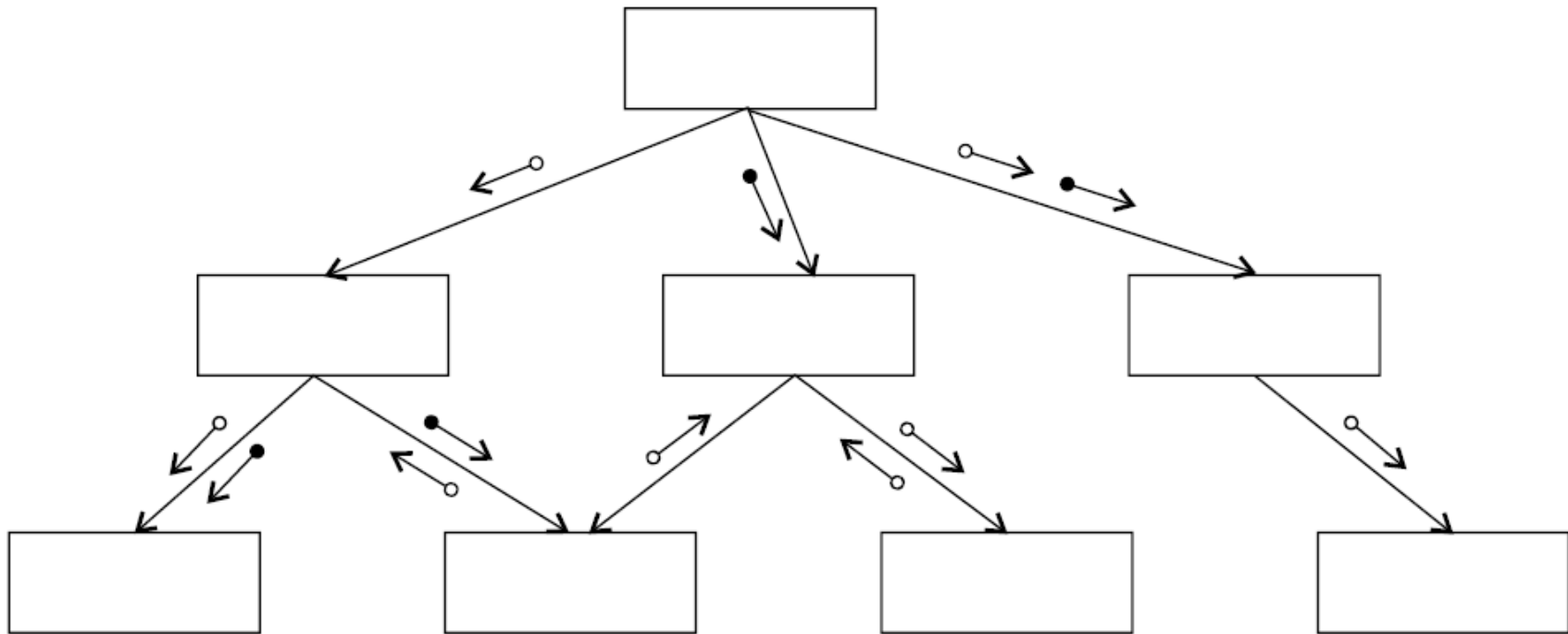


Fig. 16 : Hierarchical format of a structure chart

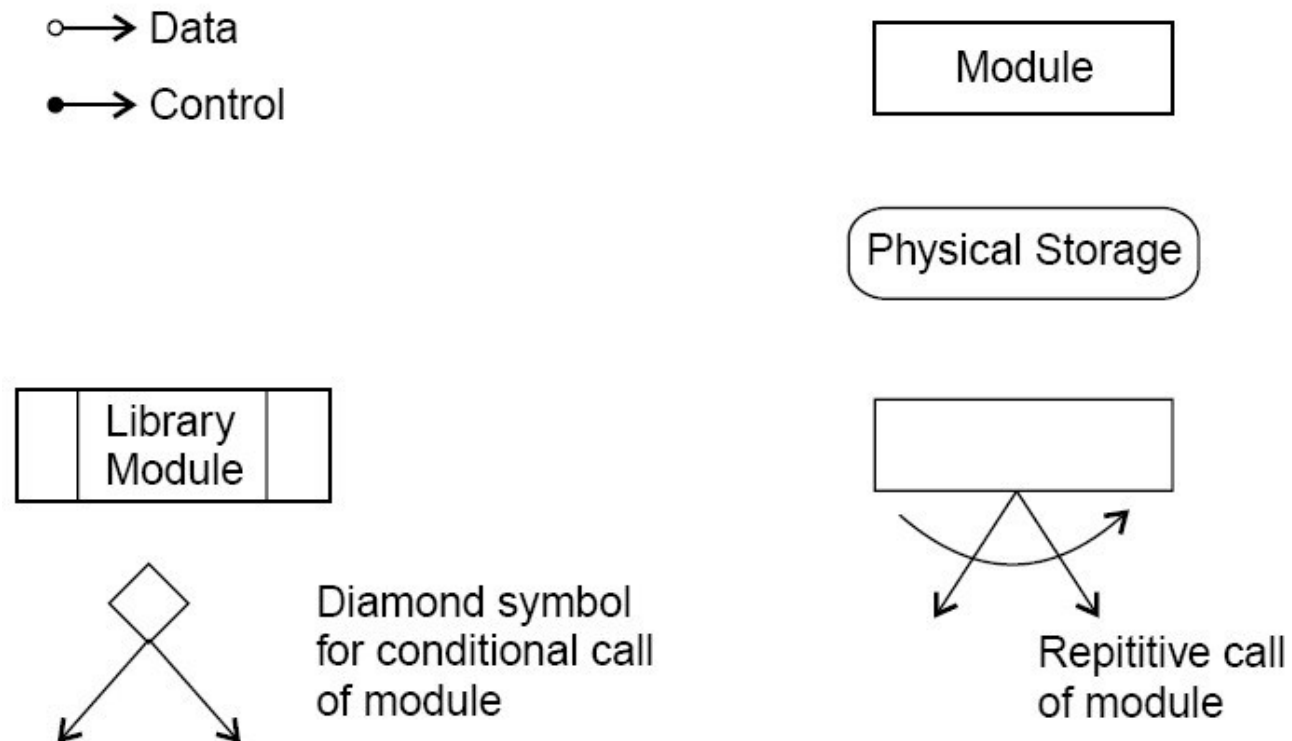


Fig. 17 : Structure chart notations

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

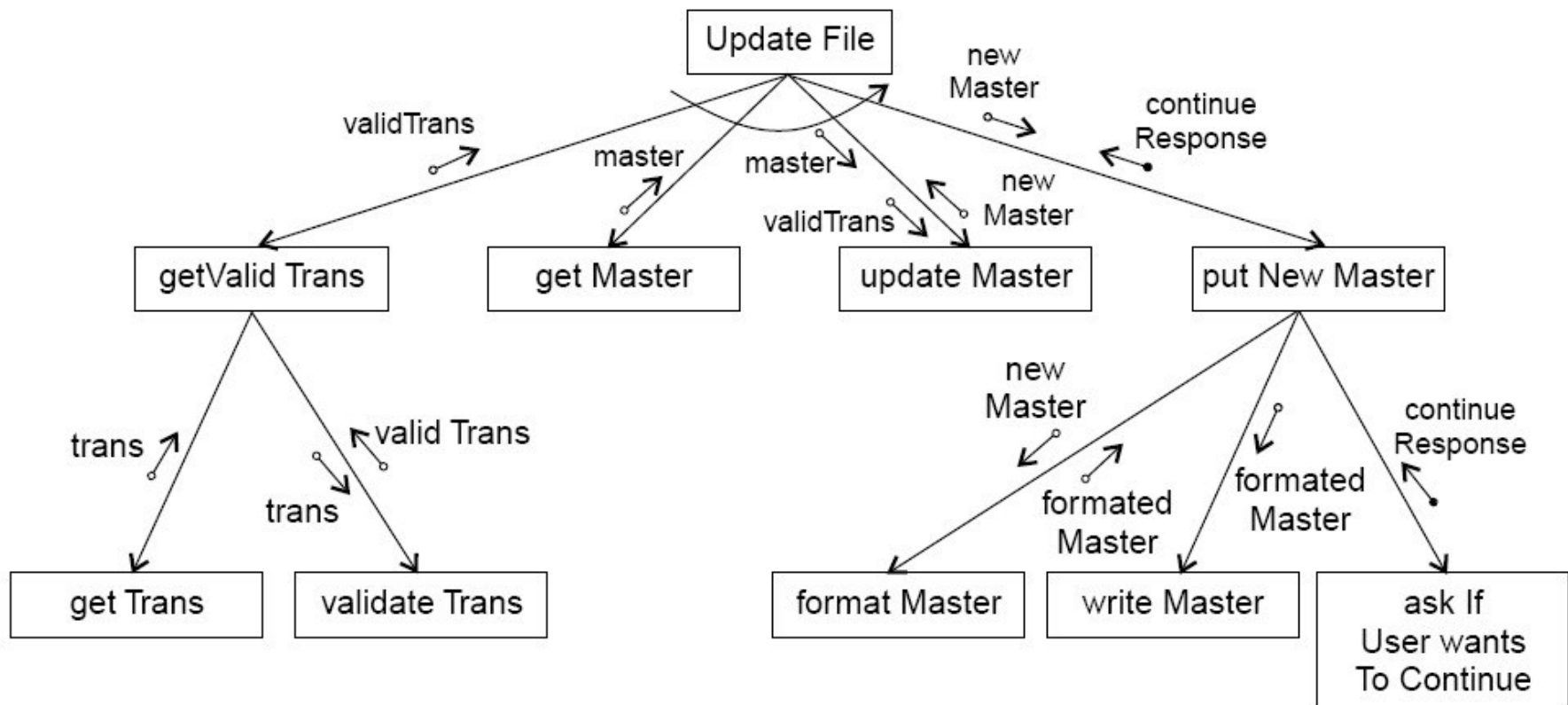


Fig. 18 : Update file

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

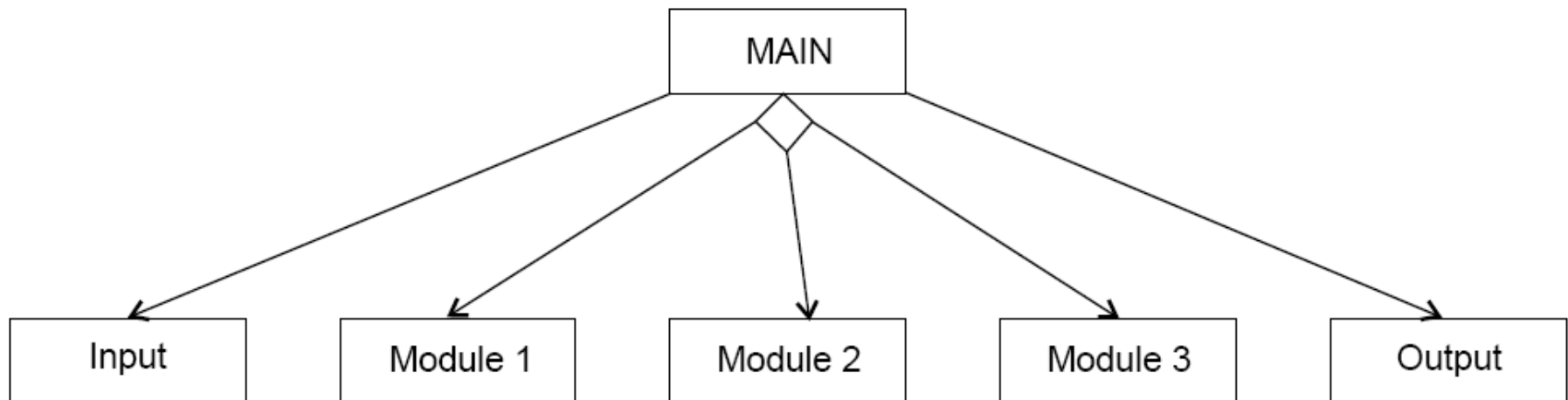


Fig. 19 : Transaction-centered structure

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.

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.

Functional Procedure Layers

- Function 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.)

IEEE Recommended practice for software design descriptions (IEEE STD 1016-1998)

➤ Scope

An SDD is a representation of a software system that is used as a medium for communicating software design information.

➤ References

- i. IEEE std 830-1998, IEEE recommended practice for software requirements specifications.
- ii. IEEE std 610.12-1990, IEEE glossary of software engineering terminology.

➤ **Definitions**

- 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 attributes.** 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.

➤ 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.

➤ Design Description Information Content

- Introduction
- Design entities
- Design entity attributes

The attributes and associated information items are defined in the following subsections:

a) Identification

b) Type

c) Purpose

d) Function

e) Subordinates

f) Dependencies

g) Interface

h) Resources

i) Processing

j) Data

➤ Design Description Organization

Each design description writer may have a different view of what are considered the essential aspects of a software design. The organization of SDD is given in table 1. This is one of the possible ways to organize and format the SDD.

