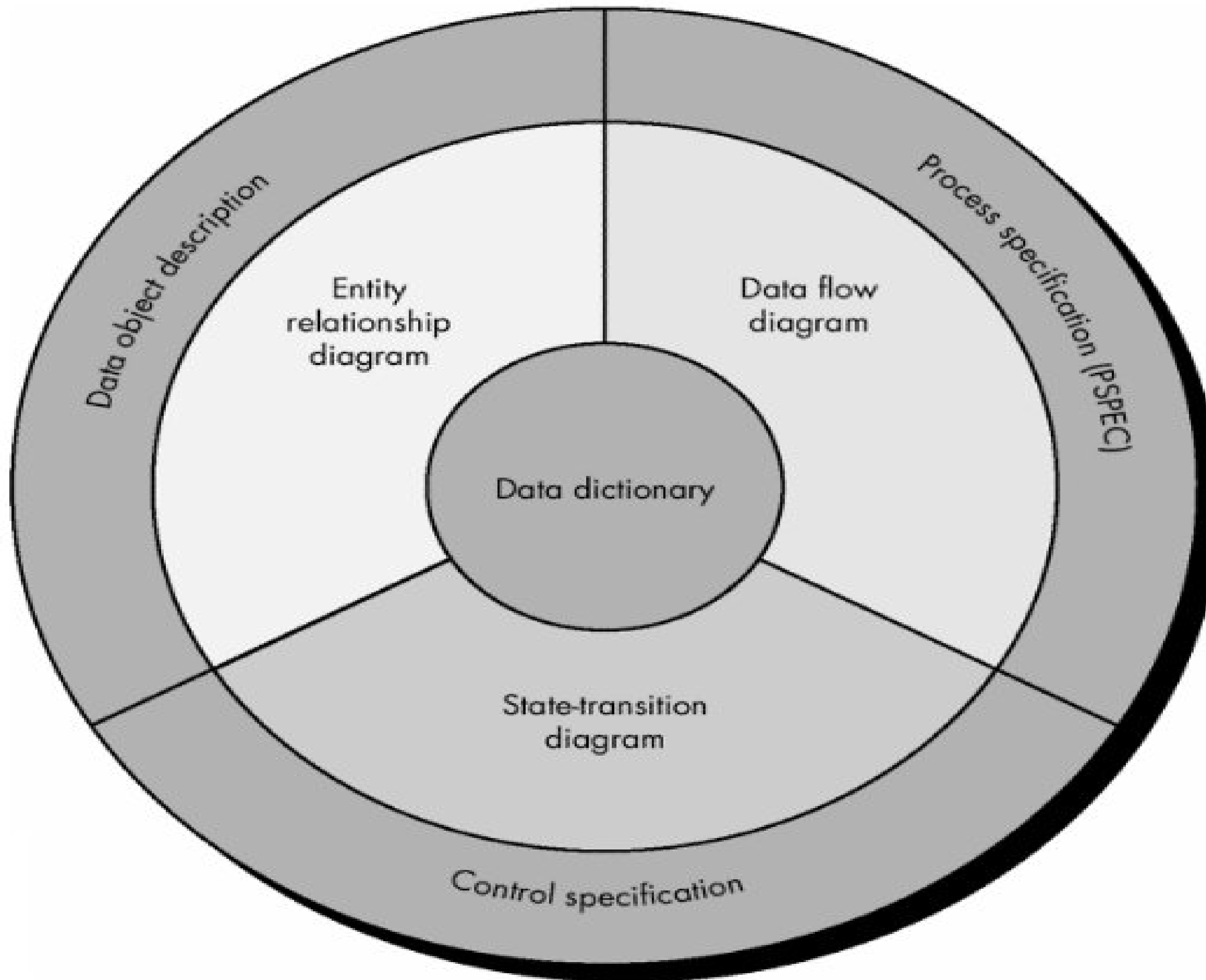
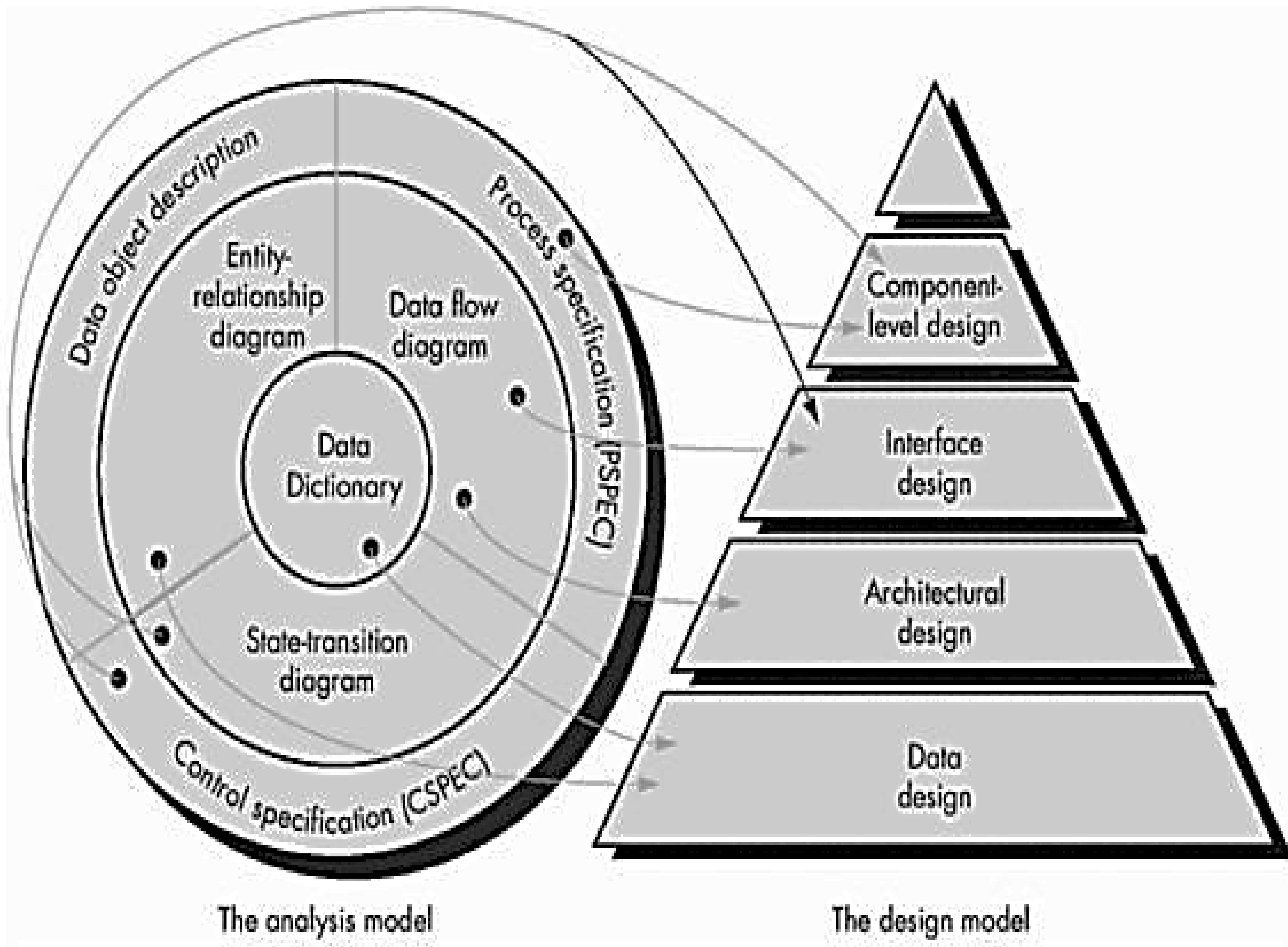
UNIT III

Software Design


Data object description
Entity relationship diagram
Data flow diagram
Process specification (PSPEC)
Data dictionary
State-transition diagram
Control specification


Data object description
Entity-relationship diagram
Data flow diagram
Process specification (PSPEC)
Data Dictionary
State-transition diagram
Control specification (CSPEC)
Component-level design
Interface design
Architectural design
Data design
The analysis model
The design model

Design Specification Models

Data design - created by transforming the analysis information model (data dictionary and ERD) into data structures required to implement the software

Architectural design - defines the relationships among the major structural elements of the software, it is derived from the system specification, the analysis model, and the subsystem interactions defined in the analysis model (DFD)

Interface design - describes how the software elements communicate with each other, with other systems, and with human users; the data flow and control flow diagrams provide much the necessary information

Component-level design - created by transforming the structural elements defined by the software architecture into procedural descriptions of software components using information obtained from the PSPEC, CSPEC, and STD

Design and Quality

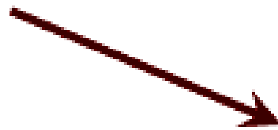
- the design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- the design must be a readable, understandable guide for those who generate code and for those who test and subsequently support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

Software Design

- More creative than analysis
- Problem solving activity

WHAT IS DESIGN

'HOW'



Software design document (SDD)

Design Definition

Software Design is the practice of taking a specification of externally observable behaviour and adding details needed for actual computer system implementation, including human interaction, task management, and data management details. [**Coad and Yourdon]**

Software Design -- An iterative process transforming requirements into a “blueprint” for constructing the software.

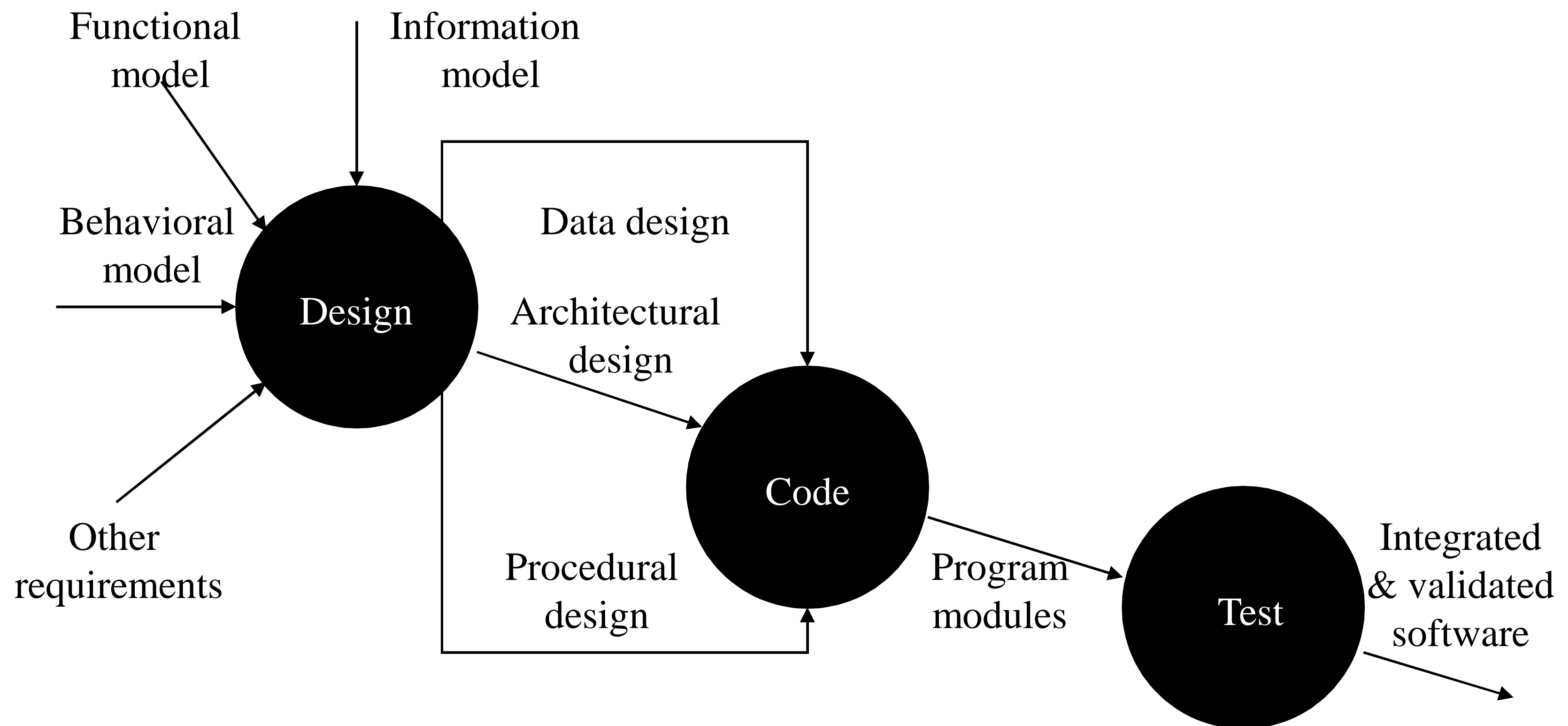
Input includes an understanding of the following:

- Requirements
- Environmental constraints
- Design criteria

The output of the design effort is composed of the following:

- Architecture design which shows how pieces are interrelated
- Specifications for any new pieces
- Definitions for any new data

Software Design Model



Software Design

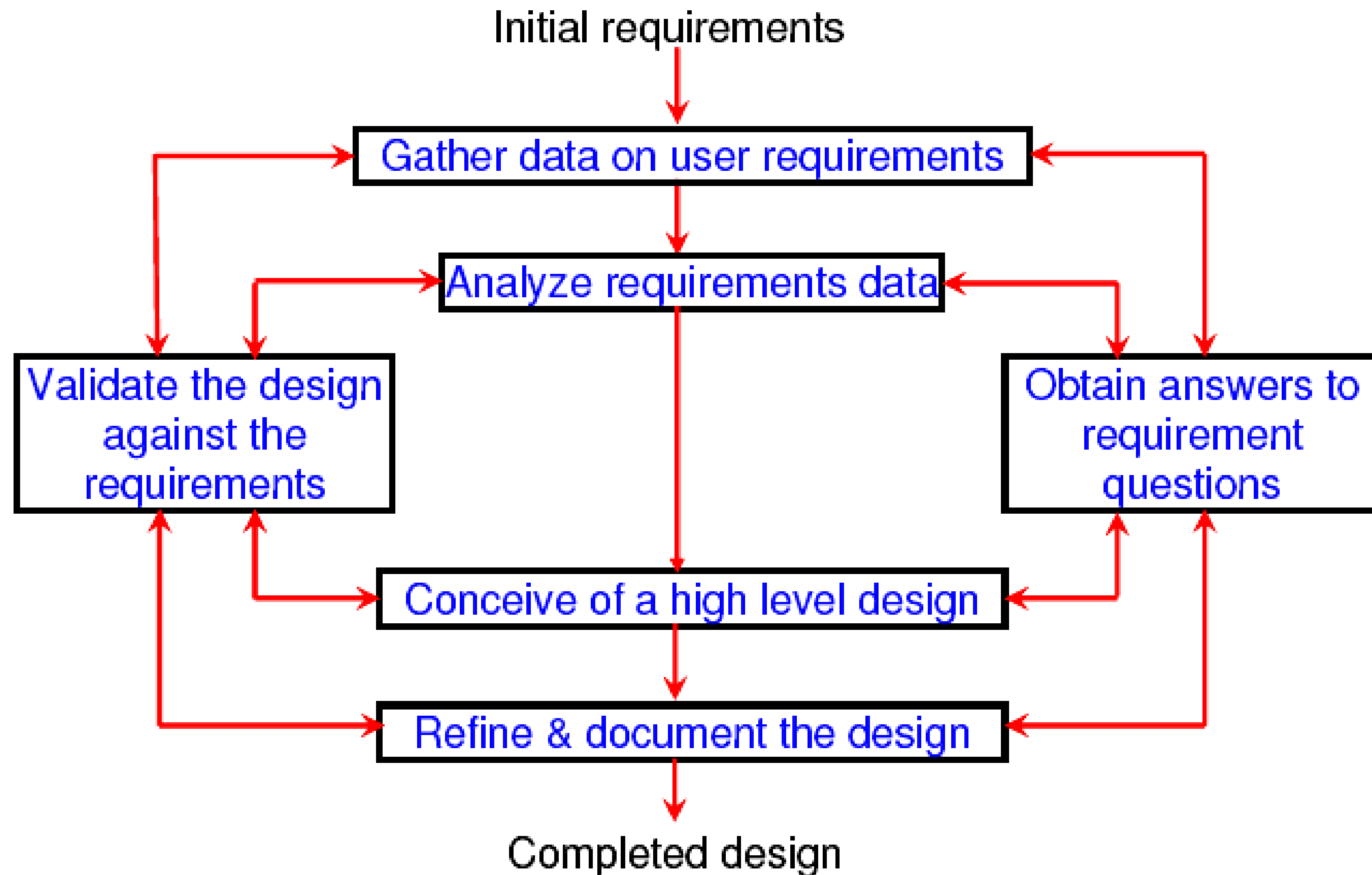


Fig. 1 : Design framework

Software Design

Conceptual Design and Technical Design

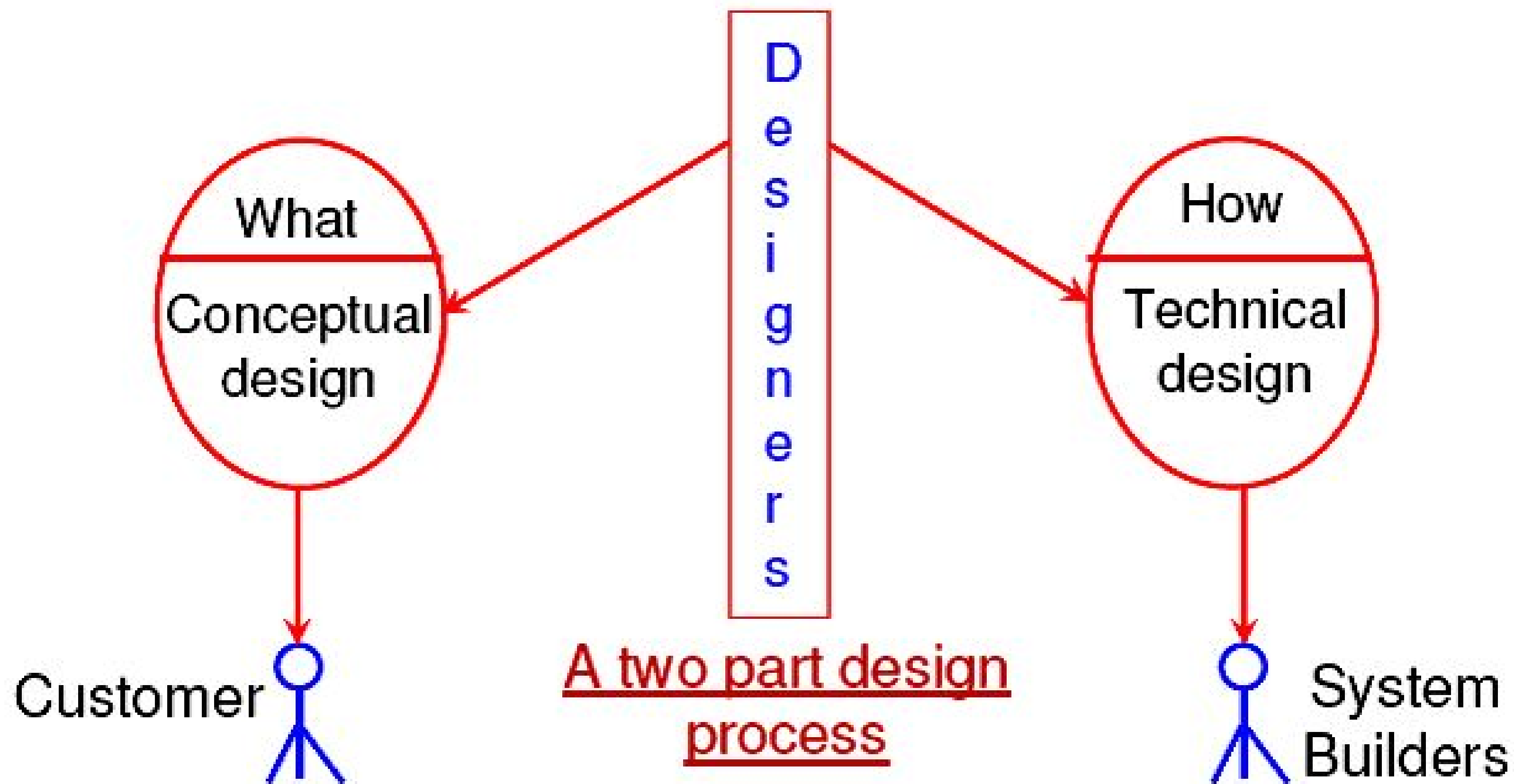


Fig. 2 : A two part design process

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.