A recommended organization of the SDD into separate design views to facilitate information access and assimilation is given in table 2.

-
1. Introduction
 - 1.1 Purpose
 - 1.2 Scope
 - 1.3 Definitions and acronyms
 2. References
 3. Decomposition description
 - 3.1 Module decomposition
 - 3.1.1 Module 1 description
 - 3.1.2 Module 2 description
 - 3.2 Concurrent Process decomposition
 - 3.2.1 Process 1 description
 - 3.2.2 Process 2 description
 - 3.3 Data decomposition
 - 3.3.1 Data entity 1 description
 - 3.3.2 Data entity 2 description

Cont...

-
- 4. Dependency description
 - 4.1 Intermodule dependencies
 - 4.2 Interprocess dependencies
 - 4.3 Data dependencies
 - 5. Interface description
 - 5.1 Module Interface
 - 5.1.1 Module 1 description
 - 5.1.2 Module 2 description
 - 5.2 Process interface
 - 5.2.1 Process 1 description
 - 5.2.2 Process 2 description
 - 6. Detailed design
 - 6.1 Module detailed design
 - 6.1.1 Module 1 detail
 - 6.1.2 Module 2 detail
 - 6.2 Data detailed design
 - 6.2.1 Data entry 1 detail
 - 6.2.2 Data entry 2 detail

Table 1:
Organization of
SDD

Design View	Scope	Entity attribute	Example representation
Decomposition 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.

Table 2: Design views

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 do 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.

➤ 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)

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.

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. Message 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.

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.

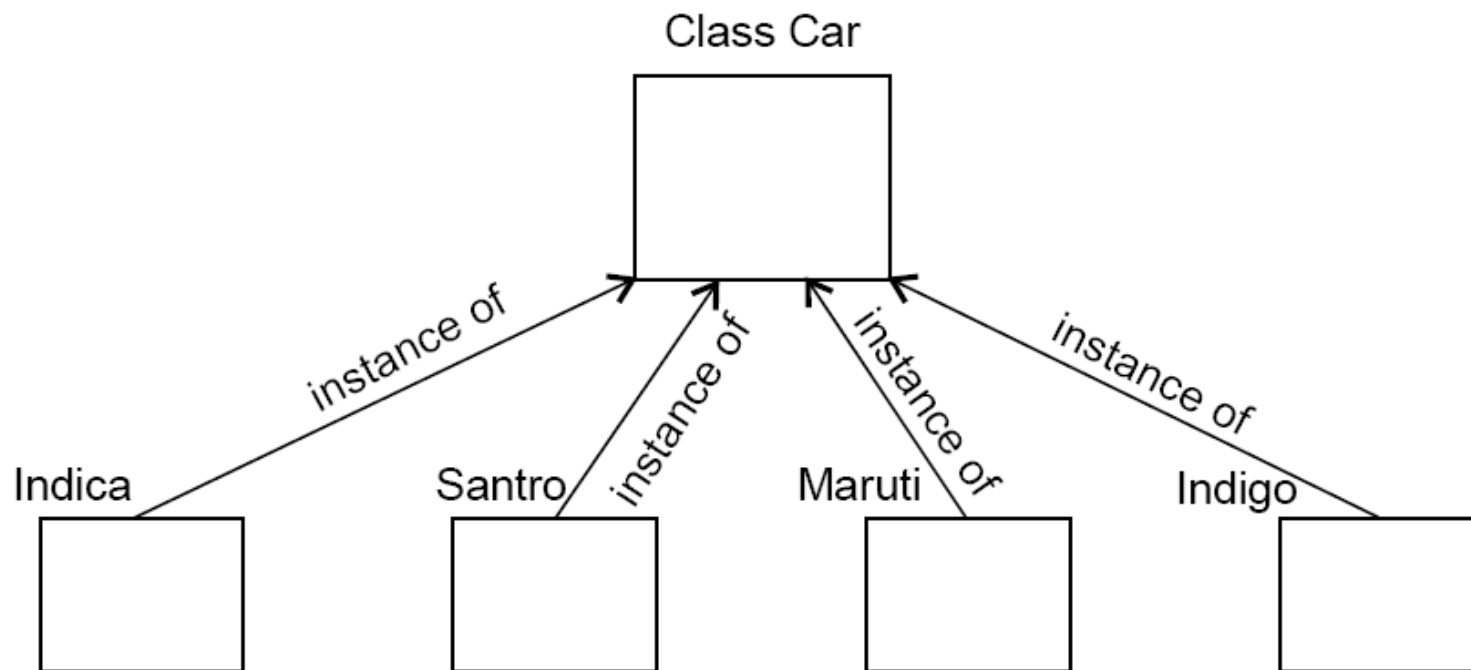


Fig.20: Indica, Santro, Maruti, Indigo are all instances of the class “car”

Class Square

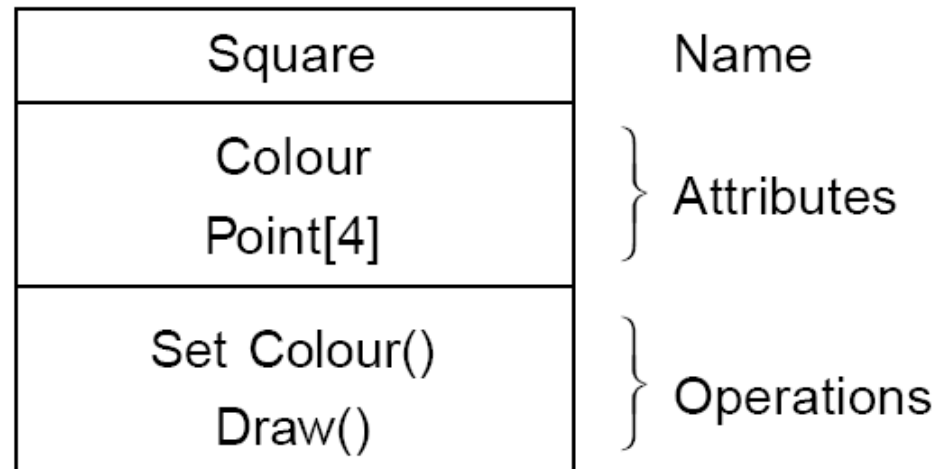


Fig. 21: The square class

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.

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

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.

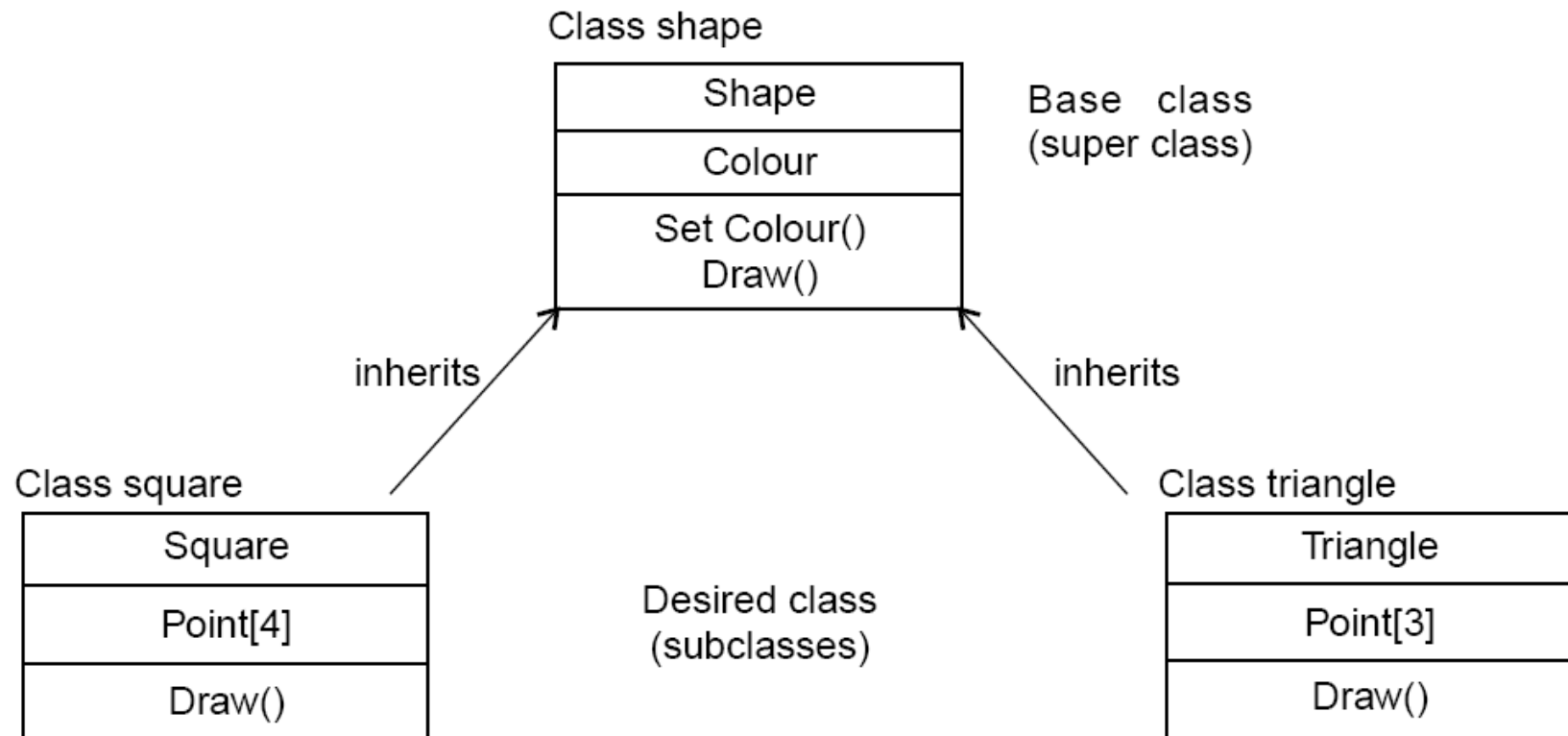


Fig. 23: Abstracting common features in a new class

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).

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.