Good Design Characteristics

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

Quality Guidelines

- A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion
  - For smaller systems, design can sometimes be developed linearly.
- A design should be modular; that is, the software should be logically partitioned into elements or subsystems
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

Design Principles

- The design process should not suffer from 'tunnel vision.'
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should "minimize the intellectual distance"

  between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

DESIGN STRATEGIES

- *Compositional.* Here entities and objects are classified, grouped, and interrelated by links.

(e.g) Jackson's structured programming and object-oriented design.

- *Decompositional.* It is a top-down approach where stepwise refinement is done. (e.g) Structured design approach
- *Evolutionary.* It is an incremental strategy
- *Template-based.* This strategy makes use of design reuse by instantiating design templates.(e.g)Software architectures, styles, and design patterns.

Design **METHODOLOGIES**

- 1. Top-Down Design
- 2. Data-Structure-Oriented Design
    - Jackson Design Methodology
    - Warnier-Orr Design Methodology
- 3. Miller's Database-Oriented Design
- 4. Constantine and Yourdon's Dataflow-Oriented Structured Design
- 5. Object-Oriented Design
- 6. Design of Architecture

1. Abstraction: Focus on solving a problem by considering the relevant details and ignoring the irrelevant

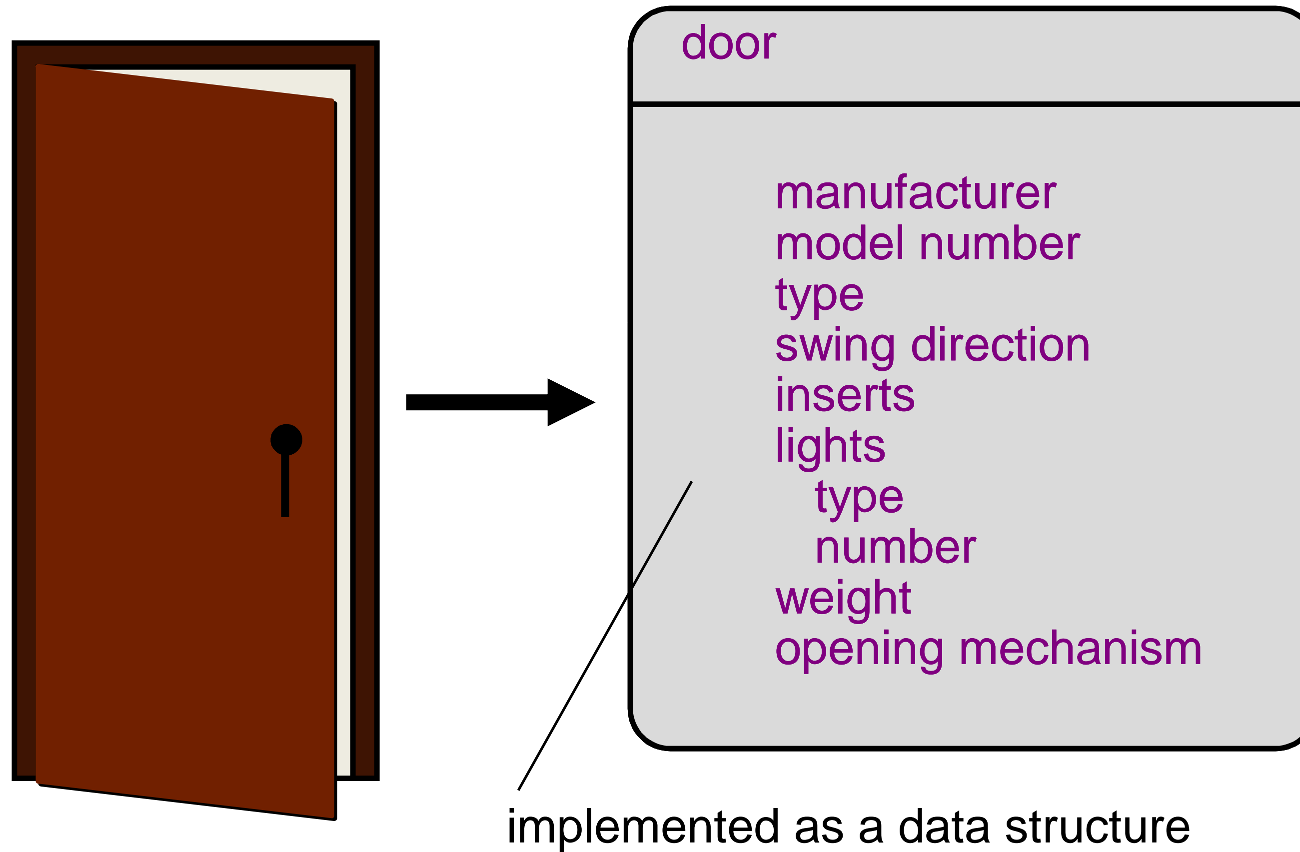
2. Encapsulation: Wrapping the internal details, thereby making these details inaccessible. Encapsulation separates interface and implementation, specifying only the public interface to the clients, hiding the details of implementation.

3. Decomposition and Modularization: Dividing the problem into smaller, independent, interactive subtasks for placing different functionalities in different components

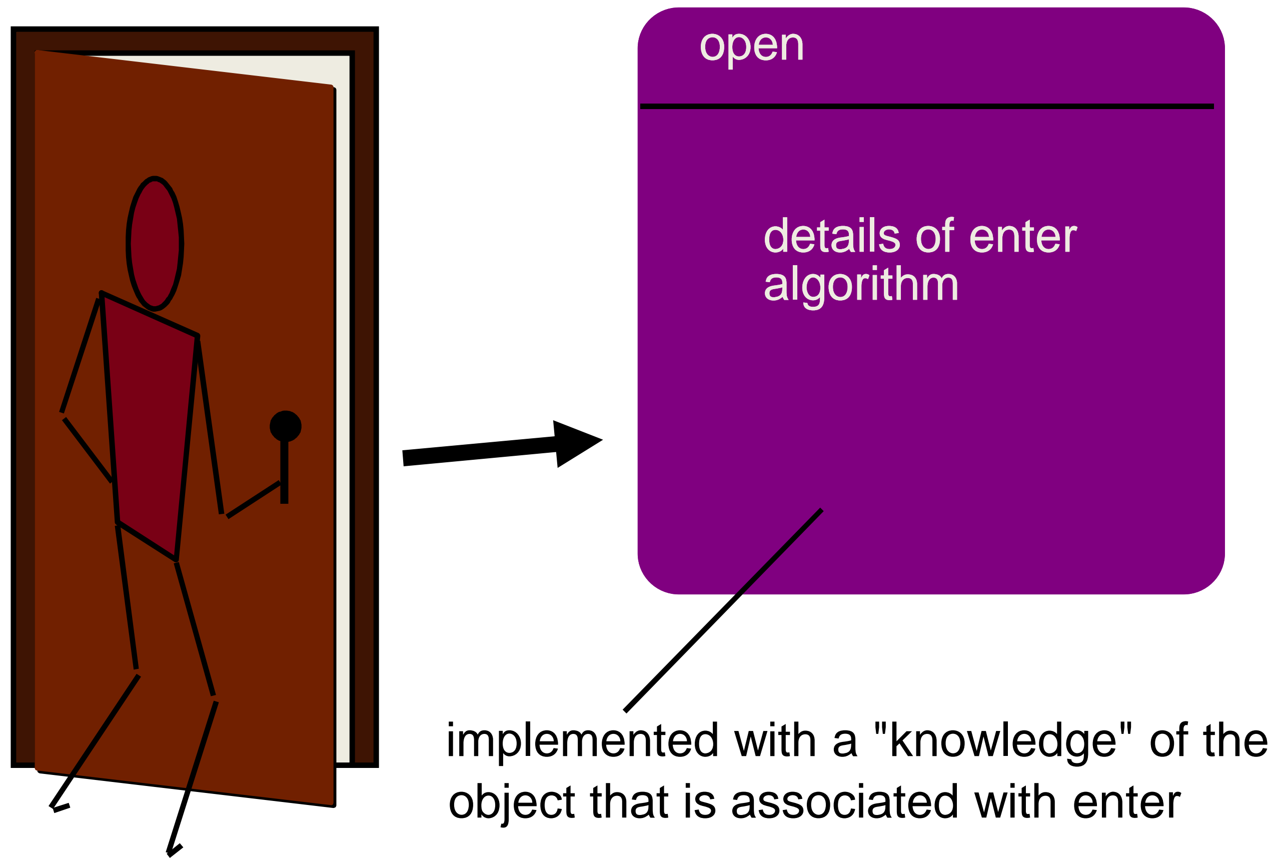
4. Coupling & Cohesion: Coupling is the degree to which modules are dependent on each other. Cohesion is the degree to which a module has a single, well defined task or responsibility. A good design is one with loose coupling and strong cohesion.

5.Sufficiency, Completeness and Primitiveness: Design should ensure the completeness and sufficiency with respect to the given specifications in a very simple way as possible

Data Abstraction



Procedural Abstraction



Architecture

“The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.” [SHA95a]

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Patterns

Design Pattern Template

Pattern name**—describes the essence of the pattern in a short but expressive name**

Intent**—describes the pattern and what it does**

Also-known-as**—lists any synonyms for the pattern**

Motivation**—provides an example of the problem**

Applicability**—notes specific design situations in which the pattern is applicable**

Structure**—describes the classes that are required to implement the pattern**

Participants**—describes the responsibilities of the classes that are required to implement the pattern**

Collaborations**—describes how the participants collaborate to carry out their responsibilities**

Consequences**—describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented**

Related patterns**—cross-references related design patterns**

Separation of Concerns

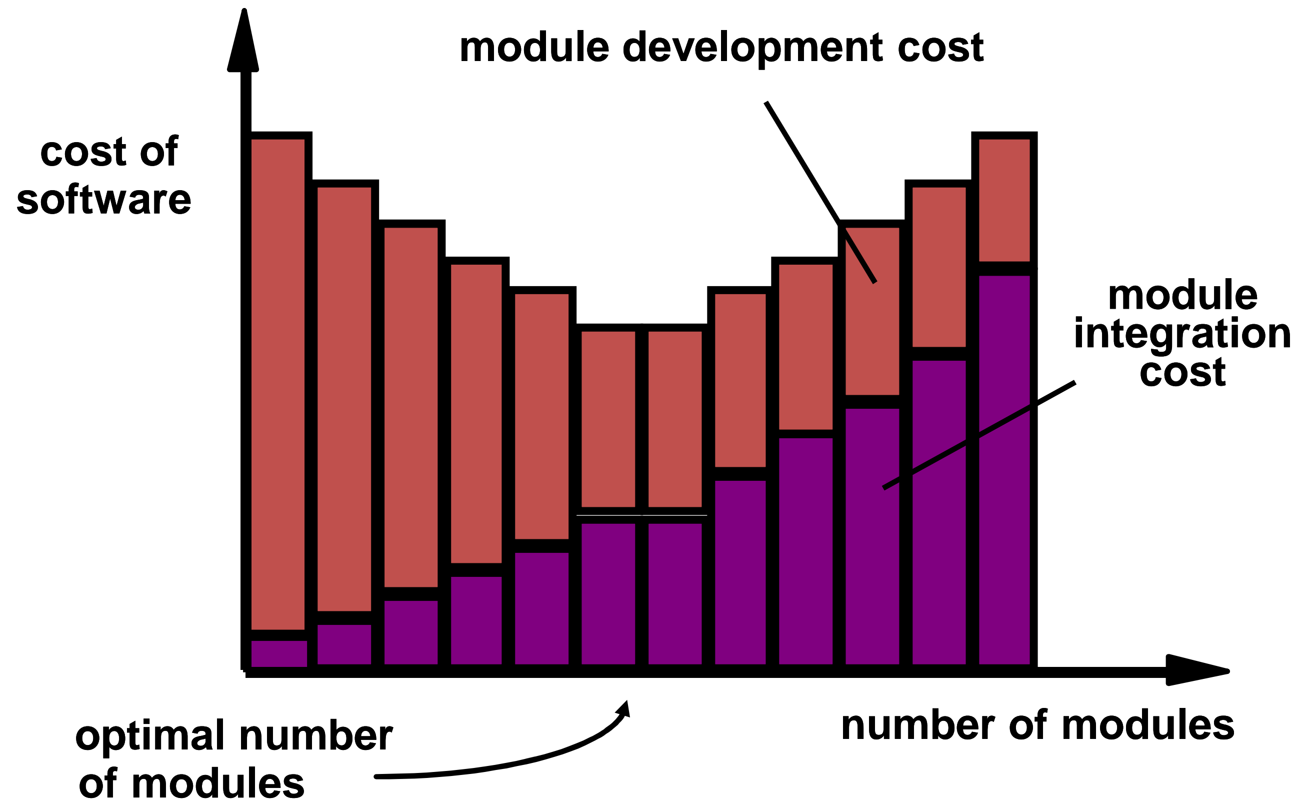
- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

Modularity

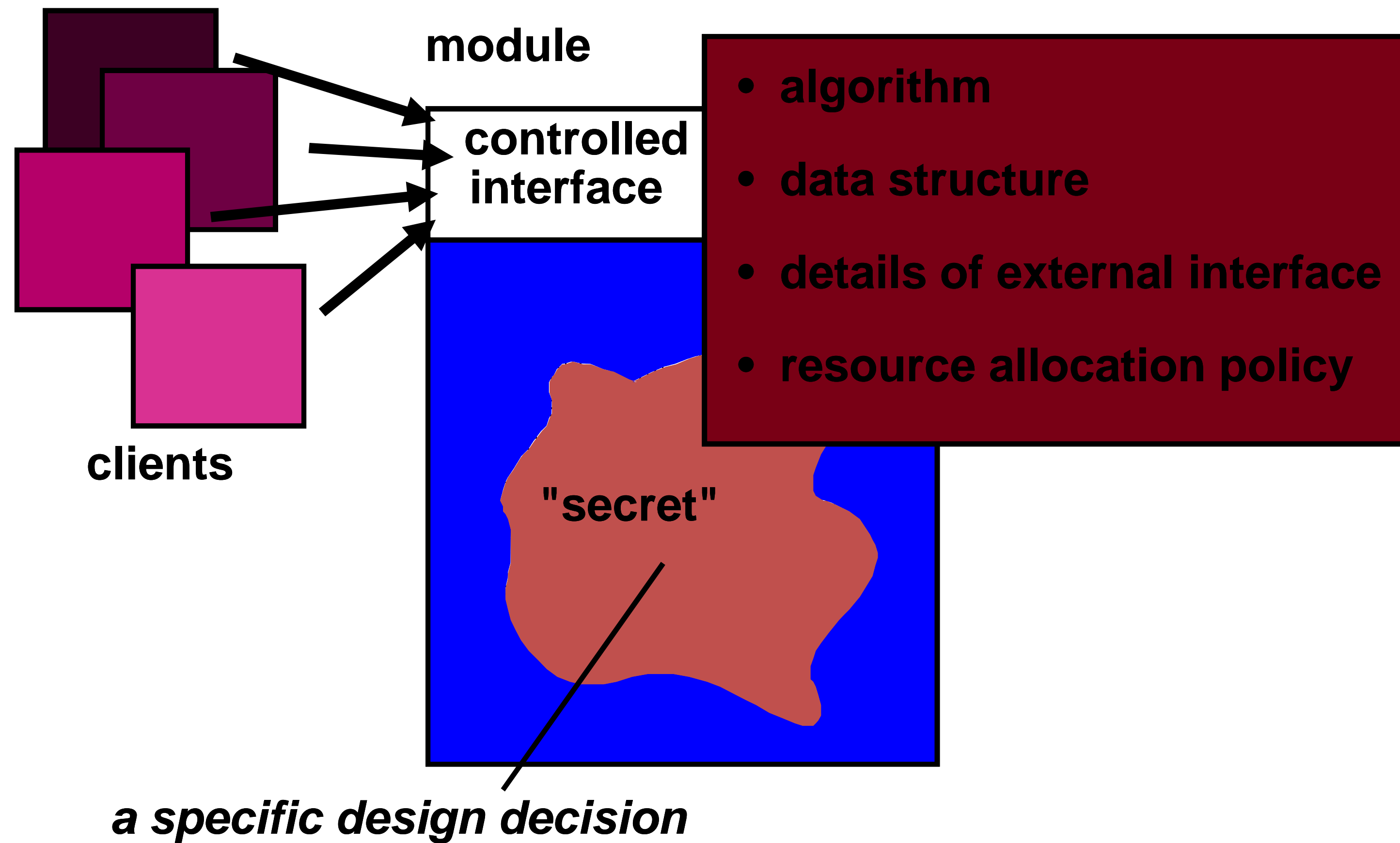
- "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

Modularity: Trade-offs

What is the "right" number of modules for a specific software design?