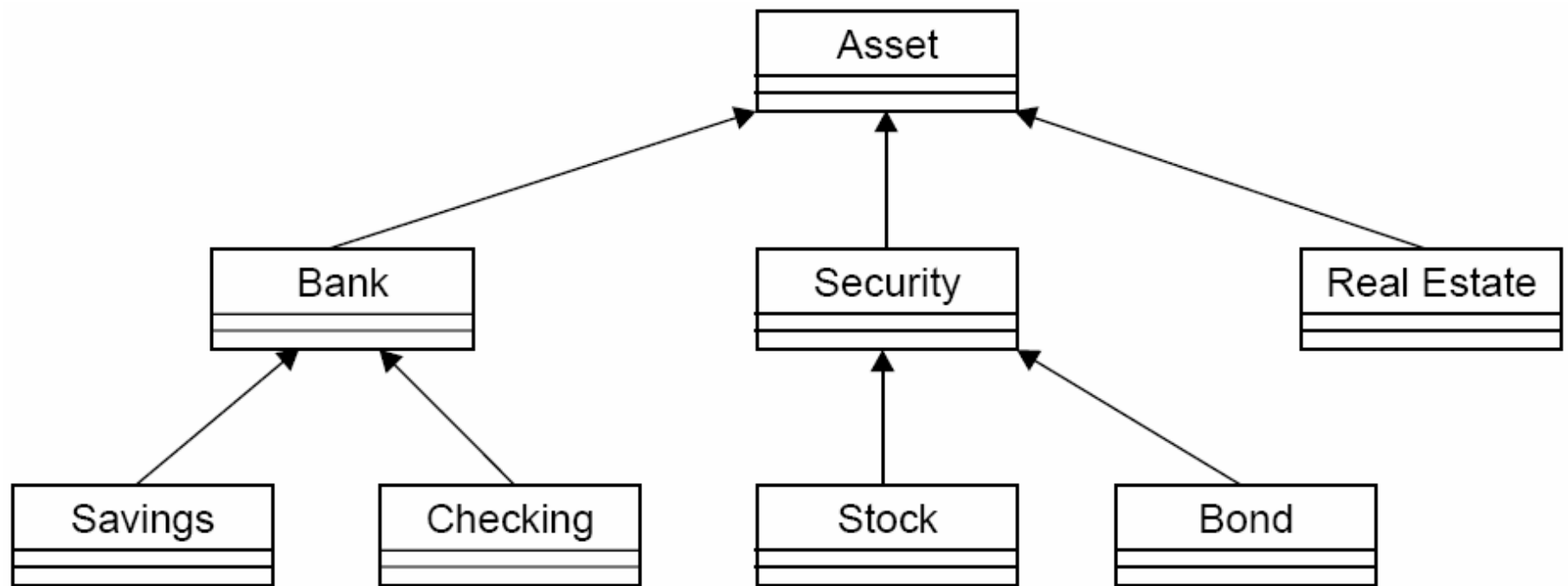


Fig. 24: Hierarchy

➤ 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. 25

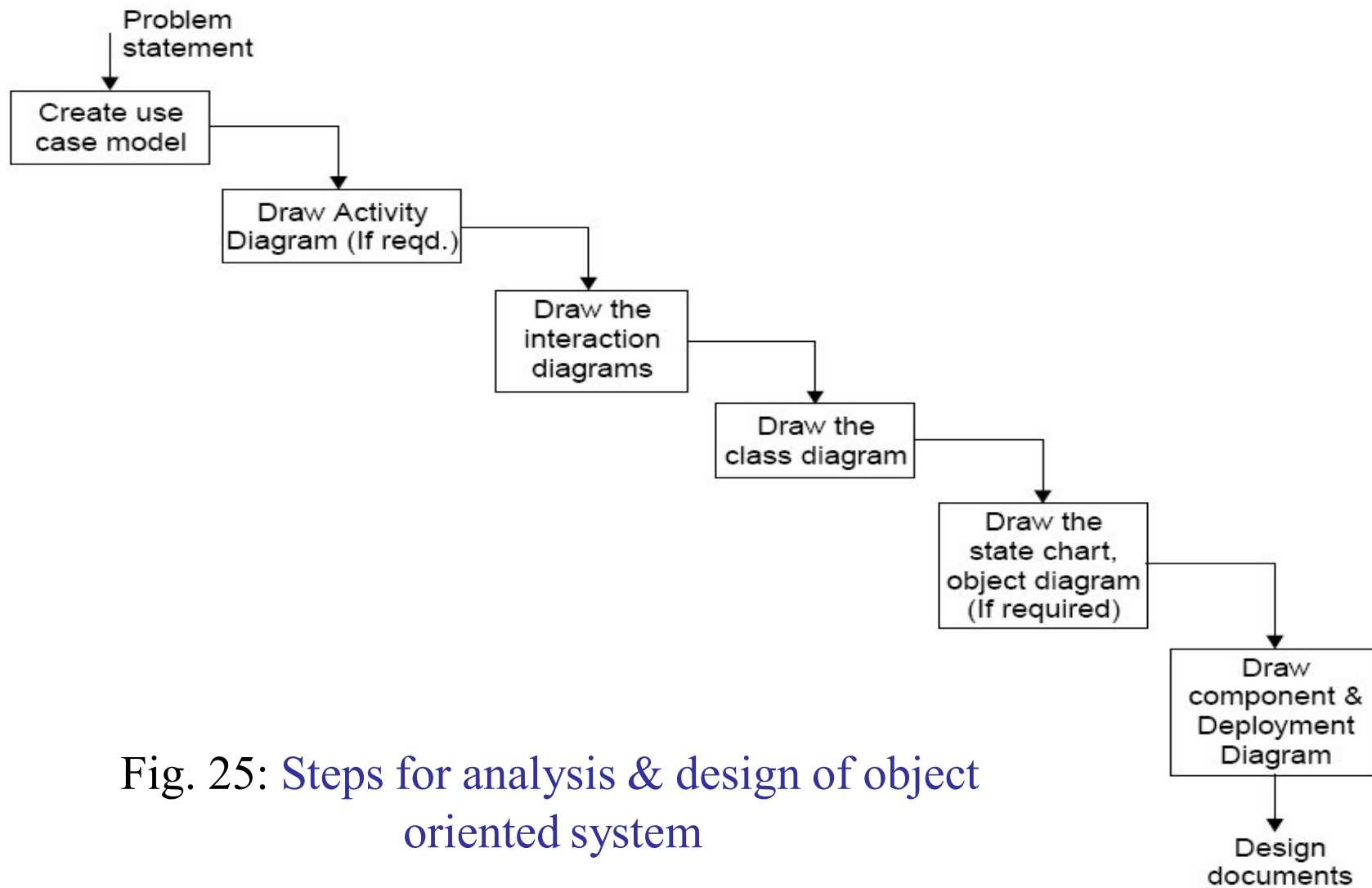


Fig. 25: Steps for analysis & design of object oriented system

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. Fig. 26 shows the activity diagram processing an order to deliver some goods.

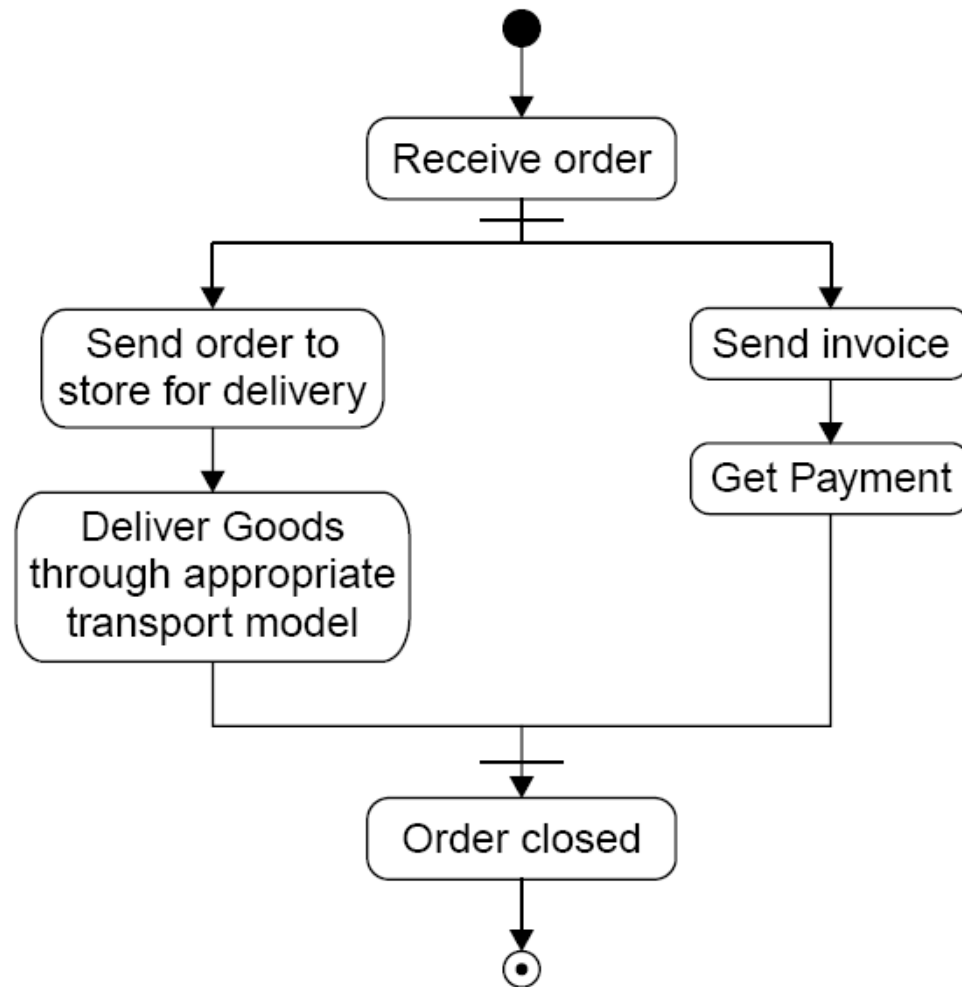


Fig. 26: Activity diagram

iii. Draw the interaction diagram

An interaction diagram shows an interaction, consisting of a set of objects and their relationship, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system.

Steps to draw interaction diagrams are as under:

- a) Firstly, we should identify the objects with respect to every use case.
- b) We draw the sequence diagrams for every use case.
- d) We draw the collaboration diagrams for every use case.

The object types used in this analysis model are entity objects, interface objects and control objects as given in fig. 27.

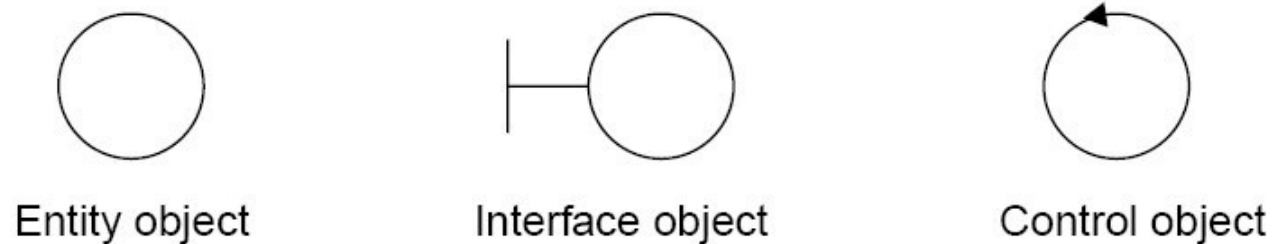


Fig. 27: Object types

iv. Draw the class diagram

The class diagram shows the relationship amongst classes. There are four types of relationships in class diagrams.

- a) **Association** are semantic connection between classes. When an association connects two classes, each class can send messages to the other in a sequence or a collaboration diagram. Associations can be bi-directional or unidirectional.
-

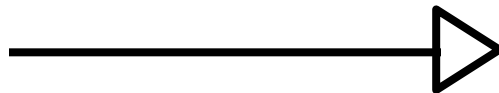
-
- b) **Dependencies** connect two classes. Dependencies are always unidirectional and show that one class depends on the definitions in another class.



- c) **Aggregations** are a stronger form of association. An aggregation is a relationship between a whole and its parts.



- d) **Generalizations** are used to show an inheritance relationship between two classes.



v. Design of state chart diagrams

A state chart diagram is used to show the state space of a given class, the event that cause a transition from one state to another, and the action that result from a state change. A state transition diagram for a “book” in the library system is given in fig. 28.

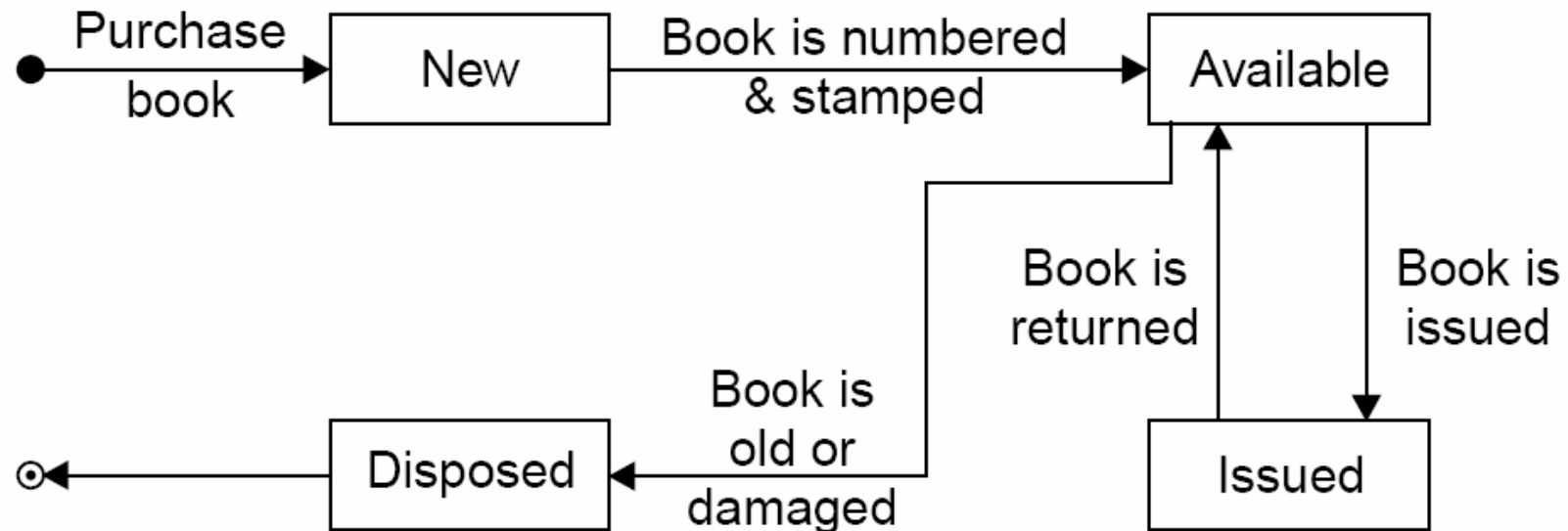


Fig. 28: Transition chart for “book” in a library system.

vi. Draw component and development diagram

Component diagrams address the static implementation view of a system they are related to class diagrams in that a component typically maps to one or more classes, interfaces or collaboration.

Deployment Diagram Captures relationship between physical components and the hardware.

A software has to be developed for automating the manual library of a University. The system should be stand alone in nature. It should be designed to provide functionality's as explained below:

Issue of Books:

- ❖ A student of any course should be able to get books issued.
- ❖ Books from General Section are issued to all but Book bank books are issued only for their respective courses.
- ❖ A limitation is imposed on the number of books a student can issue.
- ❖ A maximum of 4 books from Book bank and 3 books from General section is issued for 15 days only. The software takes the current system date as the date of issue and calculates date of return.

-
- ❖ A bar code detector is used to save the student as well as book information.
 - ❖ The due date for return of the book is stamped on the book.

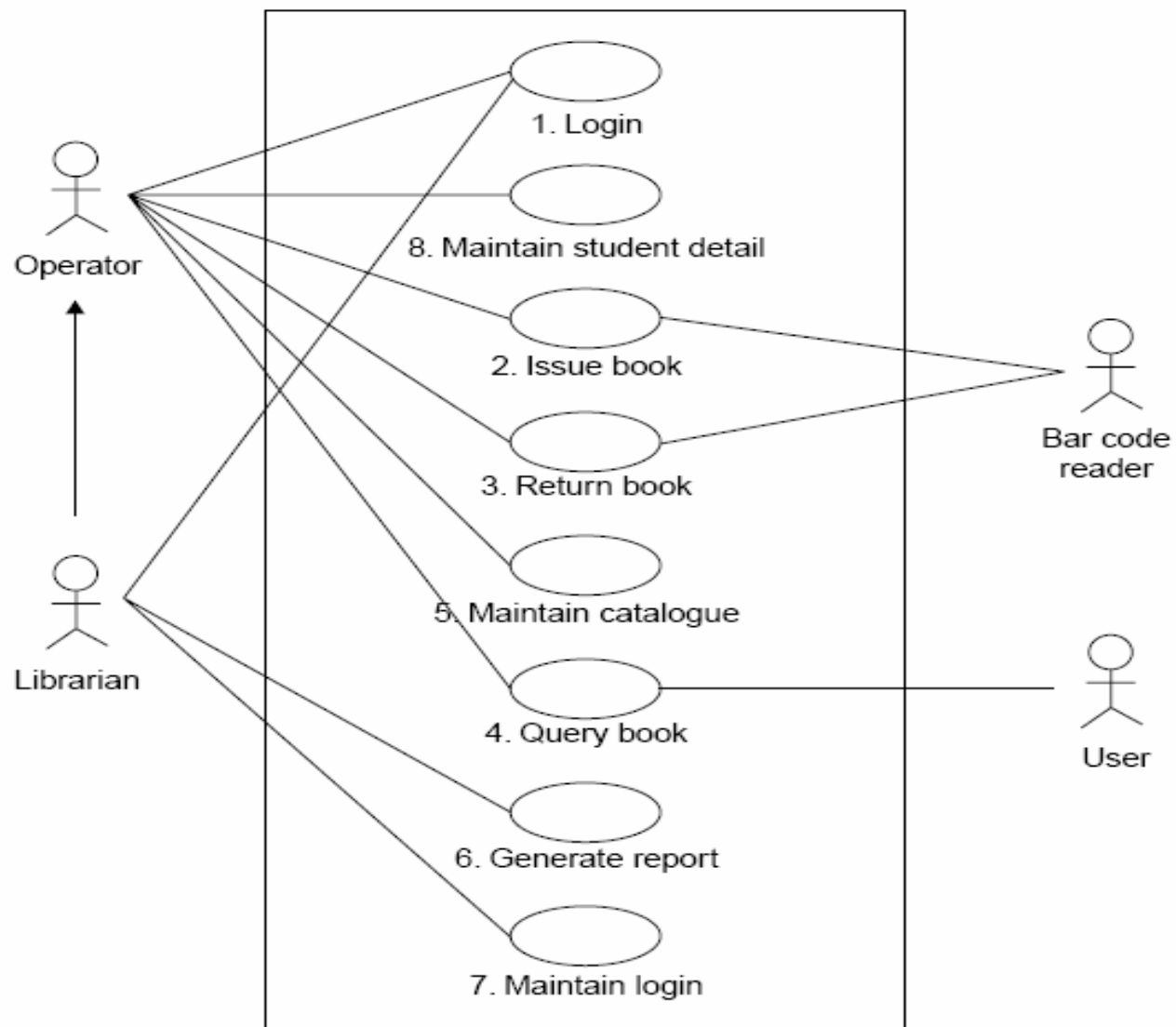
Return of Books:

- ❖ Any person can return the issued books.
- ❖ The student information is displayed using the bar code detector.
- ❖ The system displays the student details on whose name the books were issued as well as the date of issue and return of the book.
- ❖ The system operator verifies the duration for the issue.
- ❖ The information is saved and the corresponding updating take place in the database.

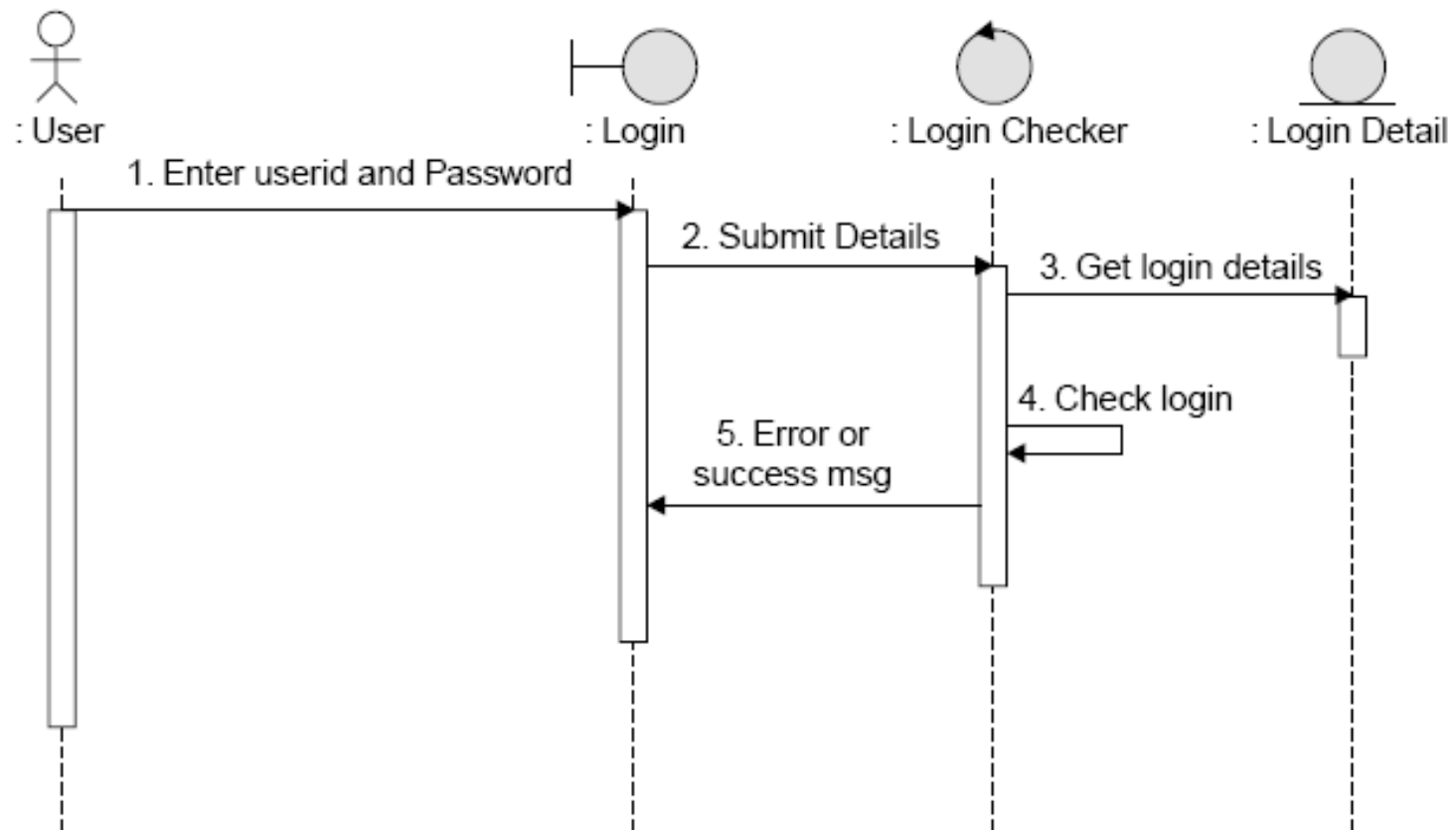
Query Processing:

- ❖ The system should be able to provide information like:
- ❖ Availability of a particular book.
- ❖ Availability of book of any particular author.
- ❖ Number of copies available of the desired book.