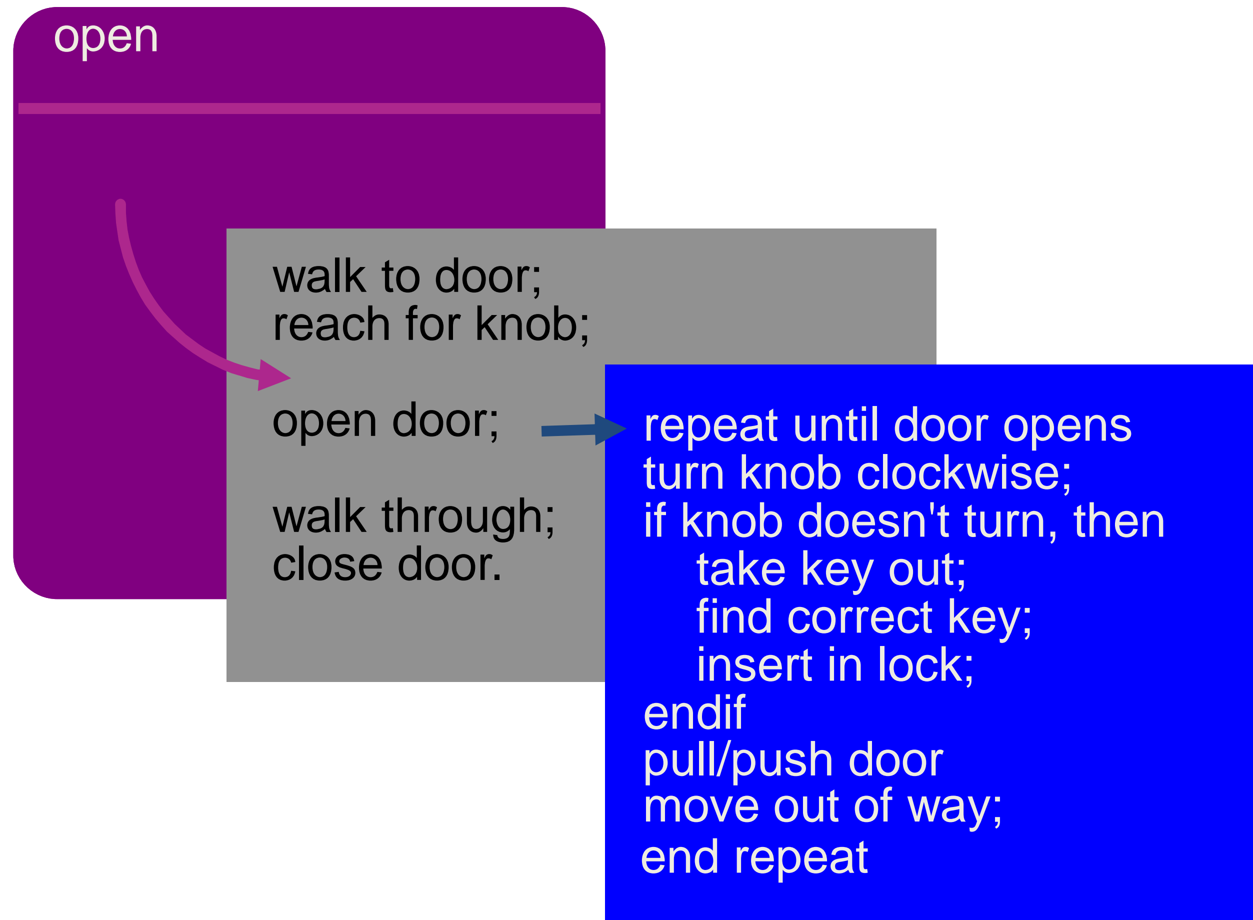
Information Hiding



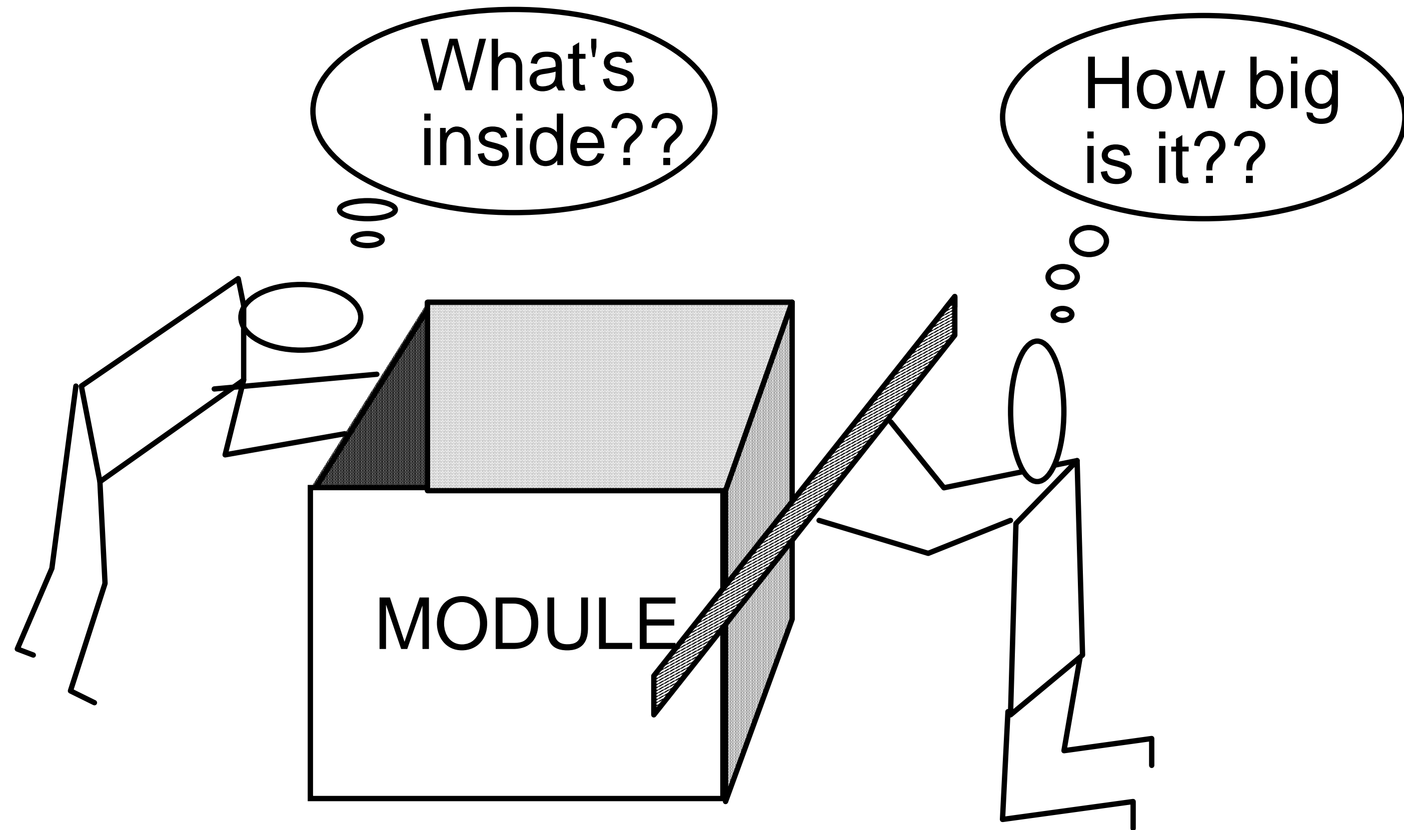
Why Information Hiding?

- reduces the likelihood of “side effects”
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

Stepwise Refinement



Sizing Modules: Two Views



Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- *Cohesion* is an indication of the relative functional strength of a module.
  - A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- *Coupling* is an indication of the relative interdependence among modules.
  - Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Aspects

- Consider two requirements, *A* and *B.* Requirement *A crosscuts* requirement *B* “if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account. [Ros04]
- An *aspect* is a representation of a cross-cutting concern.

Aspects—An Example

- Consider two requirements for the **SafeHomeAssured.com** WebApp. Requirement *A* is described via the use-case **Access camera surveillance via the Internet.** A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using* **SafeHomeAssured.com.** This requirement is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, *A** is a design representation for requirement *A* and *B** is a design representation for requirement *B*. Therefore, *A** and *B** are representations of concerns, and B* *cross-cuts* A*.
- An *aspect* is a representation of a cross-cutting concern. Therefore, the design representation, *B**, of the requirement, *a registered user must be validated prior to using* **SafeHomeAssured.com,** is an aspect of the *SafeHome* WebApp.

Refactoring

- Fowler [FOW99] defines refactoring in the following manner:
  - "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”
- When software is refactored, the existing design is examined for
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - or any other design failure that can be corrected to yield a better design.

OO Design Concepts

- Design classes
  - Entity classes
  - Boundary classes
  - Controller classes
- Inheritance—all responsibilities of a superclass is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design

Design Classes

- Analysis classes are refined during design to become entity classes
- Boundary classes are developed during design to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
  - Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- Controller class*es* are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

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

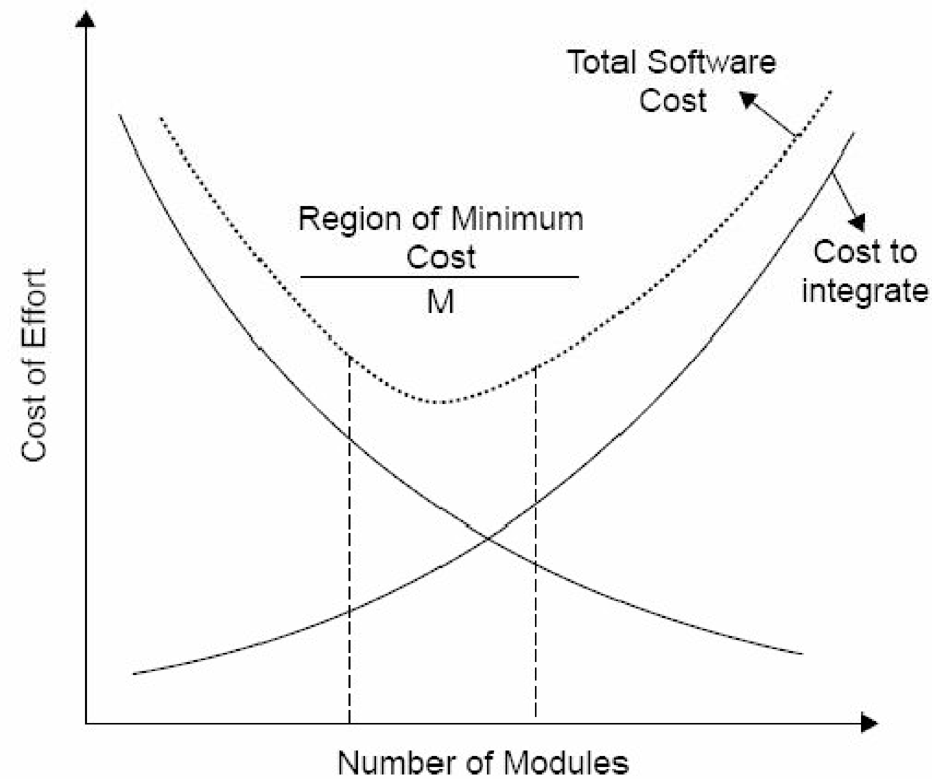
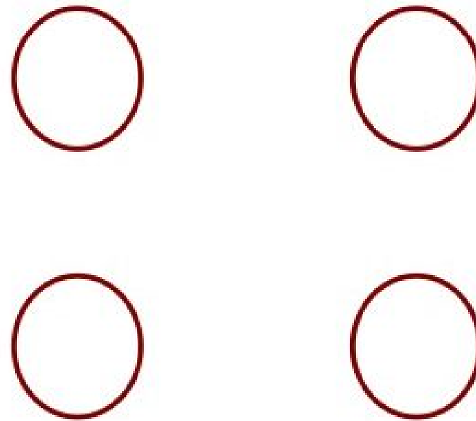


Fig. 4 : Modularity and software cost

Module Coupling

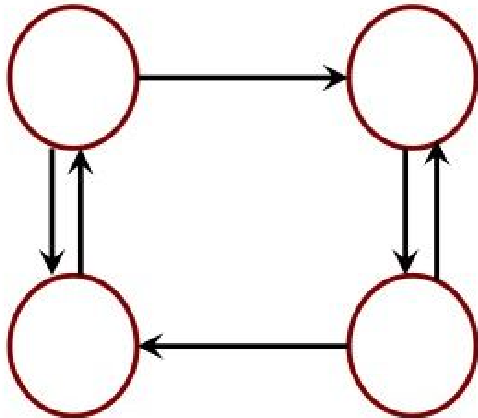
Coupling is the measure of the degree of interdependence between modules.



(Uncoupled : no dependencies)

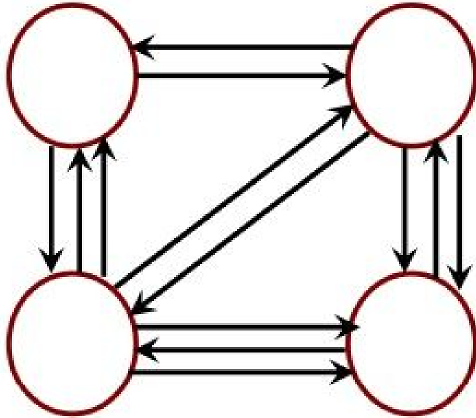
(a)

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’.

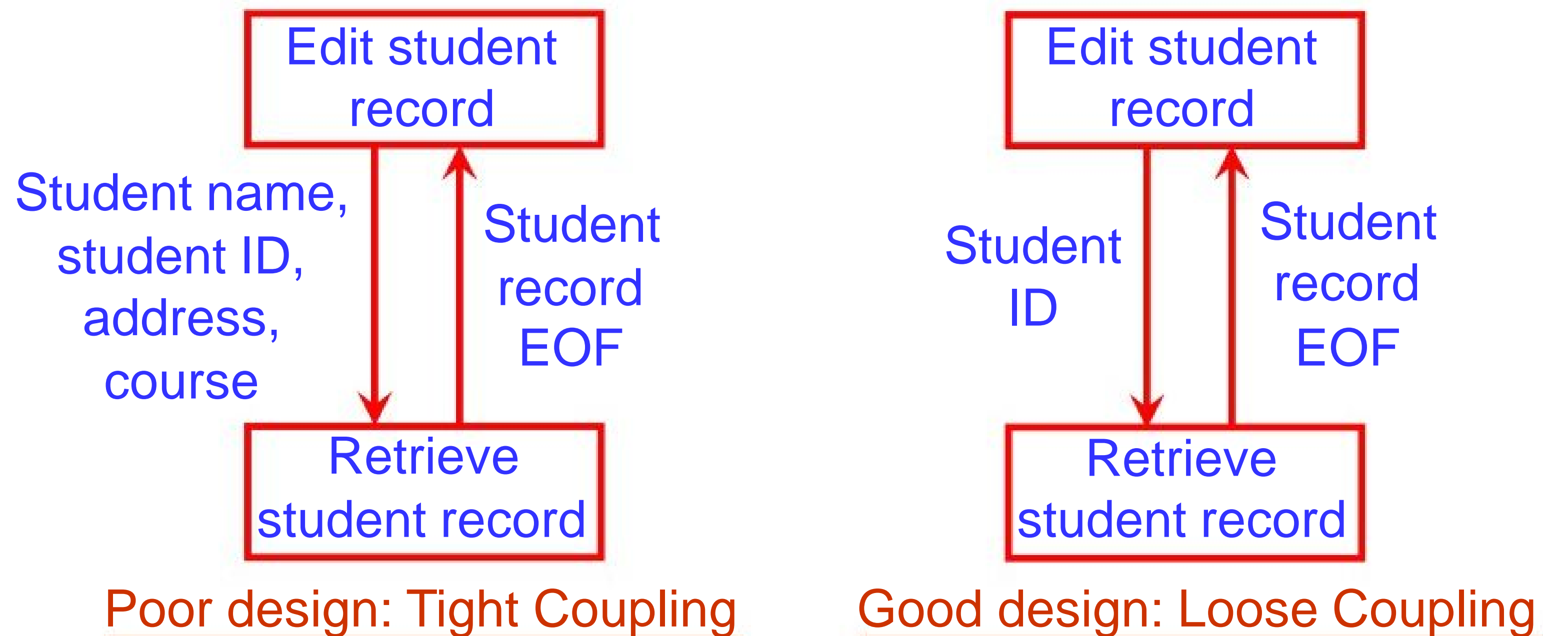
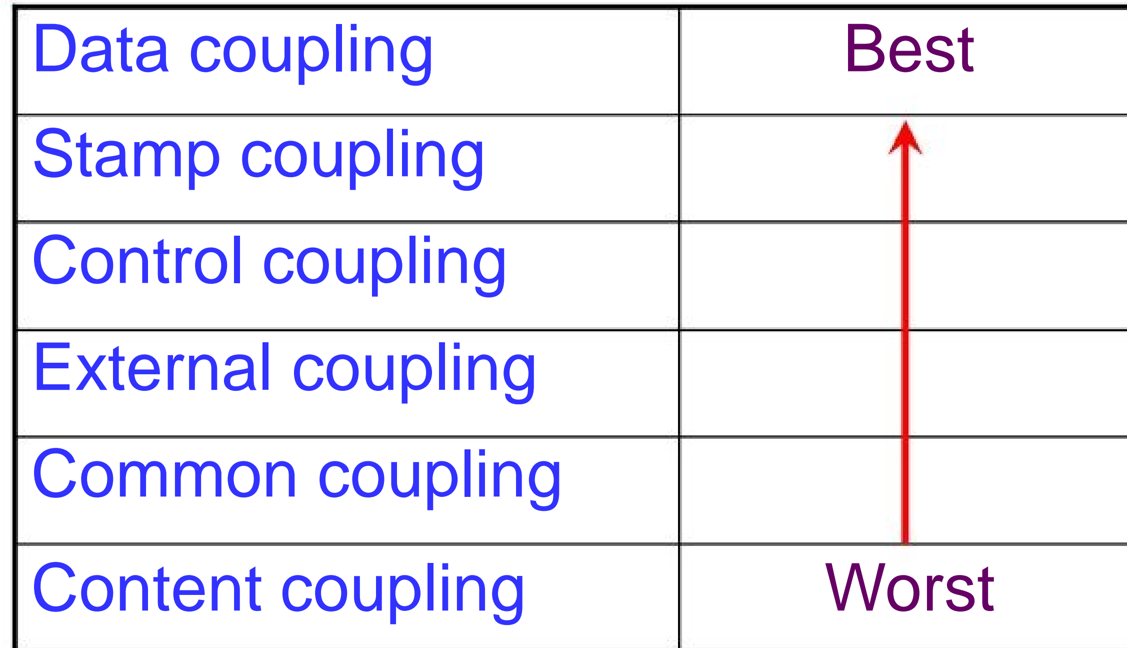


Fig. 6 : Example of coupling

Software Design



Data coupling	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.

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.

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.

Software Design

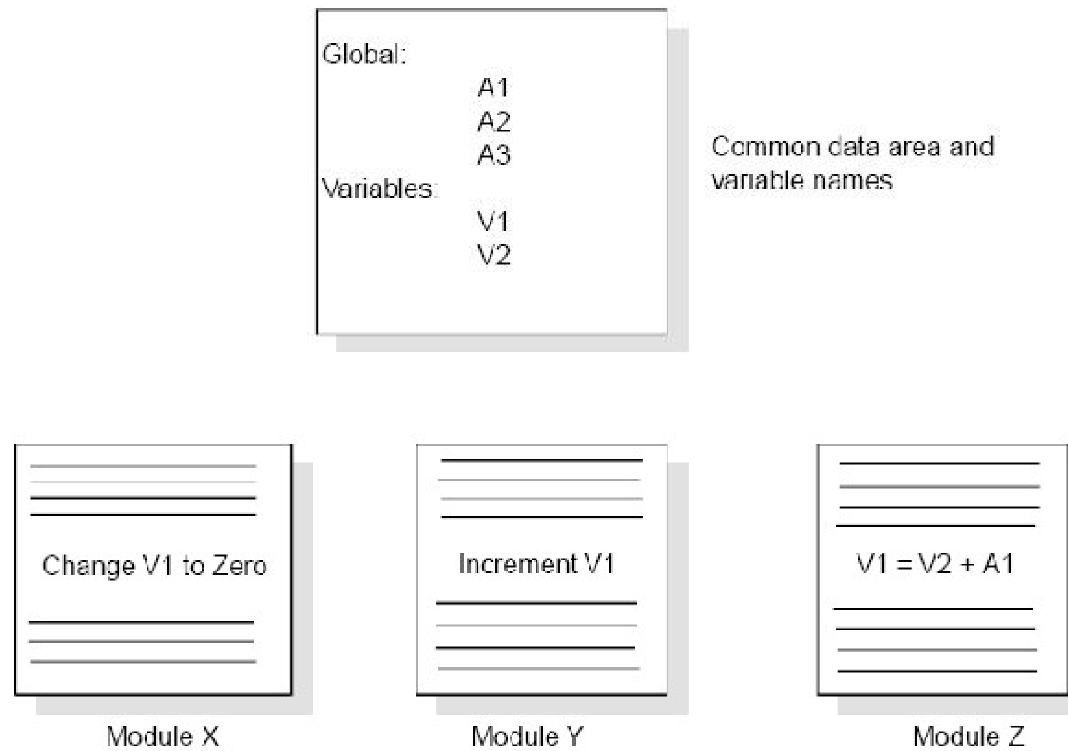


Fig. 8 : Example of common coupling

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.