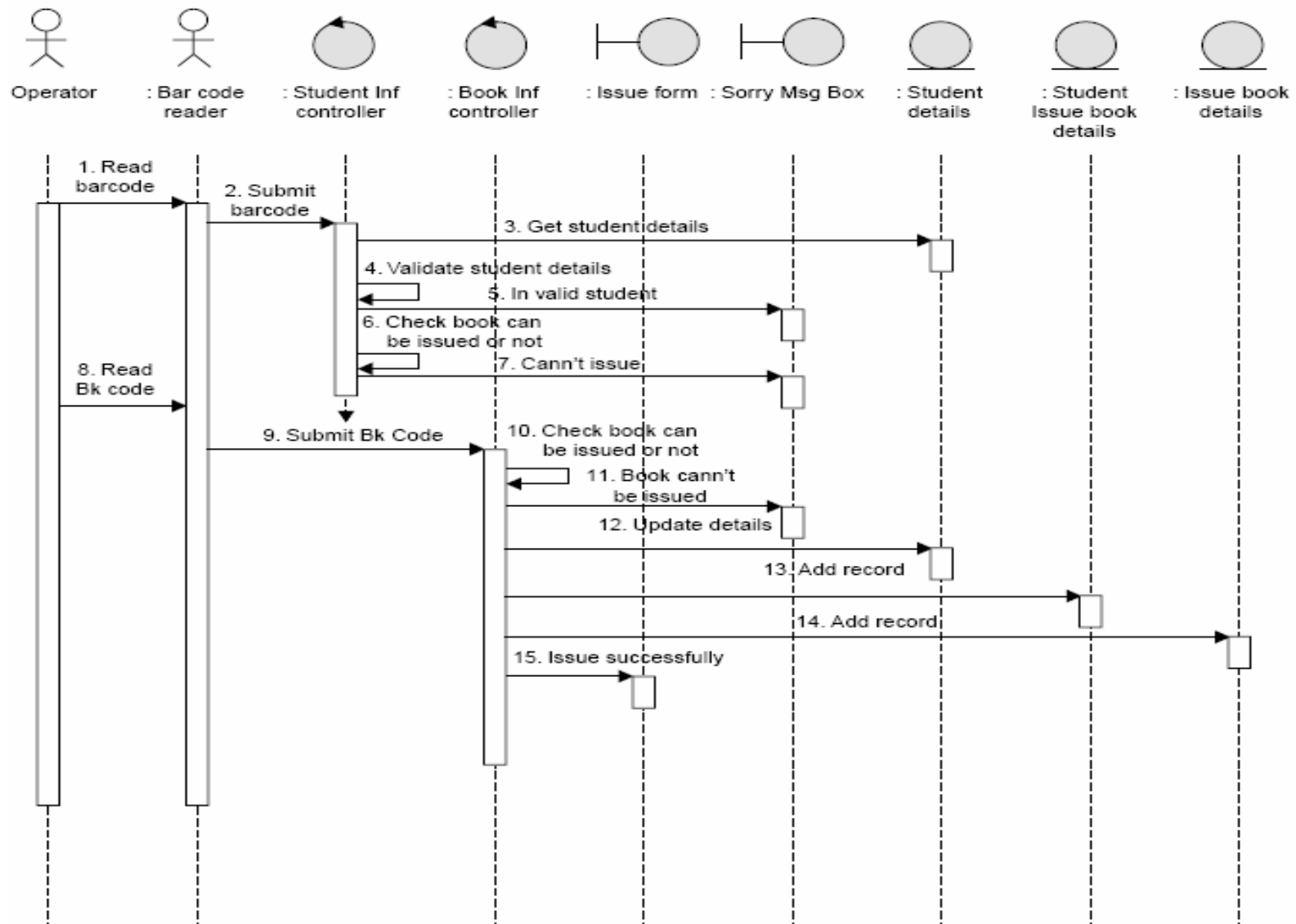
The system should also be able to generate reports regarding the details of the books available in the library at any given time. The corresponding printouts for each entry (issue/return) made in the system should be generated. Security provisions like the 'login authenticity should be provided. Each user should have a user id and a password. Record of the users of the system should be kept in the log file. Provision should be made for full backup of the system.



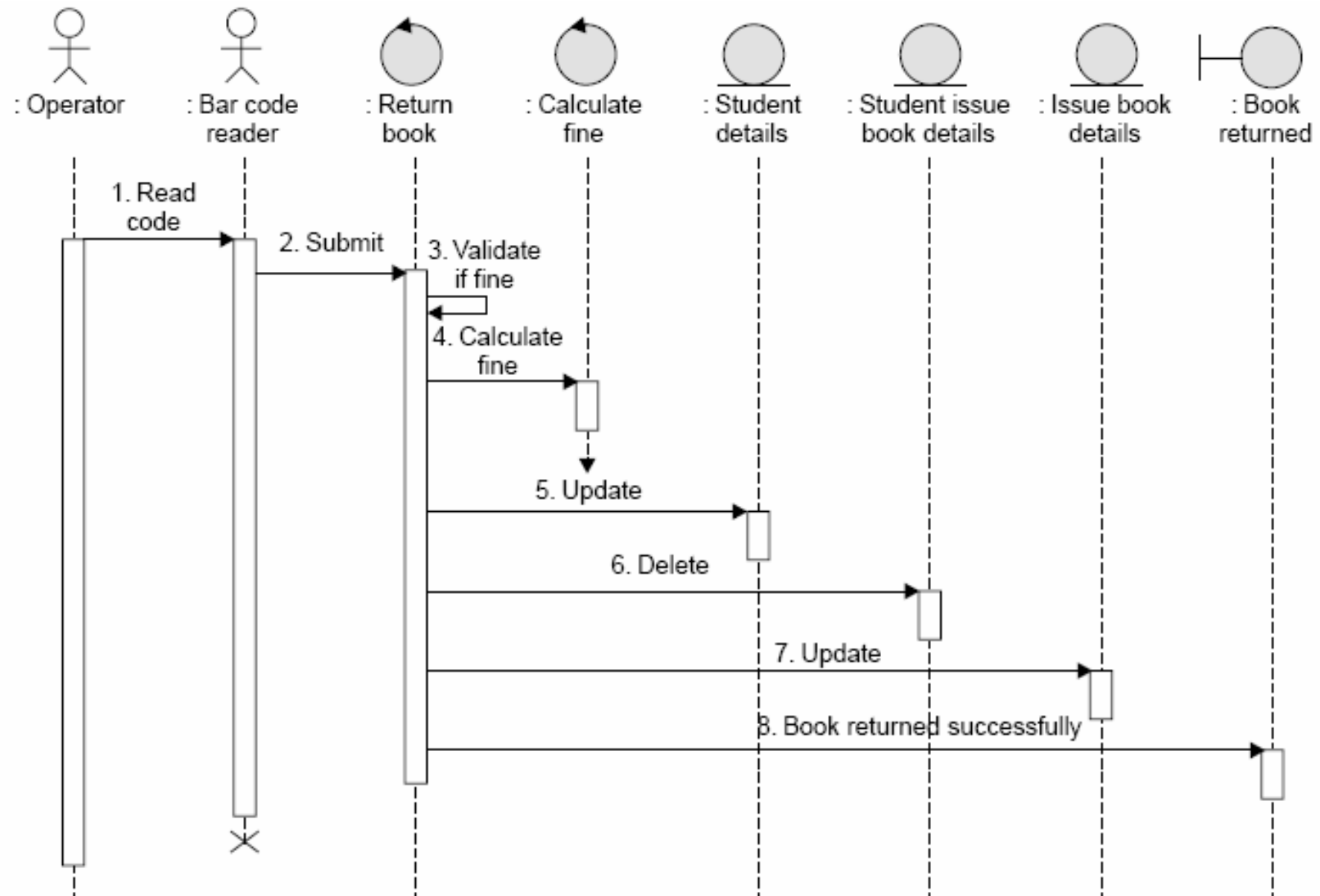
Use case diagram for library management system



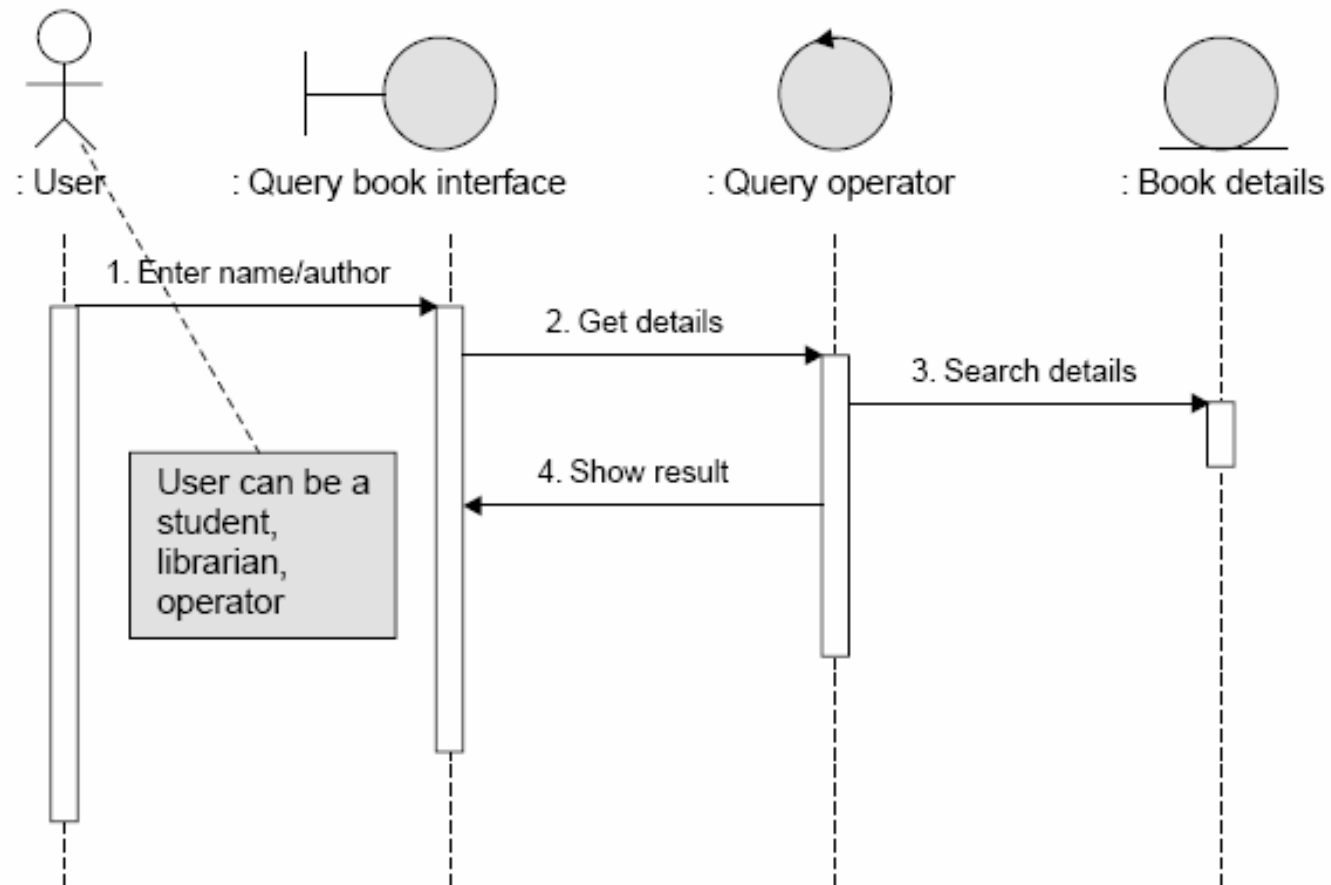
Sequence diagram—Login



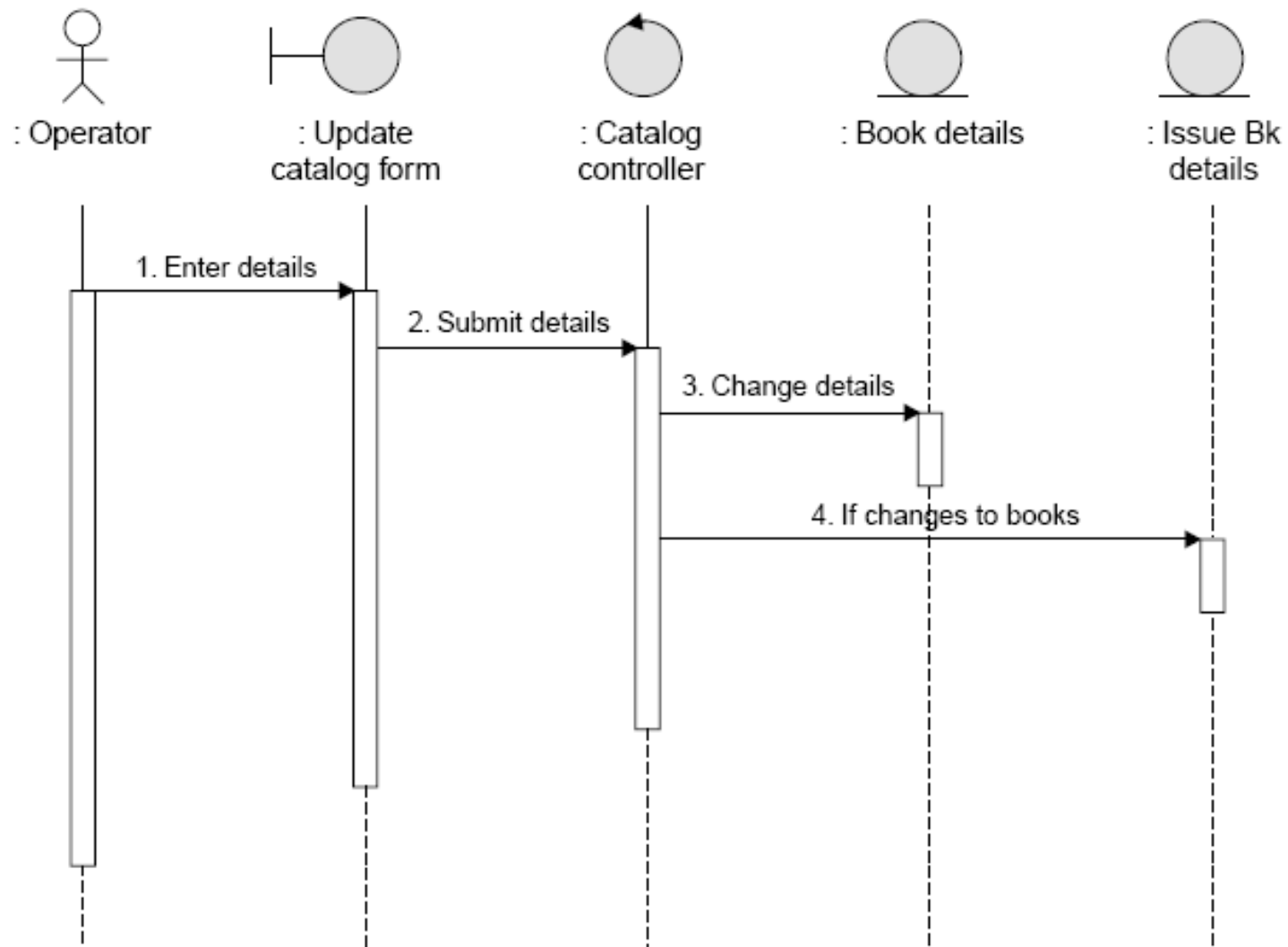
Sequence diagram—issue book



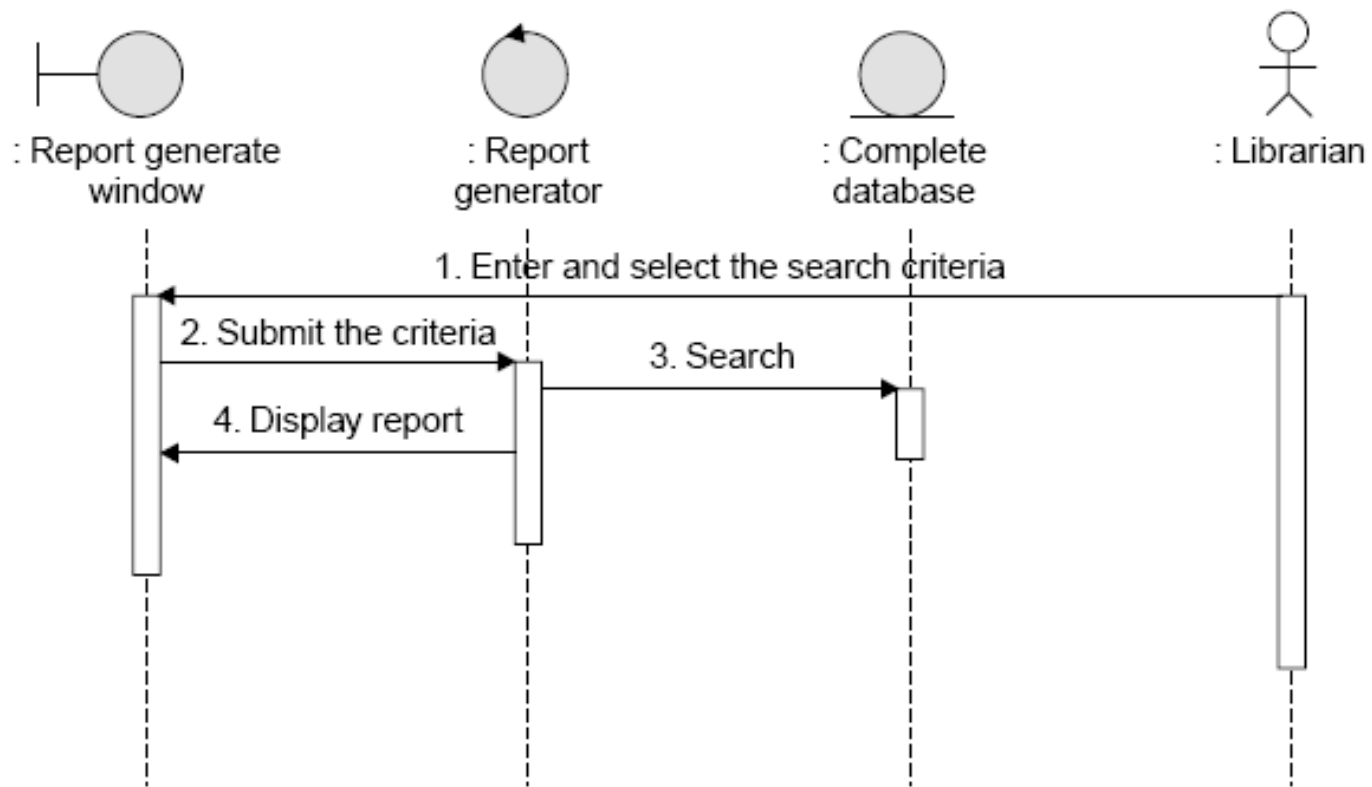
Sequence diagram—return book



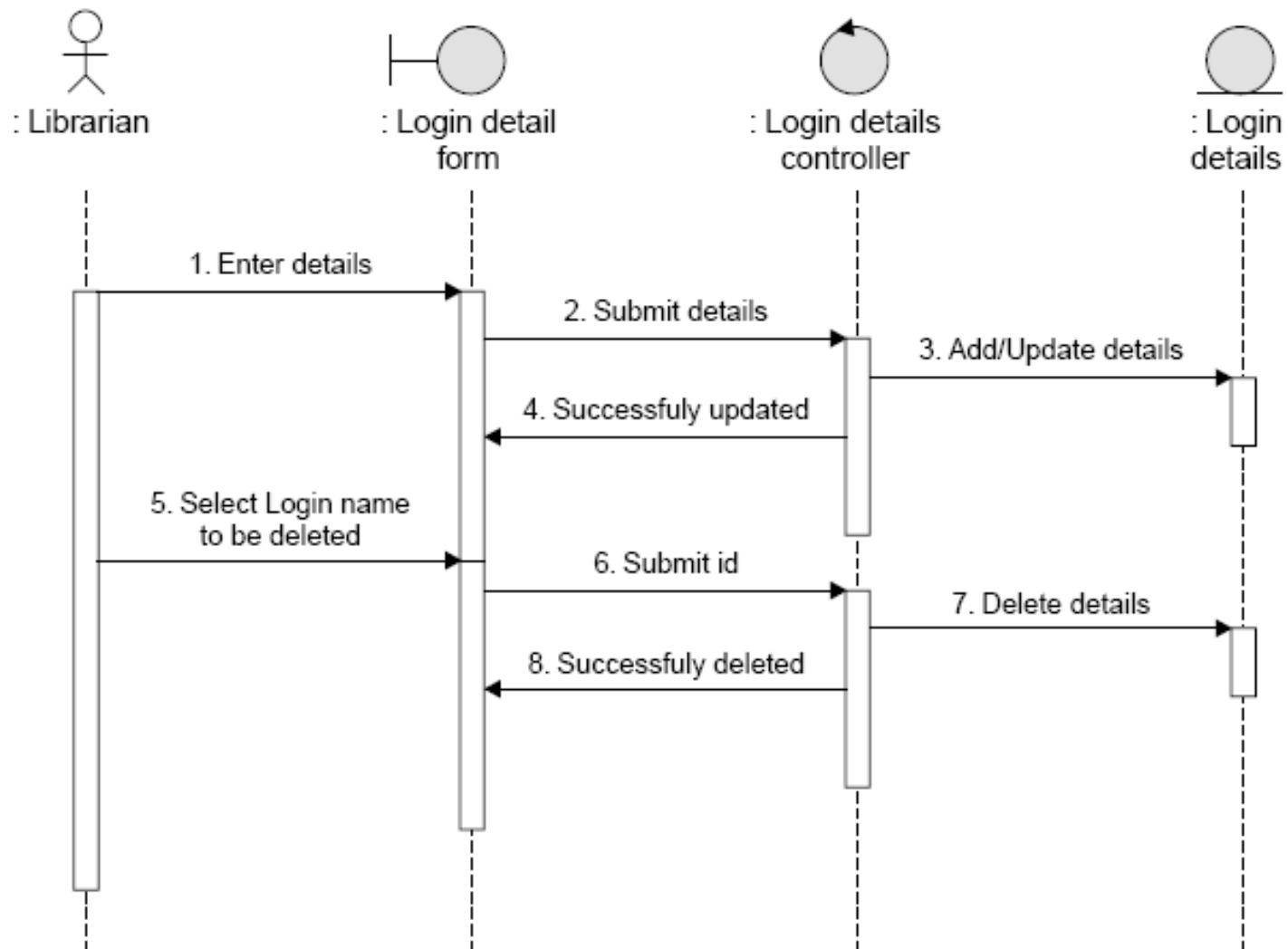