Software Design

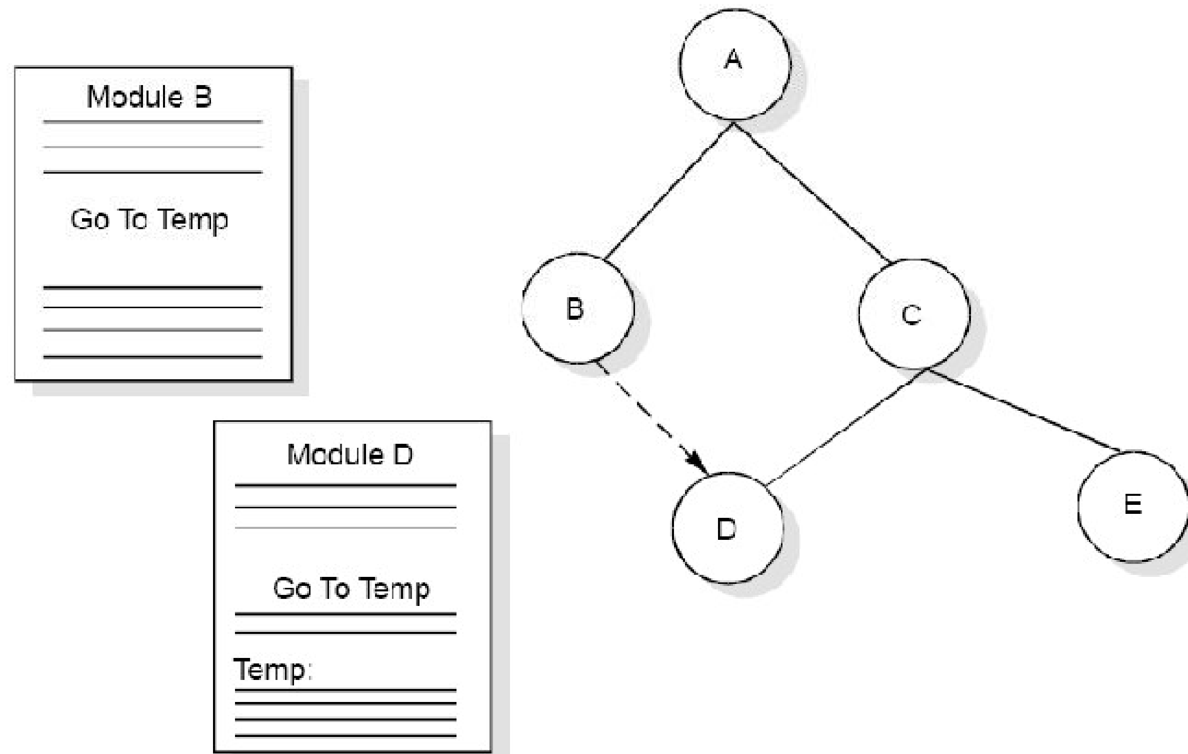
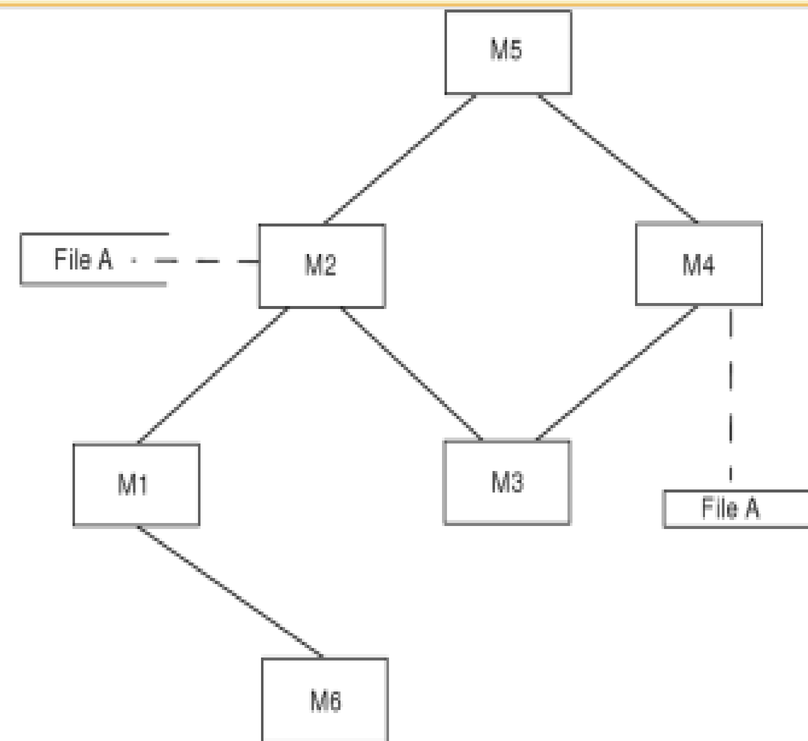


Fig. 9 : Example of content coupling

```
M1(boolean flag) { if (flag==true)call
    M6(studentRec); }
M2(int studentNum) { read studentRec from
    file A; call M1(studentRec.status); call
    M3(studentRec.studentName); }
M3(string stringToPrint) { print stringToPrint; }
M4(Record studentRec) { modify studentRec;
    update File A; call M3("record updated"); }
M5() { call M2(studentNum); call
    M4(studentRec); }
M6(studentRec) { sendEmailTo studentRec.
    studentName; }
```



	M1	M2	M3	M4	M5	M6
M1						Stamp
M2	Control		Data	Common		
M3						
M4		Common	Data			
M5		Data		Data		
M6						

Module Cohesion

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

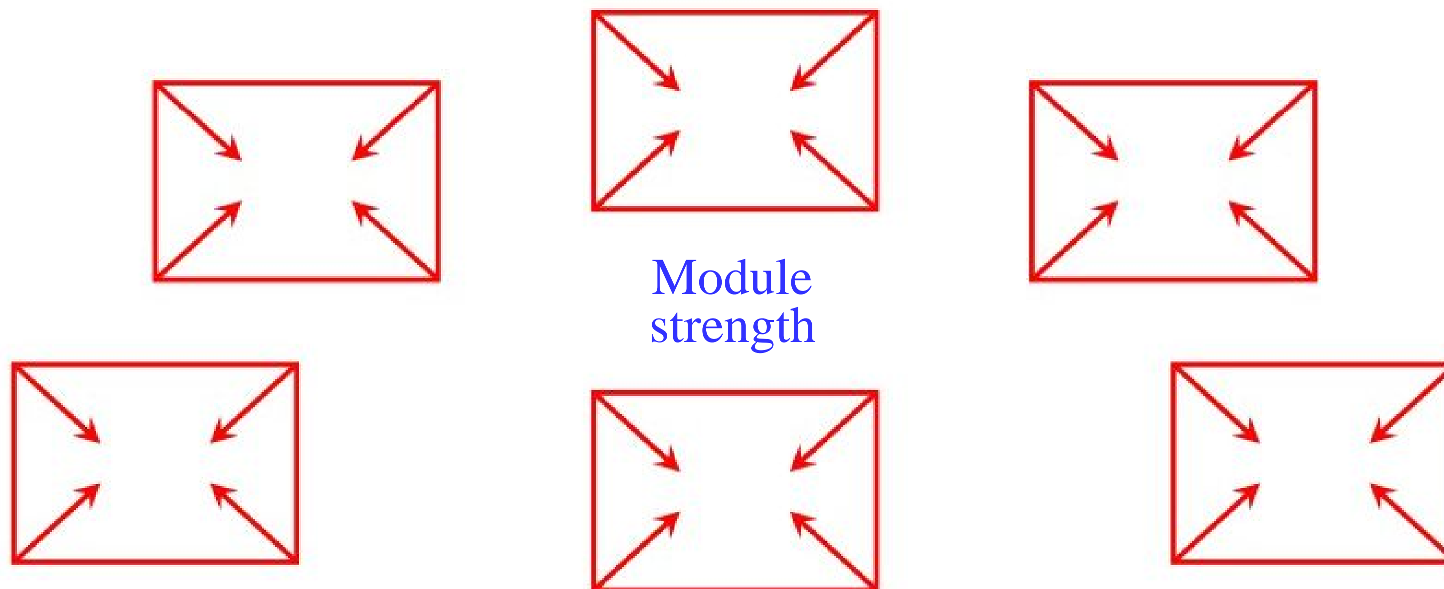
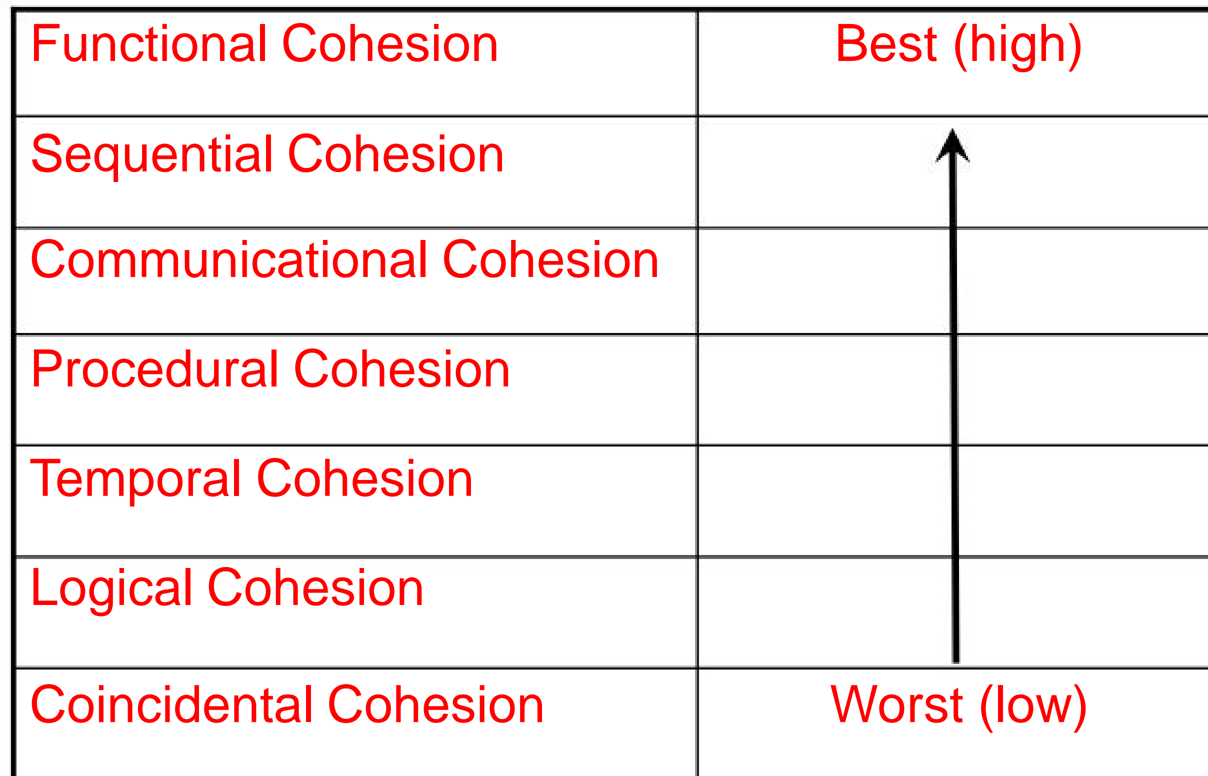


Fig. 10 : Cohesion=Strength of relations within modules

Types of cohesion

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

Software Design

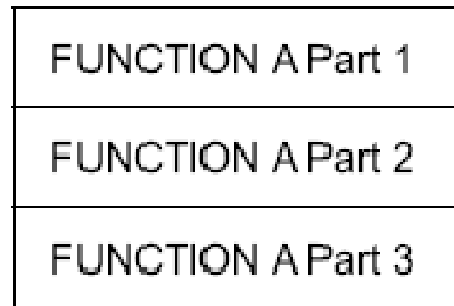


Functional Cohesion	Best (high)
Sequential Cohesion	
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	Worst (low)

Fig. 11 : Types of module cohesion

Functional Cohesion

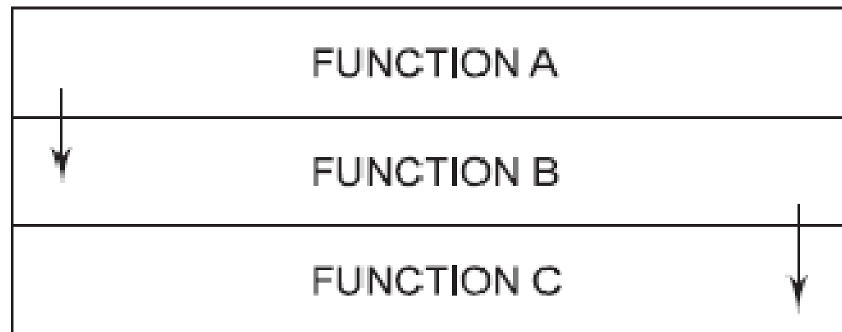
A and B are part of a single functional task. This is very good reason for them to be contained in the same procedure.



3 Functional Cohesion: Sequential with Complete, Related Functions

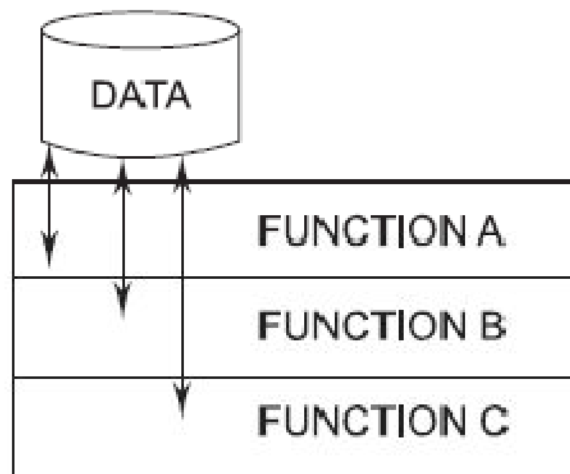
Sequential Cohesion

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



Sequential Cohesion: Output of One Part is Input to Next

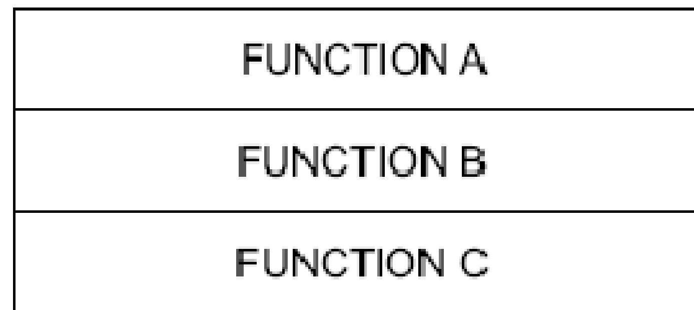
Communicational Cohesion. A module is said to have communicational cohesion if all the functions of the module refer to or update the same data structure; for example, the set of functions defined on an array or a stack.



Communicational Cohesion: Access Same Data

Procedural Cohesion

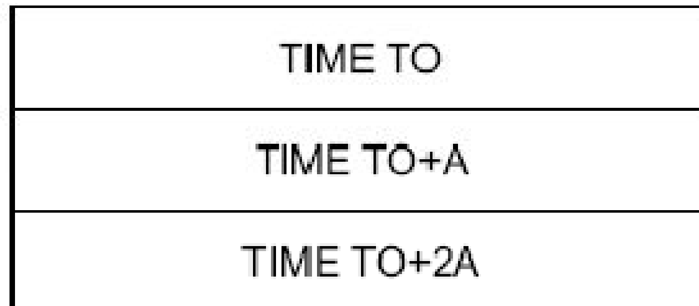
Procedural Cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed.



Procedural Cohesion Related by Order of Function

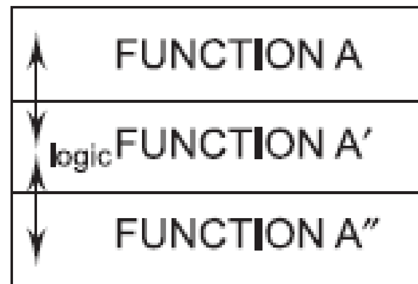
Temporal Cohesion

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.



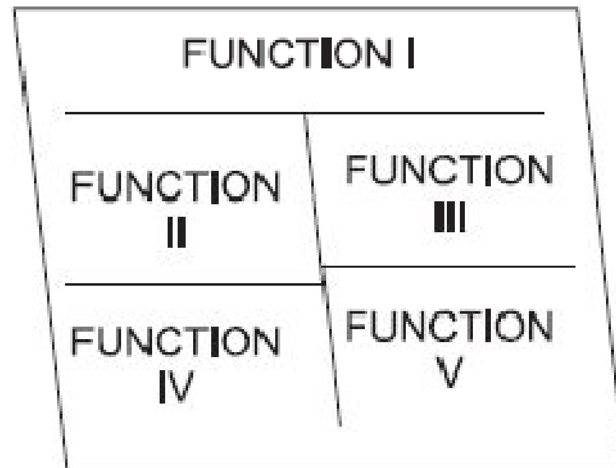
Logical Cohesion

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

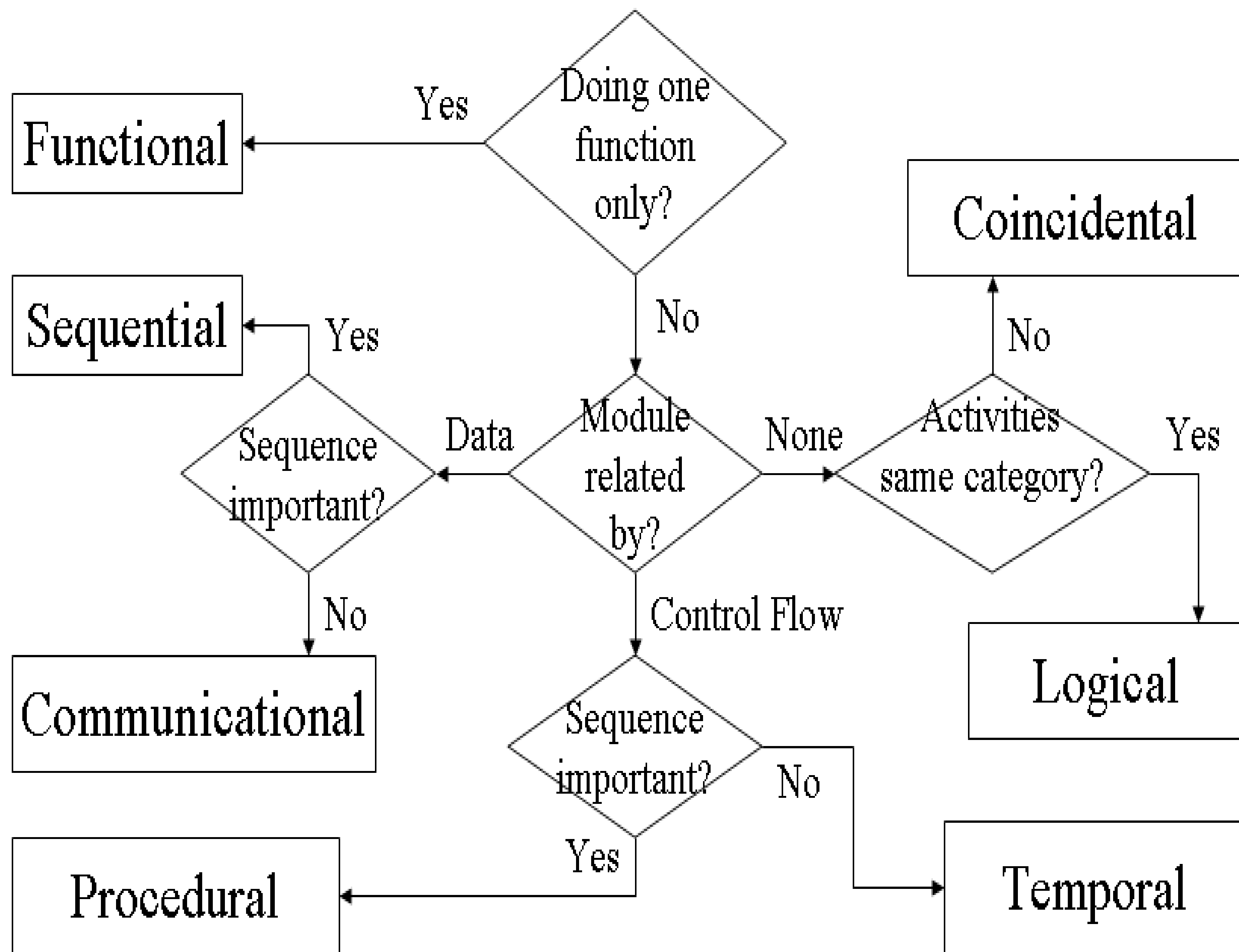
Coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another.