Sequence diagram—query book



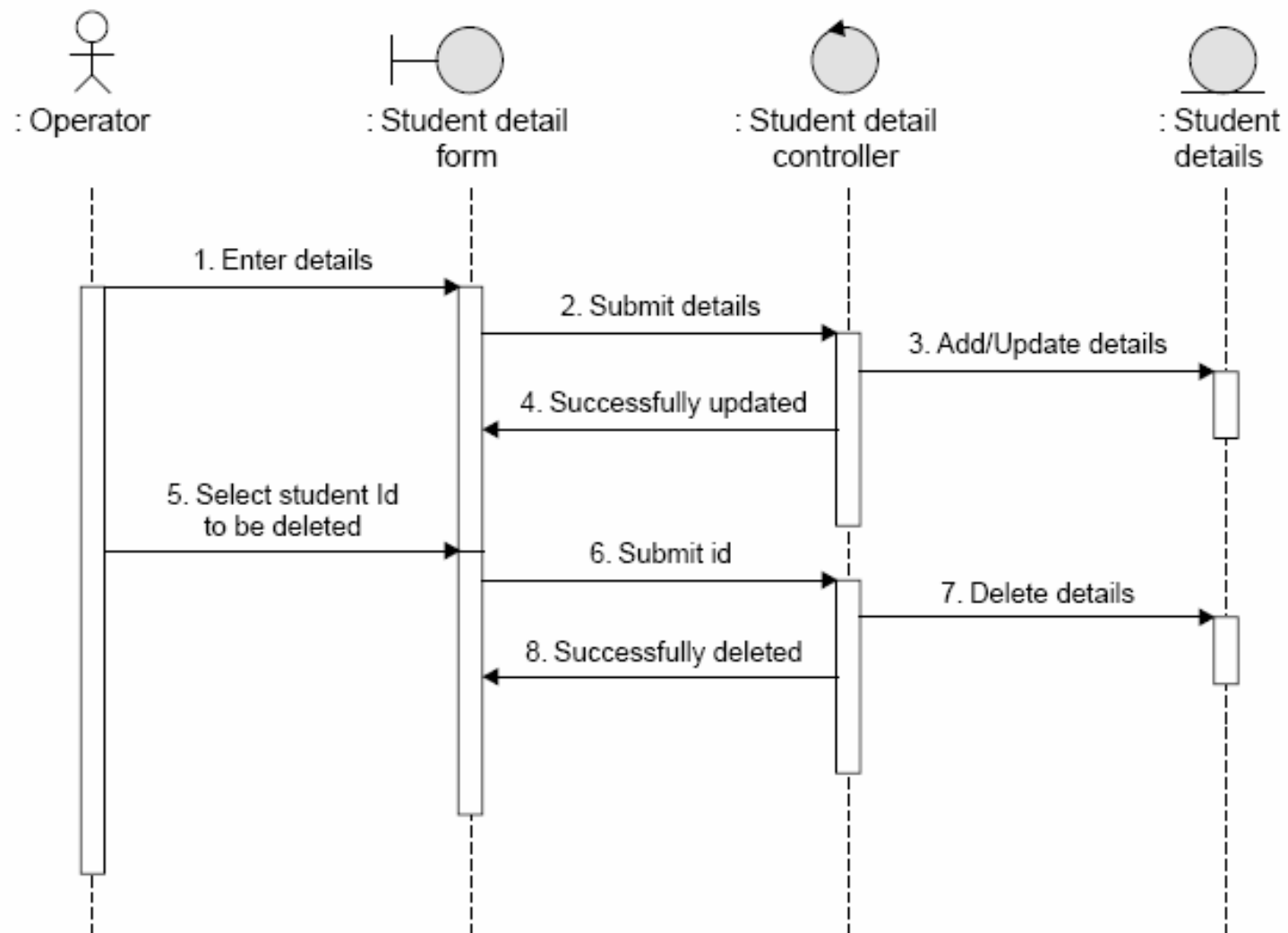
Sequence diagram—maintain catalog



Sequence diagram—generate reports



Sequence diagram—maintain login



Sequence diagram—maintain student details

Class diagram of entity classes