24 Coincidental Cohesion Parts Unrelated

Level	Cohesion	Brief description
0	Coincidental	Module performs unrelated activities that are included in the module by coincidence
1	Logical	Module performs one activity exclusively among many similar activities based on a control variable passed to the module
2	Temporal	Module includes unrelated steps that must be performed at the same time
3	Procedural	Module performs an unrelated set of activities in a specific order according to a business process or procedure
4	Communicational	Module performs many unrelated activities sharing the same input and output
5	Sequential	Module contains a sequence of activities—the output of one activity is the input needed to perform the successor activity in the sequence
6	Functional	Module contains strongly-related steps or statements that perform a single function

0 = Worst   6 = Best


Doing one function only?
Yes
Functional
No
Module related by?
Data
Sequence important?
Yes
Sequential
No
Communicational
None
Activities same category?
No
Coincidental
Yes
Logical
Control Flow
Sequence important?
Yes
Procedural
No
Temporal

P1: What is the effect of cohesion on maintenance?

P2: What is the effect of coupling on maintenance?

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.

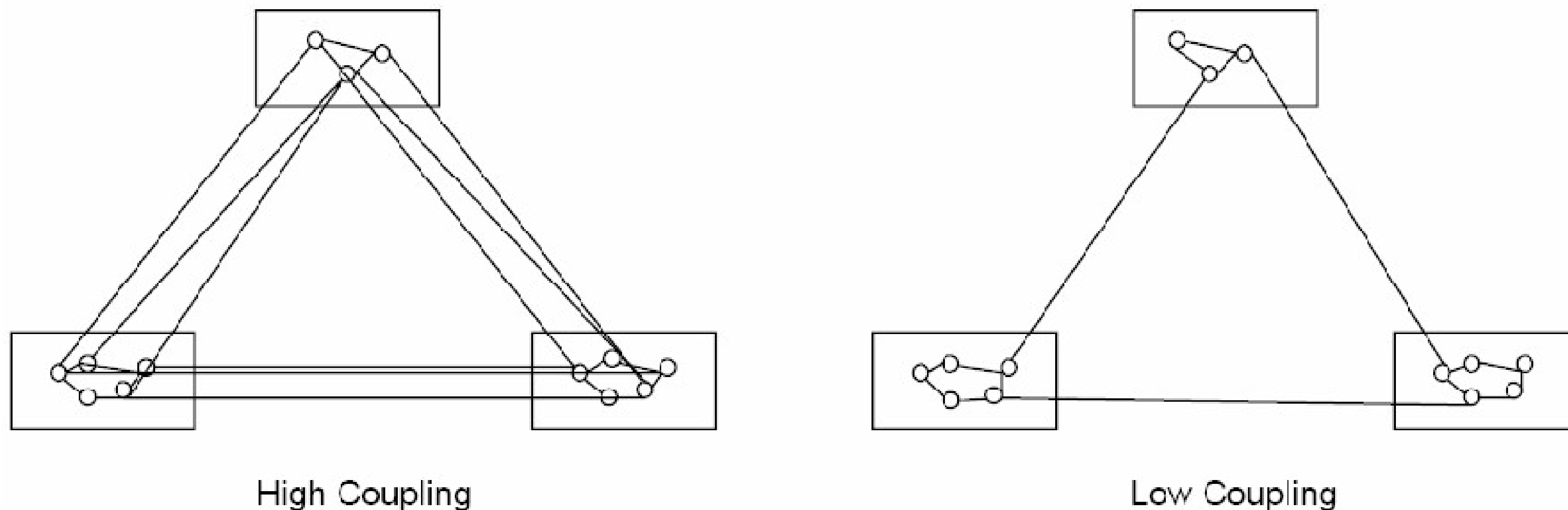
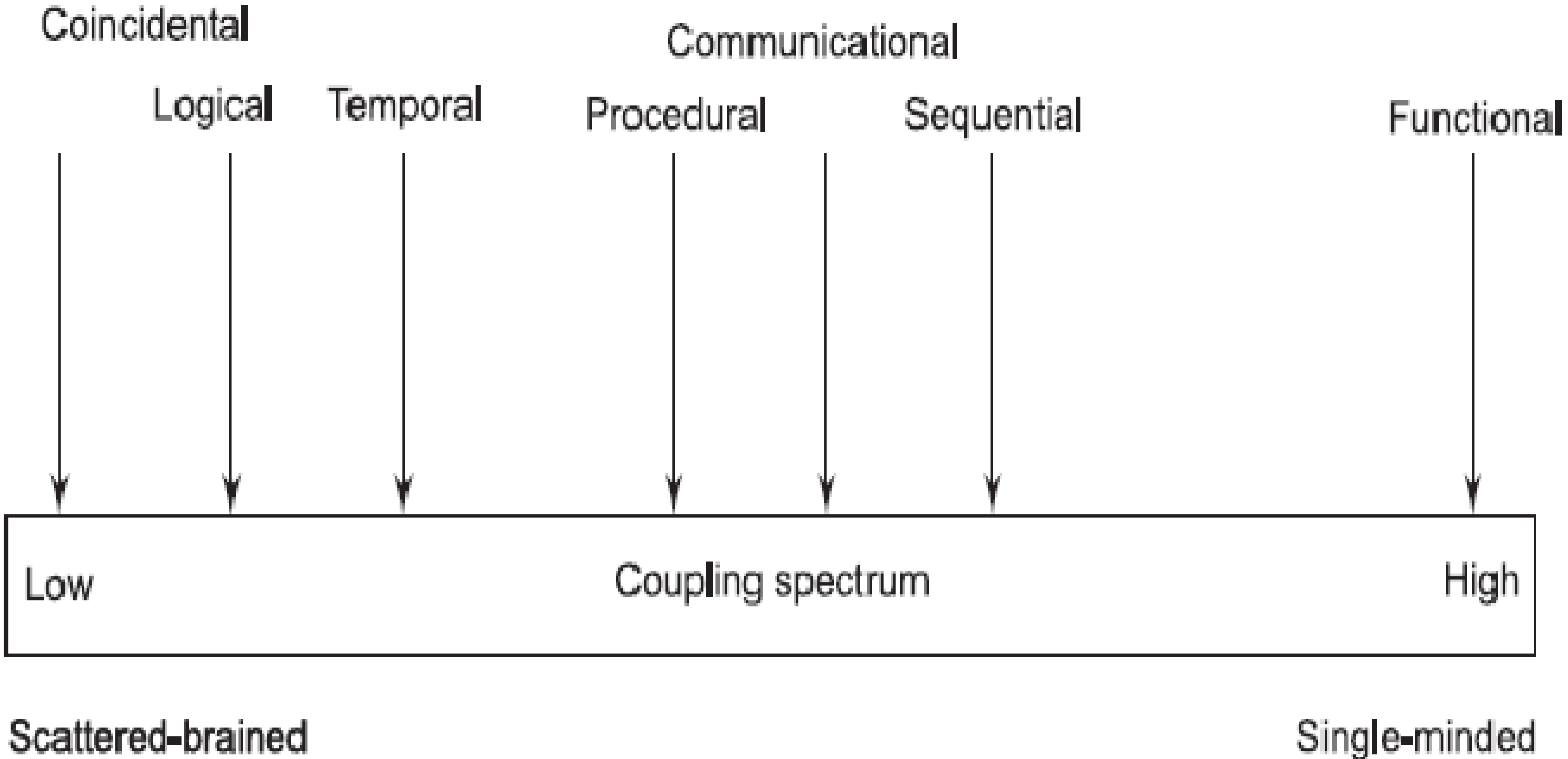


Fig. 12 : View of cohesion and coupling


Coincidental
Communicational
Logical
Temporal
Procedural
Sequential
Functional
Low
Coupling spectrum
High
Scattered-brained
Single-minded

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 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”.

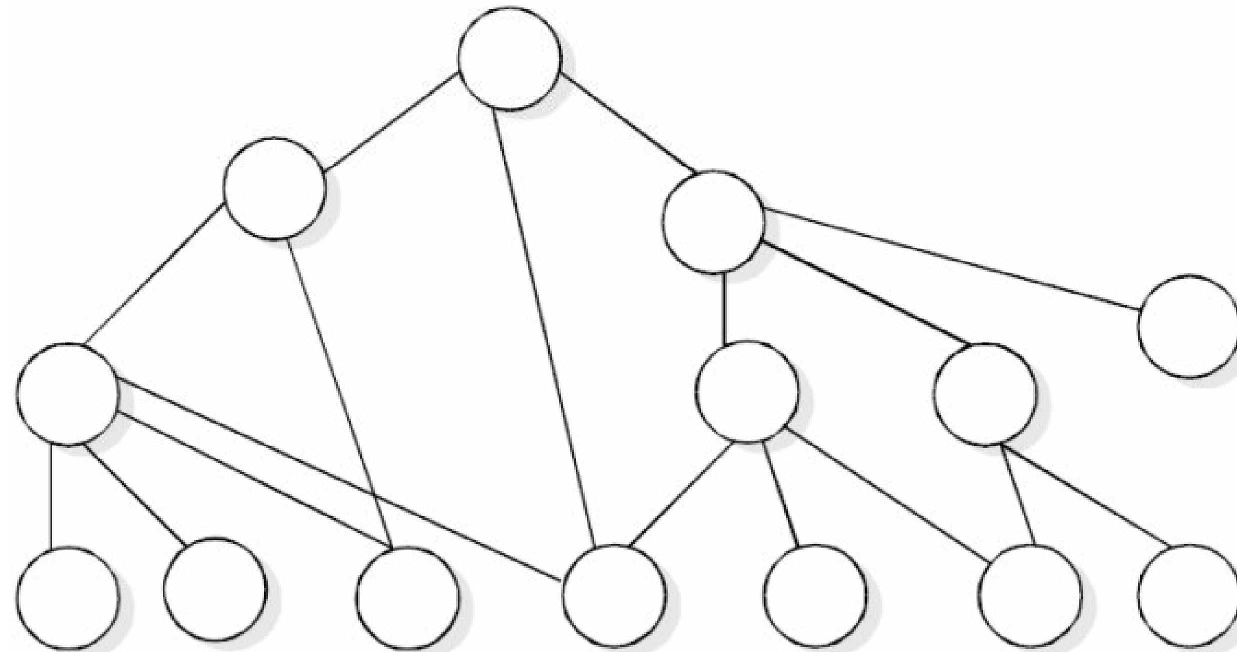


Fig. 13 : Bottom-up tree structure

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

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.

Design Heuristics

- Evaluate 1st iteration to reduce coupling & improve cohesion
- Minimize structures with high fan-out; strive for depth
- Keep scope of effect of a module within scope of control of that module
- Evaluate interfaces to reduce complexity and improve consistency

Design Heuristics

- Define modules with predictable function & avoid being overly restrictive
  - Avoid static memory between calls where possible
- Strive for controlled entry -- no jumps into the middle of things
- Package software based on design constraints and portability requirements

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 → 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.

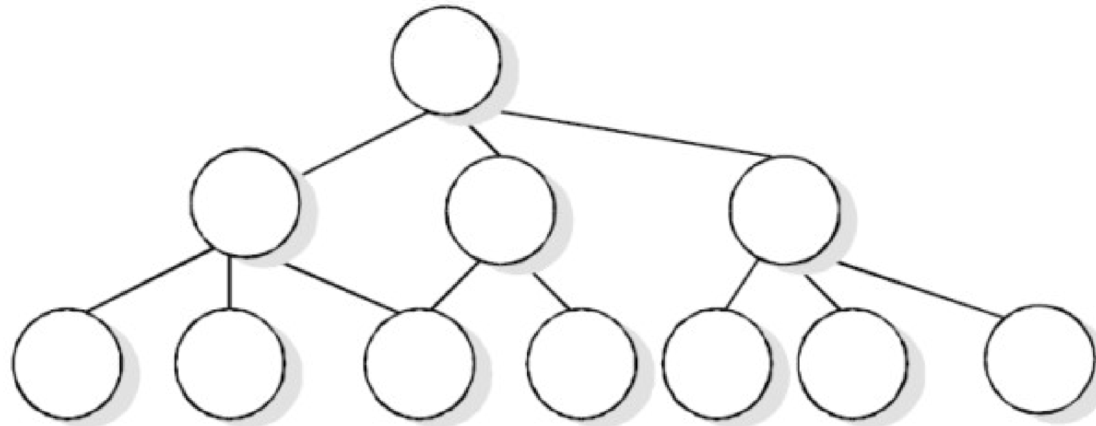


Fig. 14 : Top-down structure

Software Design

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.

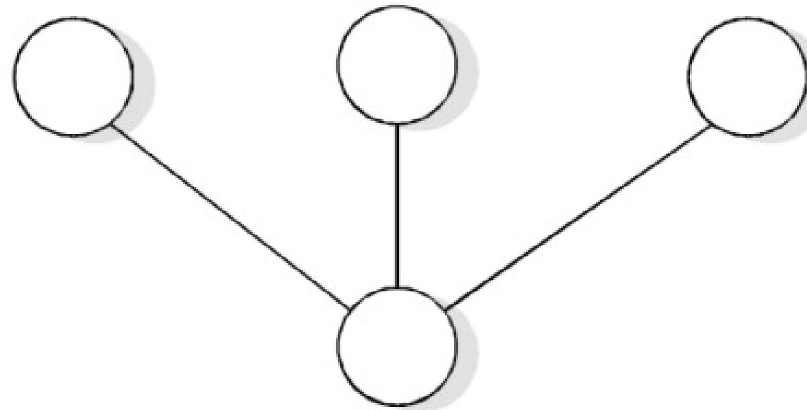


Fig. 15 : Design reusable structure

The number of levels of a component in the structure is called depth and the number of components across the horizontal section is called width. The number of components, which controls the component, is called fan-in, i.e., the number of incoming edges to a component. The number of components that are controlled by the module is called fan-out, i.e., the number of outgoing edges.

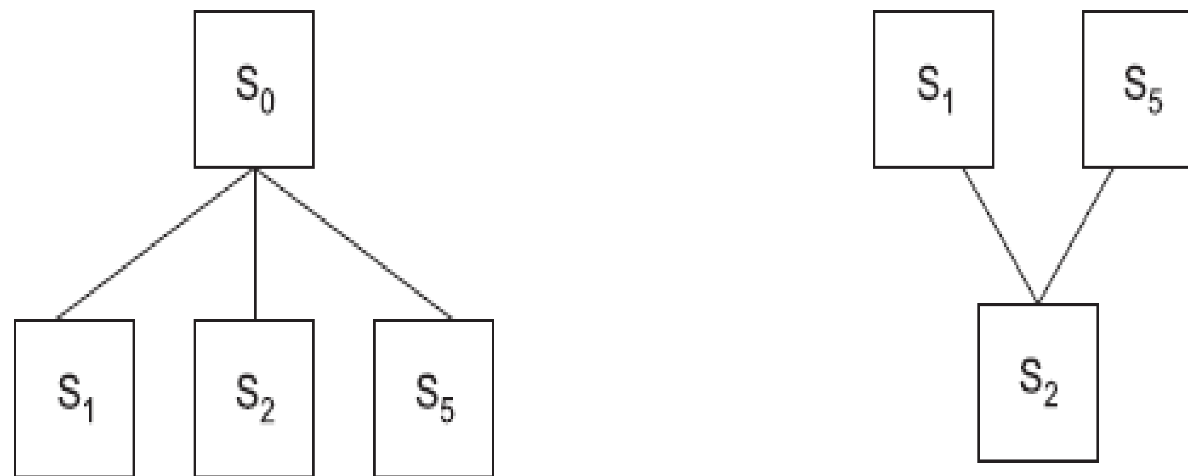


FIGURE Fan-in and Fan-out

S_0 controls three components, hence, the fan-out is 3. S_2 is controlled by two components, namely, S_1 and S_2, hence, the fan-in is 2 (see Figure).

S. No.	Functional-oriented Approach	Object-oriented Approach
1.	In the functional-oriented design approach, the basic abstractions, which are given to the user, are real-world functions, such as sort, merge, track, display, etc.	In the object-oriented design approach, the basic abstractions are not the real-world functions, but are the data abstraction where the real-world entities are represented, such as picture, machine, radar system, customer, student, employee, etc.
2.	In function-oriented design, functions are grouped together by which a higher-level function is obtained. An example of this technique is SA/SD.	In this design, the functions are grouped together on the basis of the data they operate on, such as in class person, function displays are made member functions to operate on its data members such as the person name, age, etc.
3.	In this approach, the state information is often represented in a centralized shared memory.	In this approach, the state information is not represented in a centralized shared memory but is implemented/distributed among the objects of the system.

Software Architecture

- What Is Architecture?
  - The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.
  - The architecture is not the operational software.

Definitions

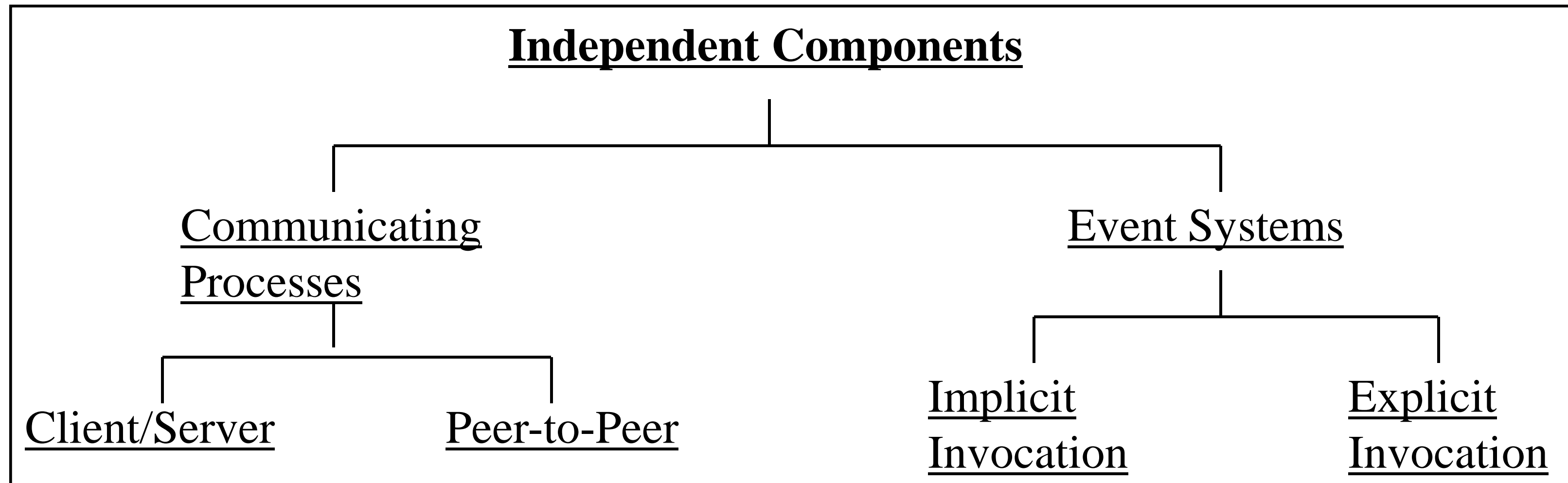
- The <u>software architecture</u> of a program or computing system is the structure or structures of the system which <u>comprise</u>
  - The software <u>components</u>
  - The externally visible <u>properties</u> of those components
  - The <u>relationships</u> among the components
- <u>Software architectural design</u> represents the <u>structure</u> of the data and program <u>components</u> that are required to build a computer-based system
- An architectural design model is <u>transferable</u>
  - It can be <u>applied</u> to the design of other systems
  - It <u>represents</u> a set of <u>abstractions</u> that enable software engineers to describe architecture in <u>predictable</u> ways

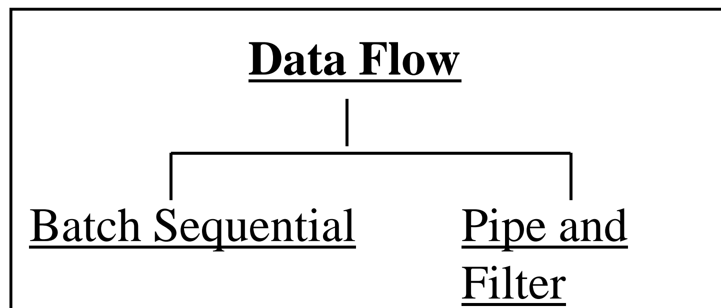
Architectural Styles

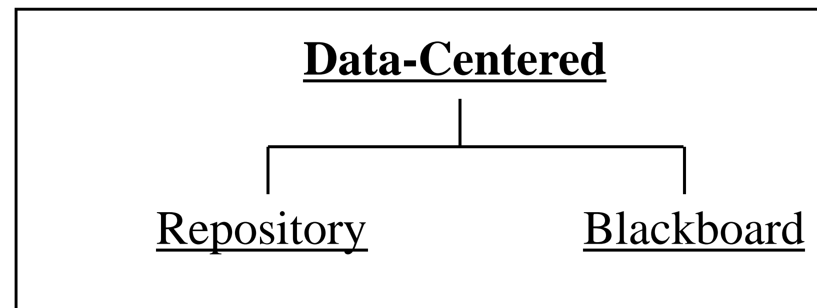
Each style describes a system category that encompasses: (1) a set of components (e.g., a database, computational modules) that perform a function required by a system, (2) a set of connectors that enable "communication, coordination and cooperation" among components, (3) constraints that define how components can be integrated to form the system, and (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

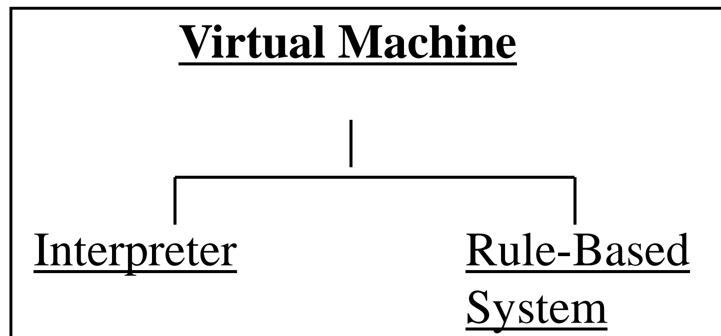
- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

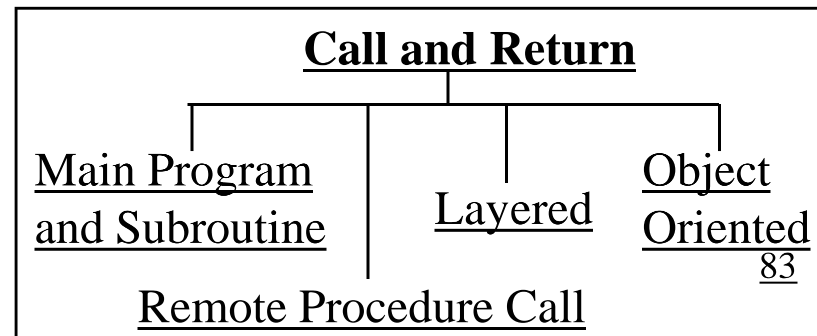
A Taxonomy of Architectural Styles



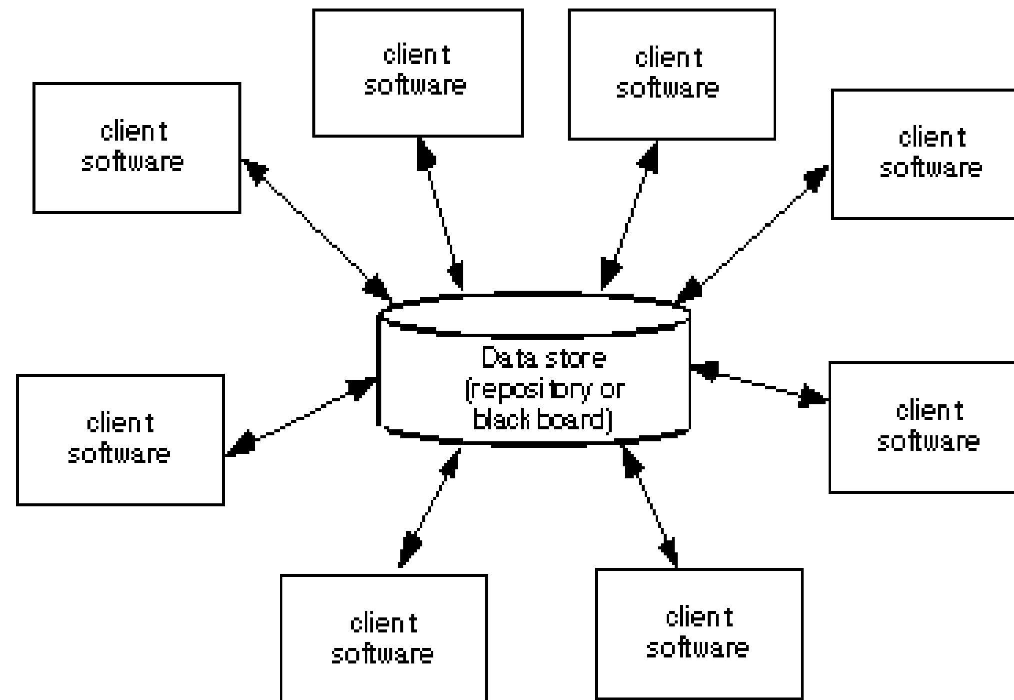








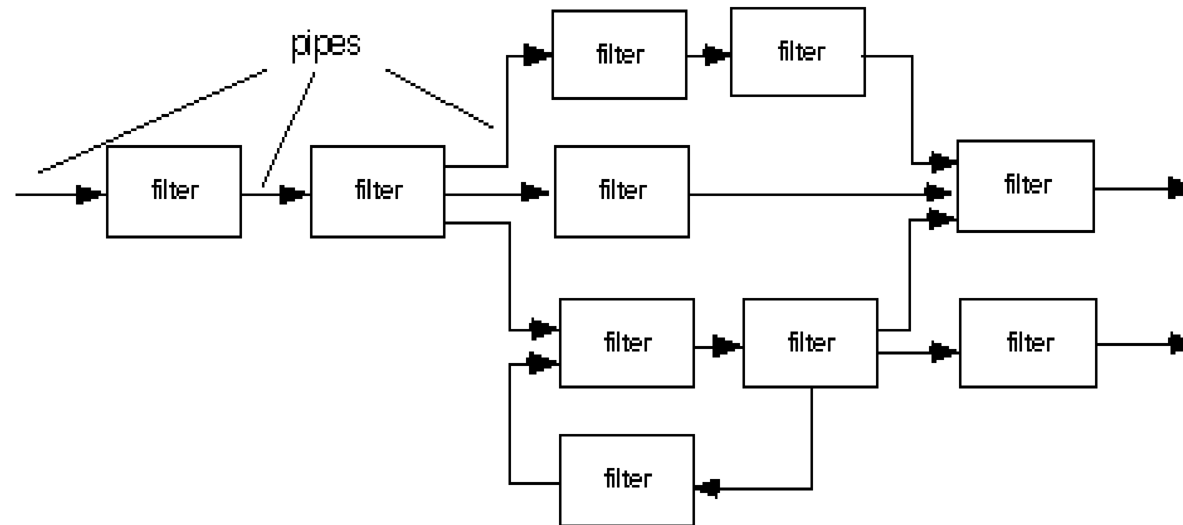
Data-Centered Architecture



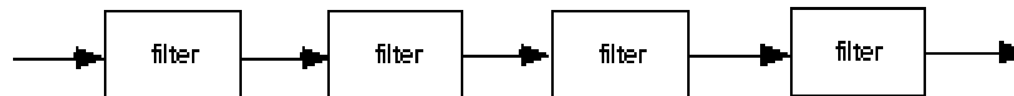
Data Flow Style

- Has the goal of modifiability
- Characterized by viewing the system as a series of transformations on successive pieces of input data
- Data enters the system and then flows through the components one at a time until they are assigned to output or a data store
- Batch sequential style
  - The processing steps are independent components
  - Each step runs to completion before the next step begins
- Pipe-and-filter style
  - Emphasizes the incremental transformation of data by successive components
  - The filters incrementally transform the data (entering and exiting via streams)
  - The filters use little contextual information and retain no state between instantiations
  - The pipes are stateless and simply exist to move data between filters

Data Flow Architecture

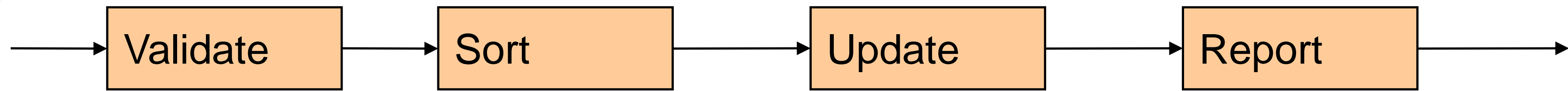


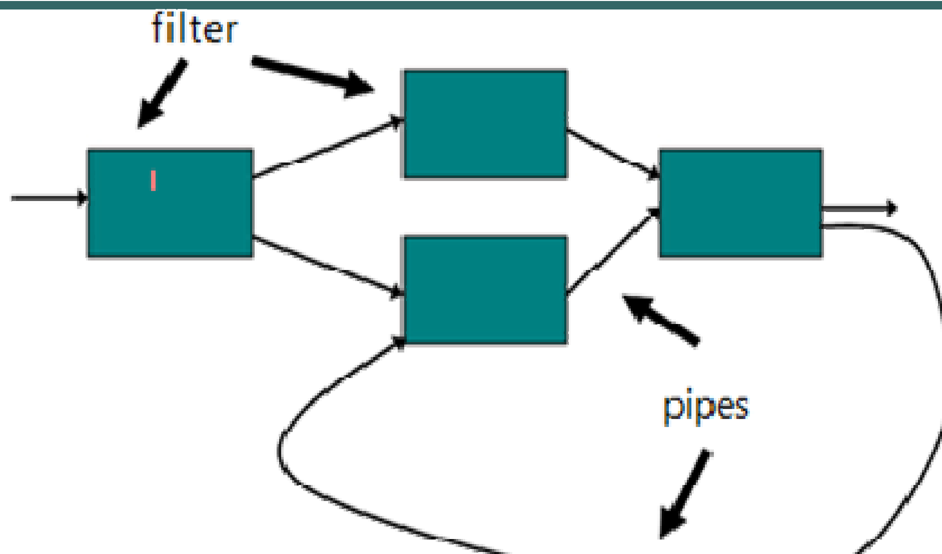
(a) pipes and filters



(b) batch sequential

Data Flow Style

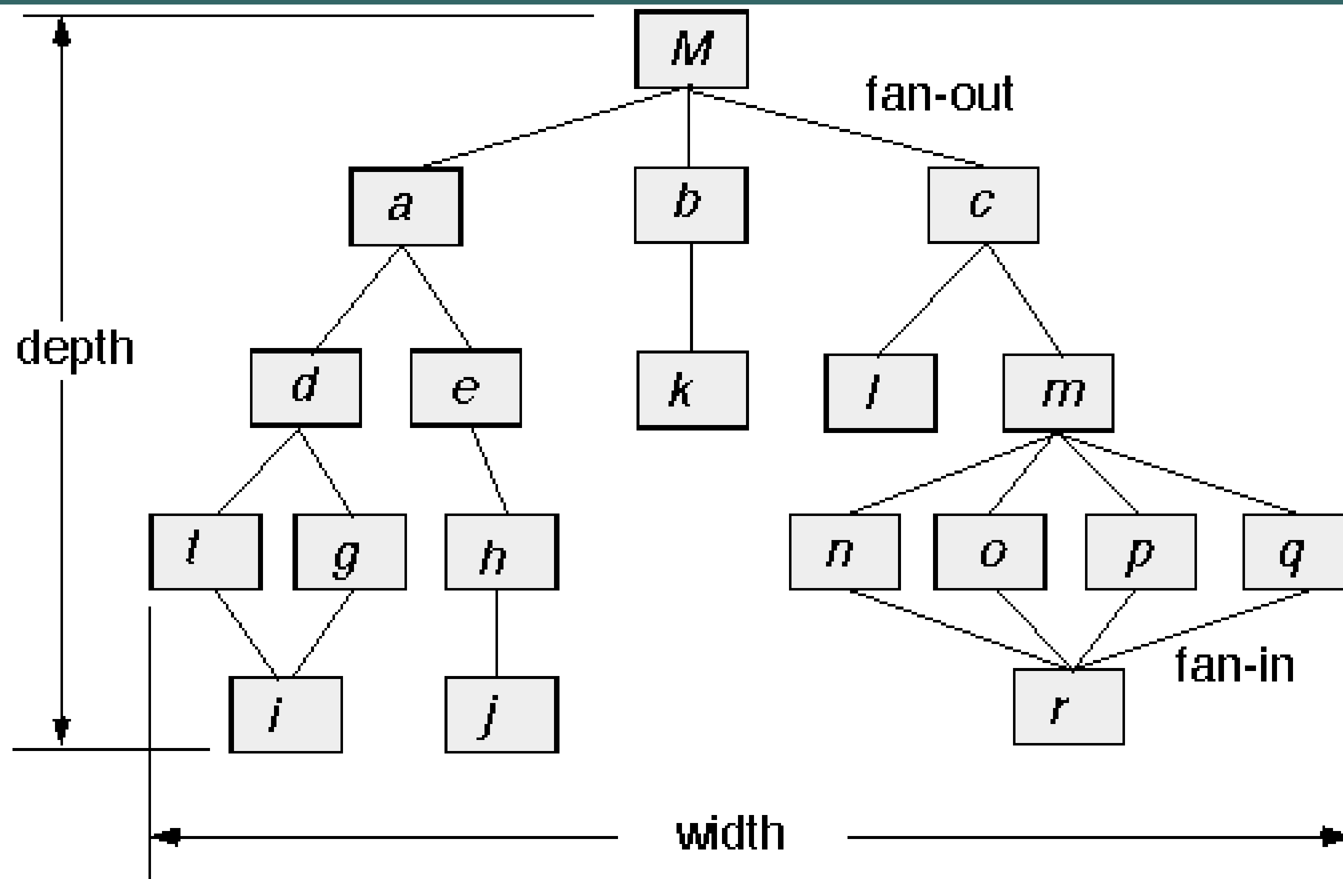




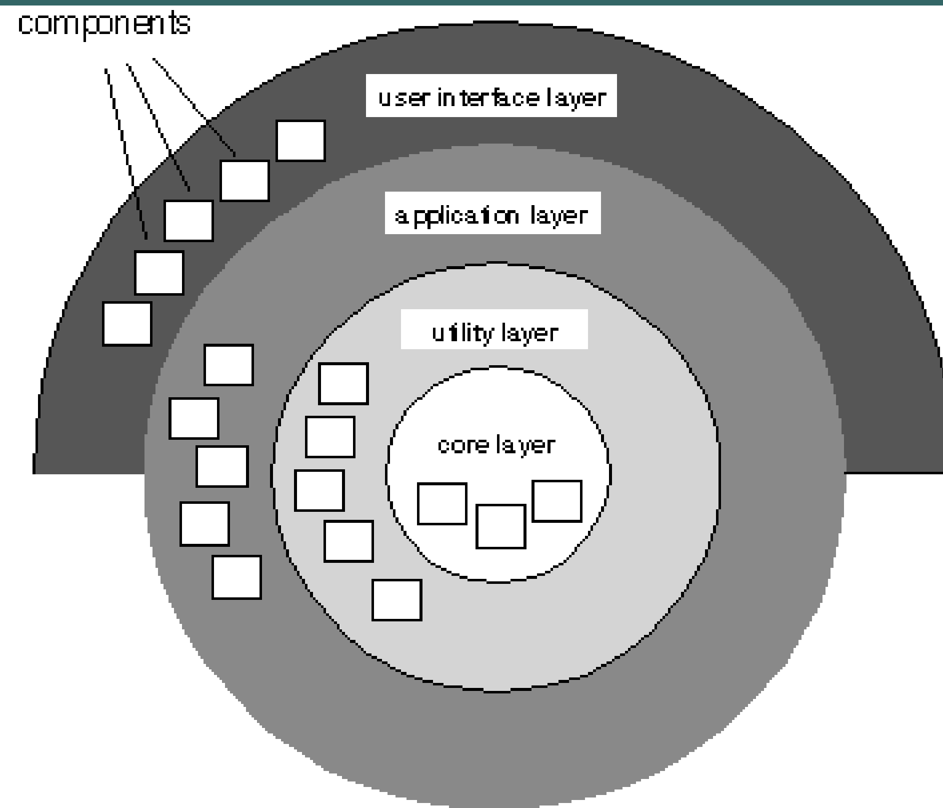
Compilation phases are pipelined, though the phases are not always incremental. The phases in the pipeline include:

– lexical analysis + parsing + semantic analysis + code generation

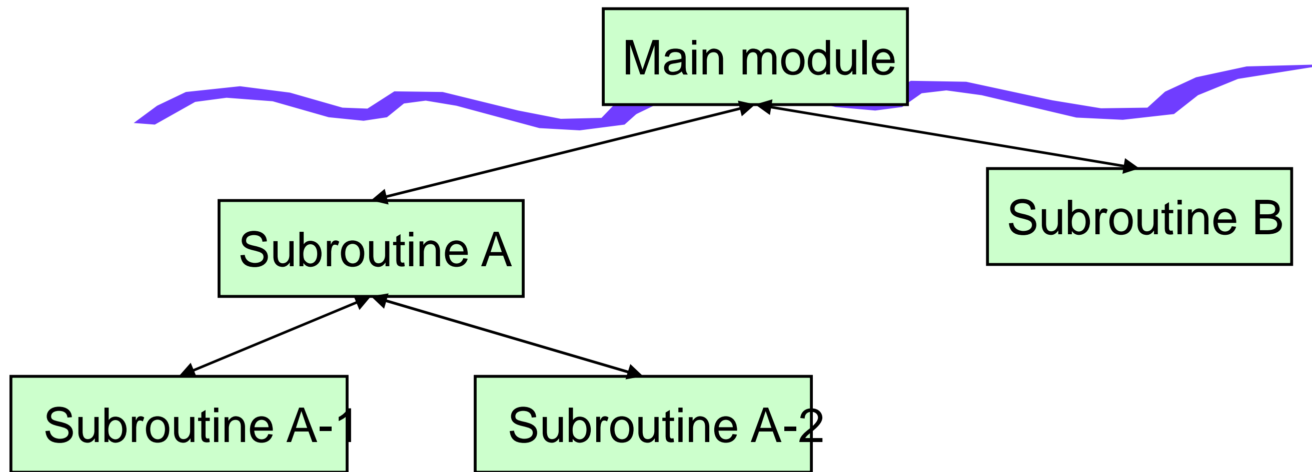
Call and Return Architecture

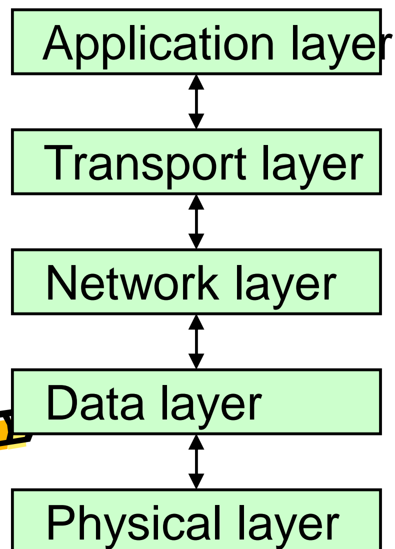


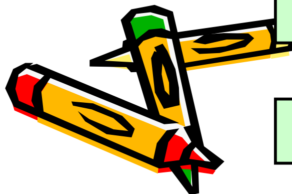
Layered Architecture

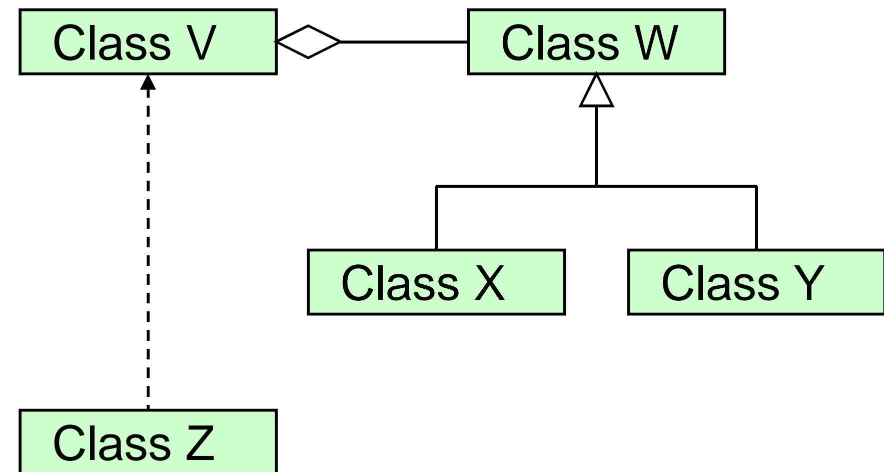


Call-and-Return Style









Software Design

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.

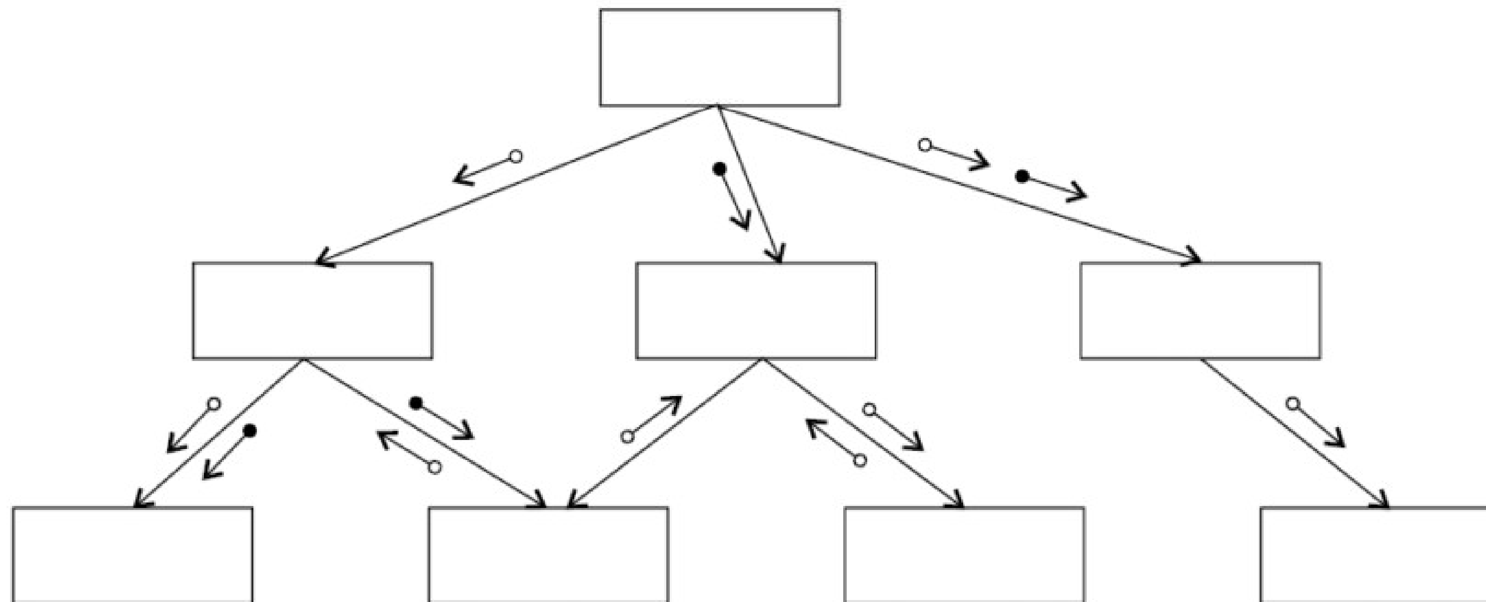
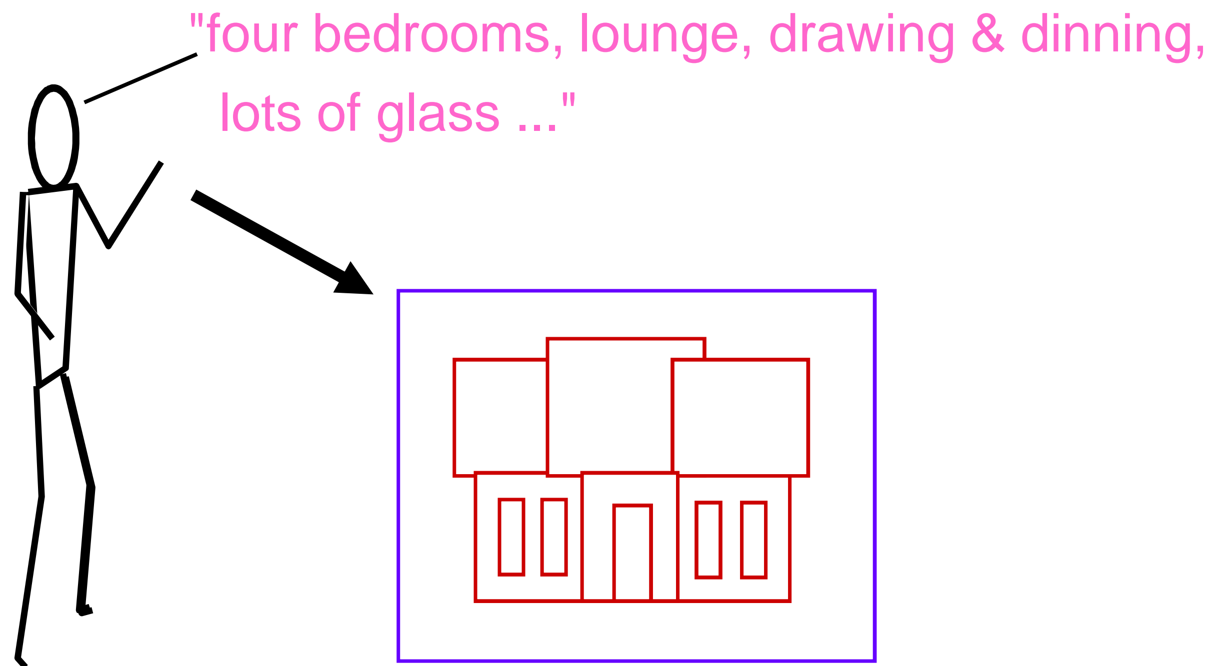


Fig. 16 : Hierarchical format of a structure chart

Mapping Requirements into a Software Architecture

An Architectural Design Method

customer requirements



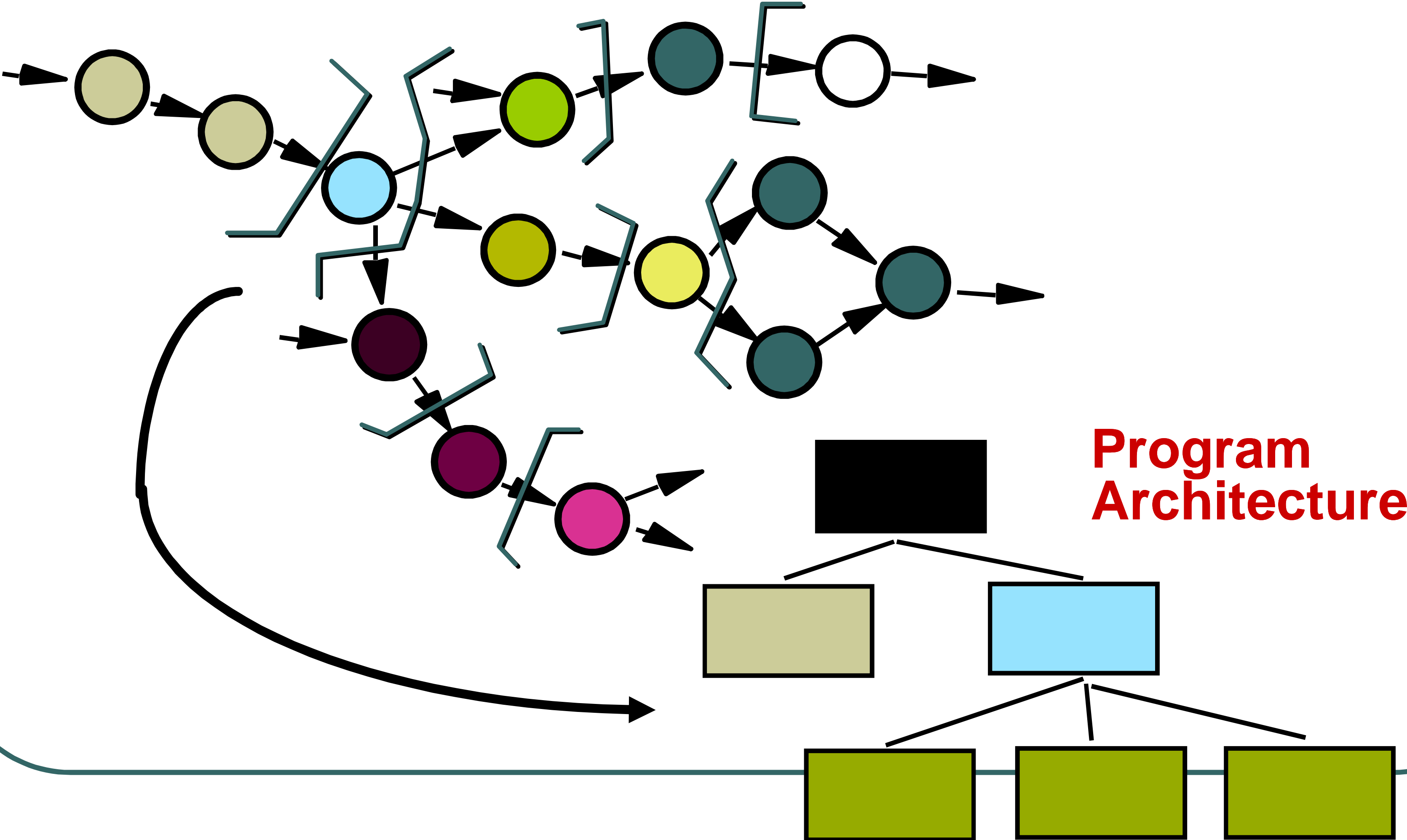
Mapping Requirements Into Software Architecture

Earlier had said certain models can be mapped directing into an architectural design

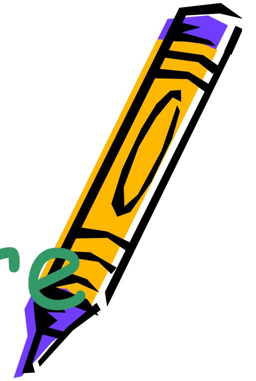
- methods do not exist for all architectural styles
- Will look at 1 approach for the ***call & return*** architecture sometimes called ***structured design*** - origins in ***top-down design*** ,structured programming ,

- **SD is a data-flow oriented design method**
  - **Provides a method to go from a DFD to program structure**

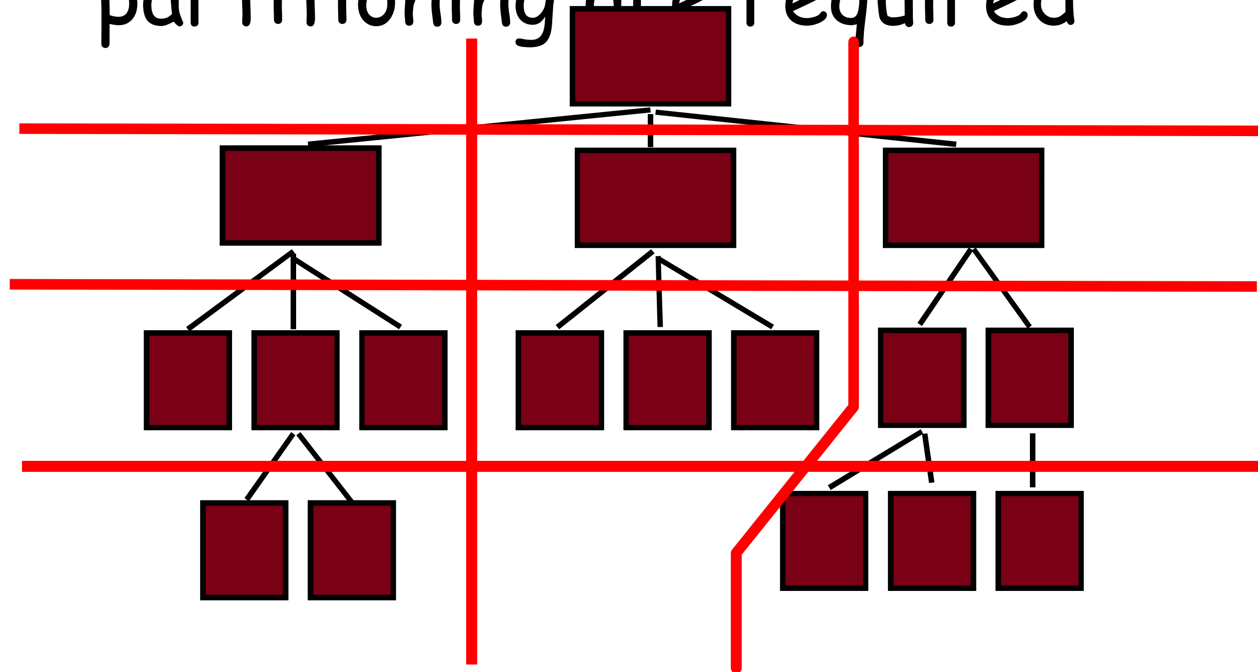
Deriving Program Architecture



Partitioning the Architecture

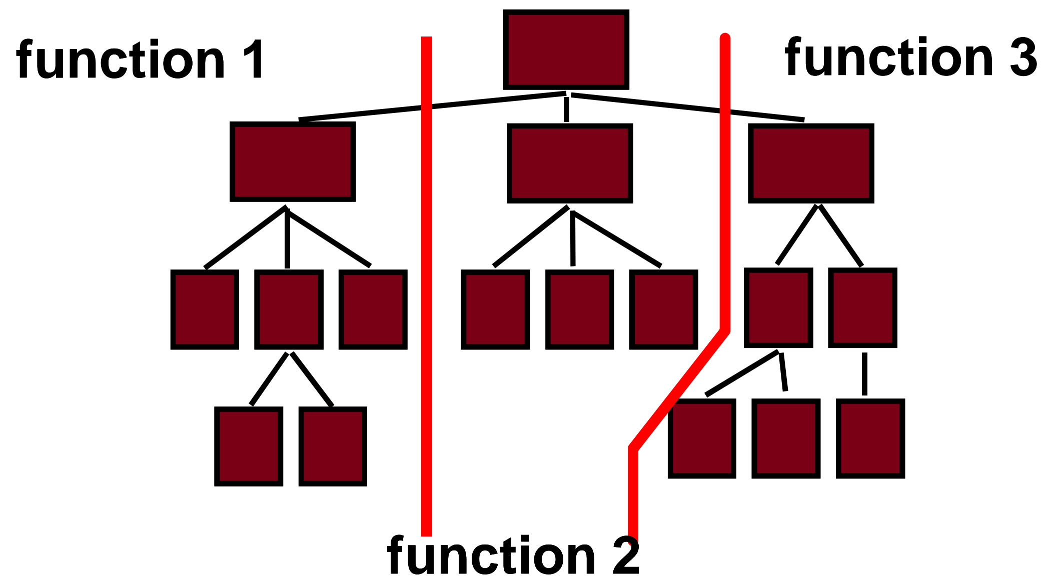


- "horizontal" and "vertical" partitioning are required



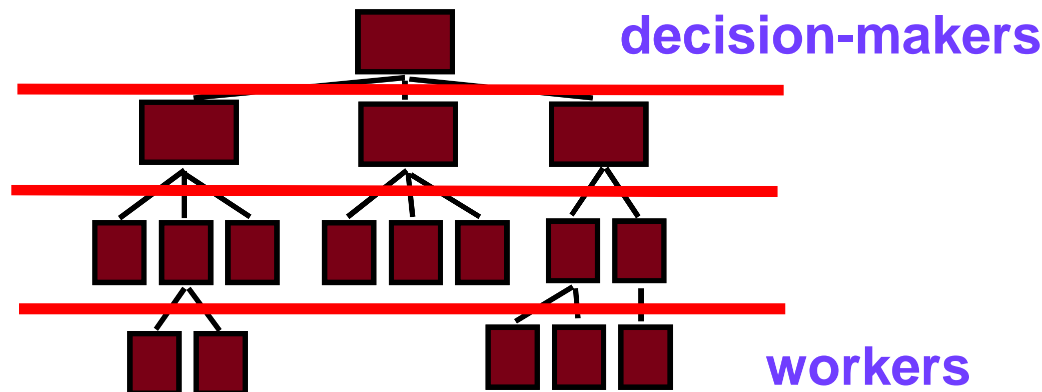
Horizontal Partitioning

- define separate branches of the module hierarchy for each major function
- use control modules to coordinate communication between functions



Vertical Partitioning: Factoring

- design so that decision making and work are stratified
- decision making modules should reside at the top of the architecture



Why Partitioned Architecture?

- results in software that is easier to test
- leads to software that is easier to maintain
- results in propagation of fewer side effects
- results in software that is easier to extend

Structured Design

- **objective:** develop a modular program structure and represent control relationships between modules
- **approach:**
  - the DFD is mapped into a program architecture
  - the PSPEC and STD are used to indicate the content of each module
- **notation:** structure chart

Structured Design

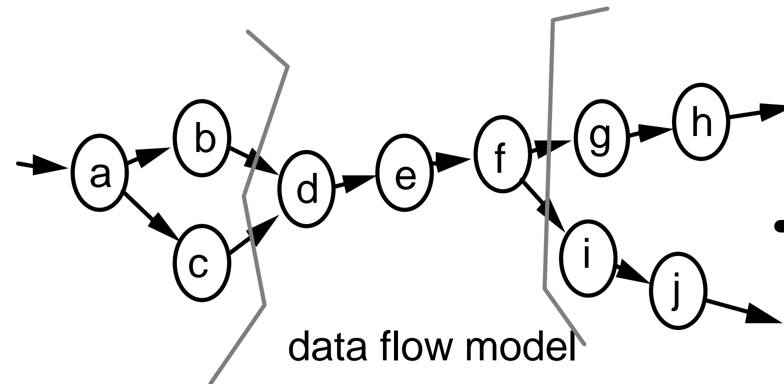
- It provides a convenient transition from a data flow diagram to software Architecture

1. **The type of information flow is established**
2. **Flow boundaries are indicated**
3. **The DFD is mapped into program structure**
4. **Control hierarchy is defined**
5. **Resultant structure is refined using design measures and heuristics**
6. **The architectural description is refined and elaborated**

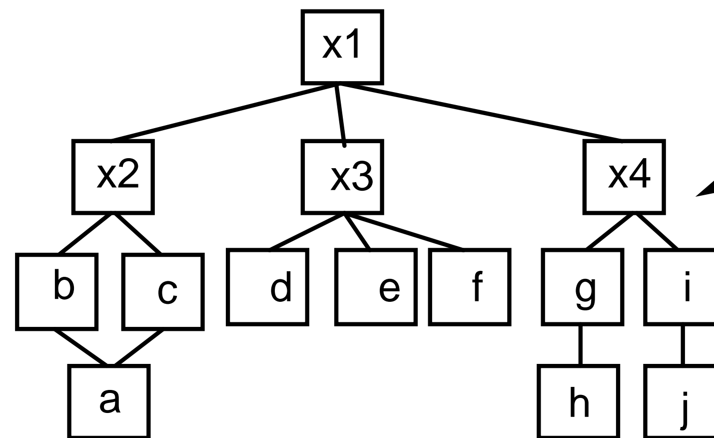
Types of Information Flow

- There are 2 different types of information flow that have different treatments
  - *Transform Flow* - Overall data flows in sequential manner and follows one, or only a few, "straight line" paths. (incoming, transform, output)
  - *Transaction Flow* - Info flow has a single transaction node that *triggers* other data flow

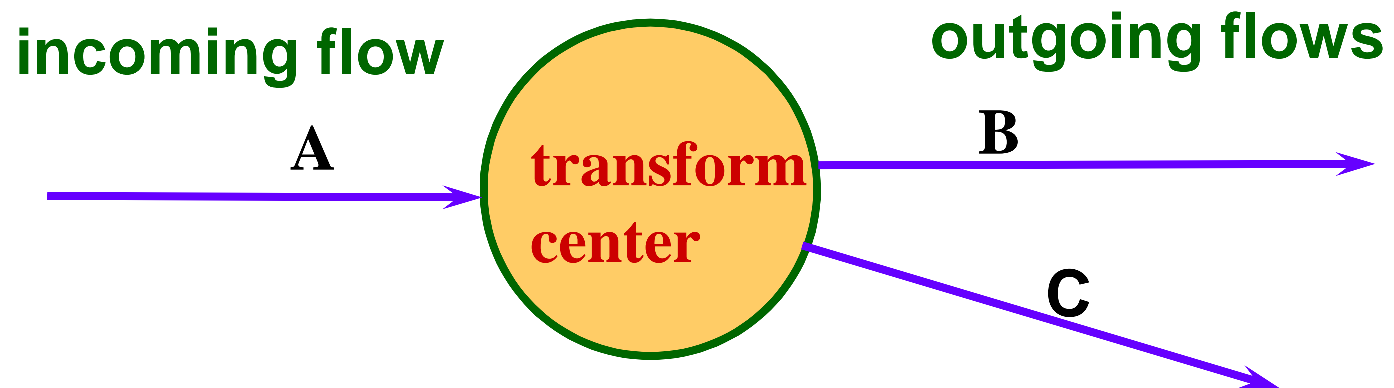
Transform Mapping



"Transform" mappin



Transform Flow



Transform Flow

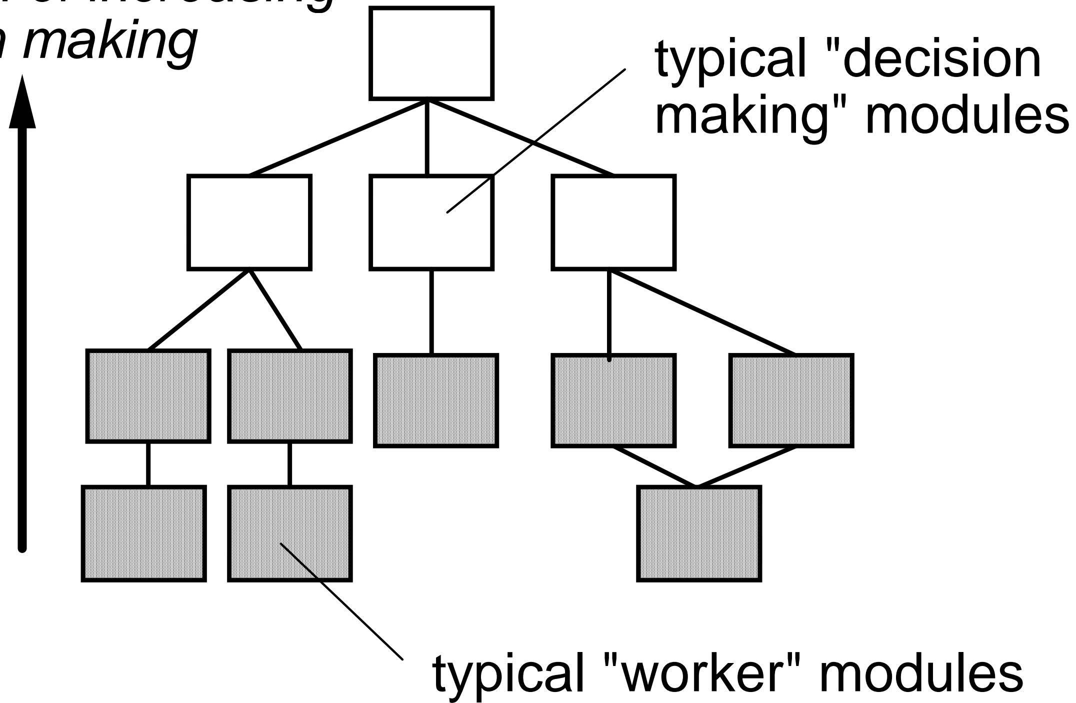
- ***Incoming Flow:*** The paths that transform the external data into an internal form
- ***Transform Center:*** The incoming data are passed through a transform center and begin to move along paths that lead it out of the software
- ***Outgoing Flow:*** The paths that move the data out of the software

Transform flow

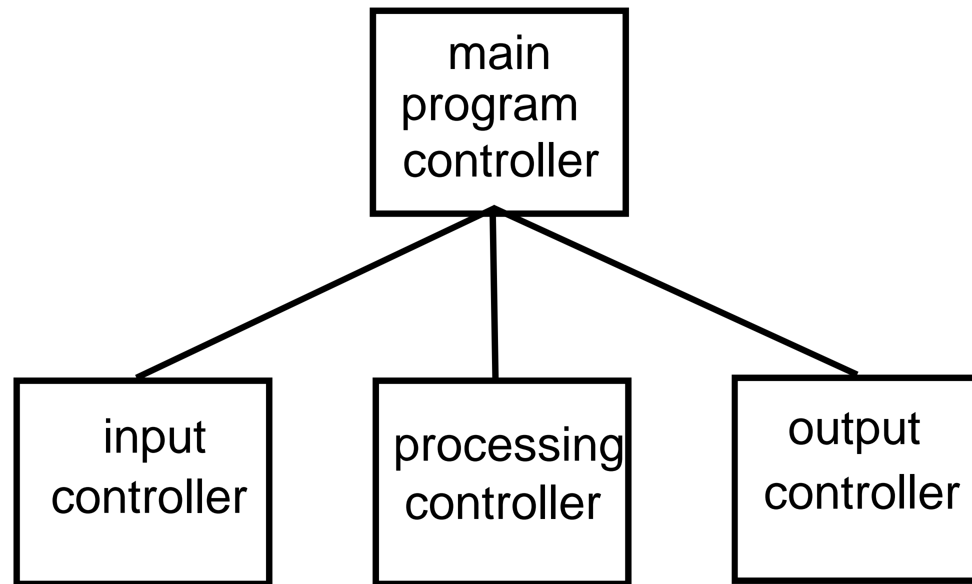
Transform Flow Characteristics

- the system has a single, coherent objective
- transformation center executes algorithms, data transformation, database manipulation, ...
- input-driven processes filter, check and translate external data flows
- output-driven processes format results for presentation to the environment (user)
- multiple paths to obtain input

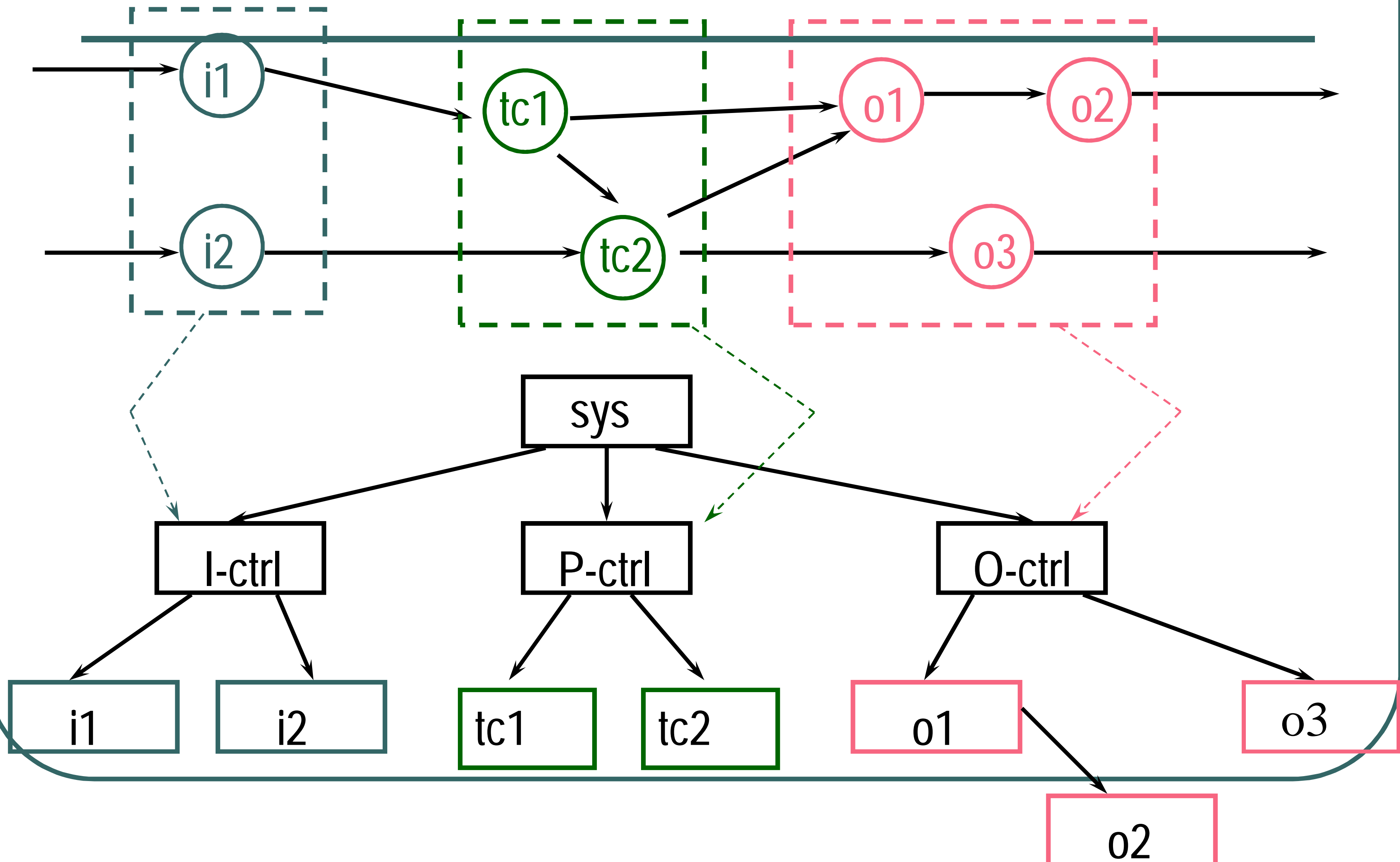
Factoring



First Level Factoring



Transform Analysis: mapping heuristic



Transaction Flow

Transaction

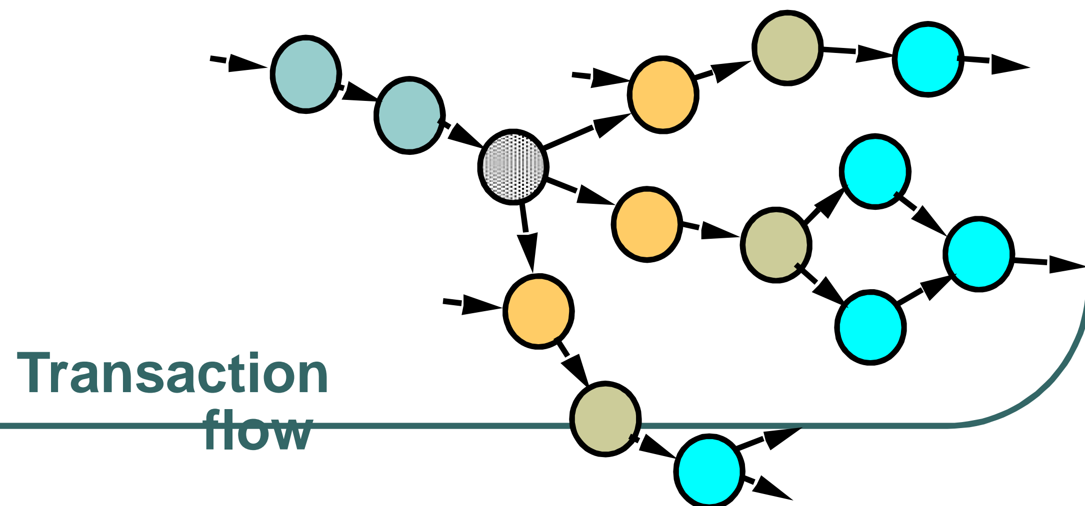
T

Action paths

Transaction center

Transaction Flow

- Information flow is often characterized by a single data item, called a **transaction** that triggers other data flow along one of many paths
- ***Action Paths :*** The transaction is evaluated and based on its value flow along one of many action paths
- ***Transaction center :*** The hub of info flow from which many action paths originate



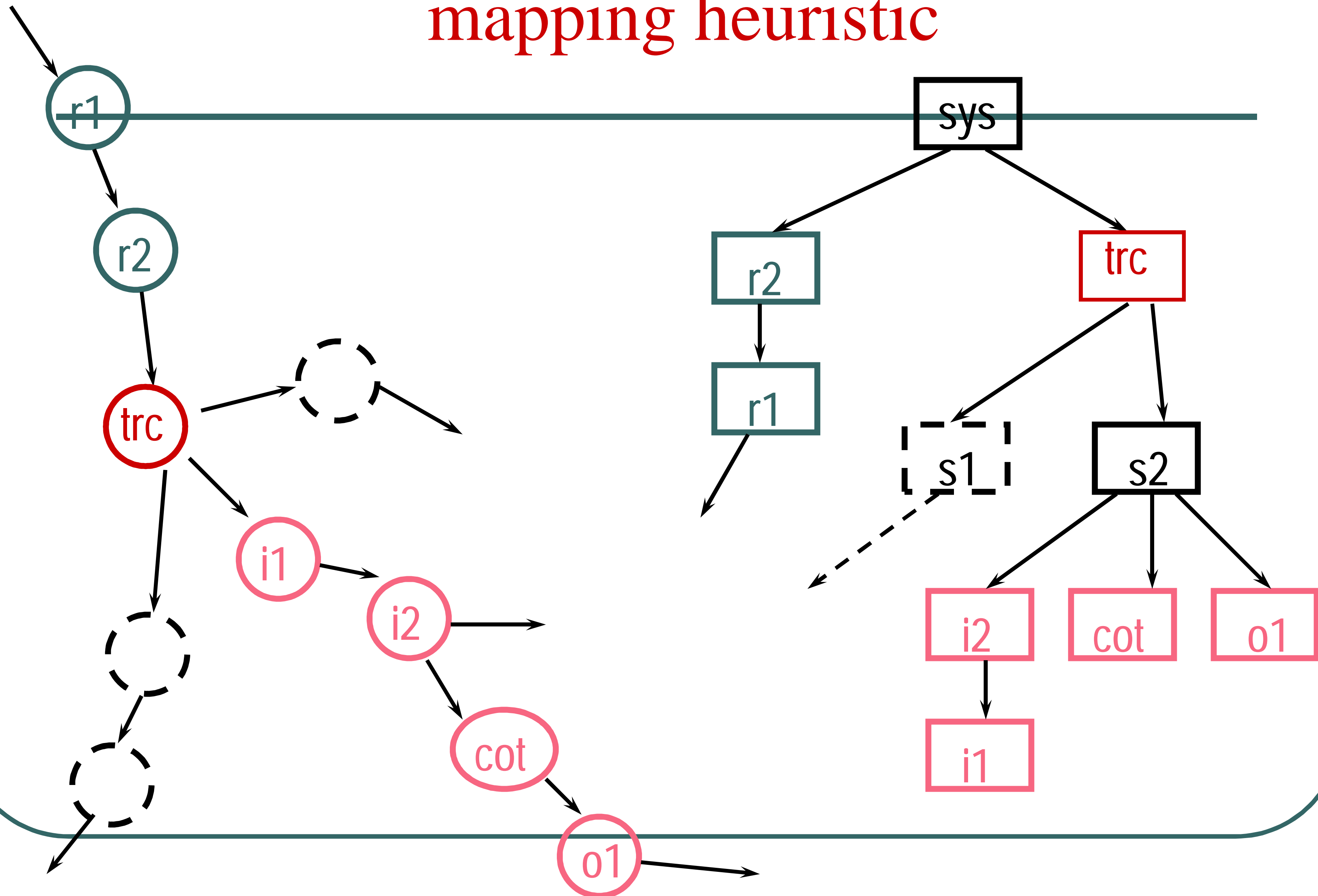
Transaction flow

Transaction Flow Characteristics

- single line of reception processes
- transaction: a single data item that includes all necessary information for execution
- transaction center evaluates transaction & initializes correct action-path => distribution
- action-paths implement (clearly) different types of functionality => execution
- an action-path could be a complete

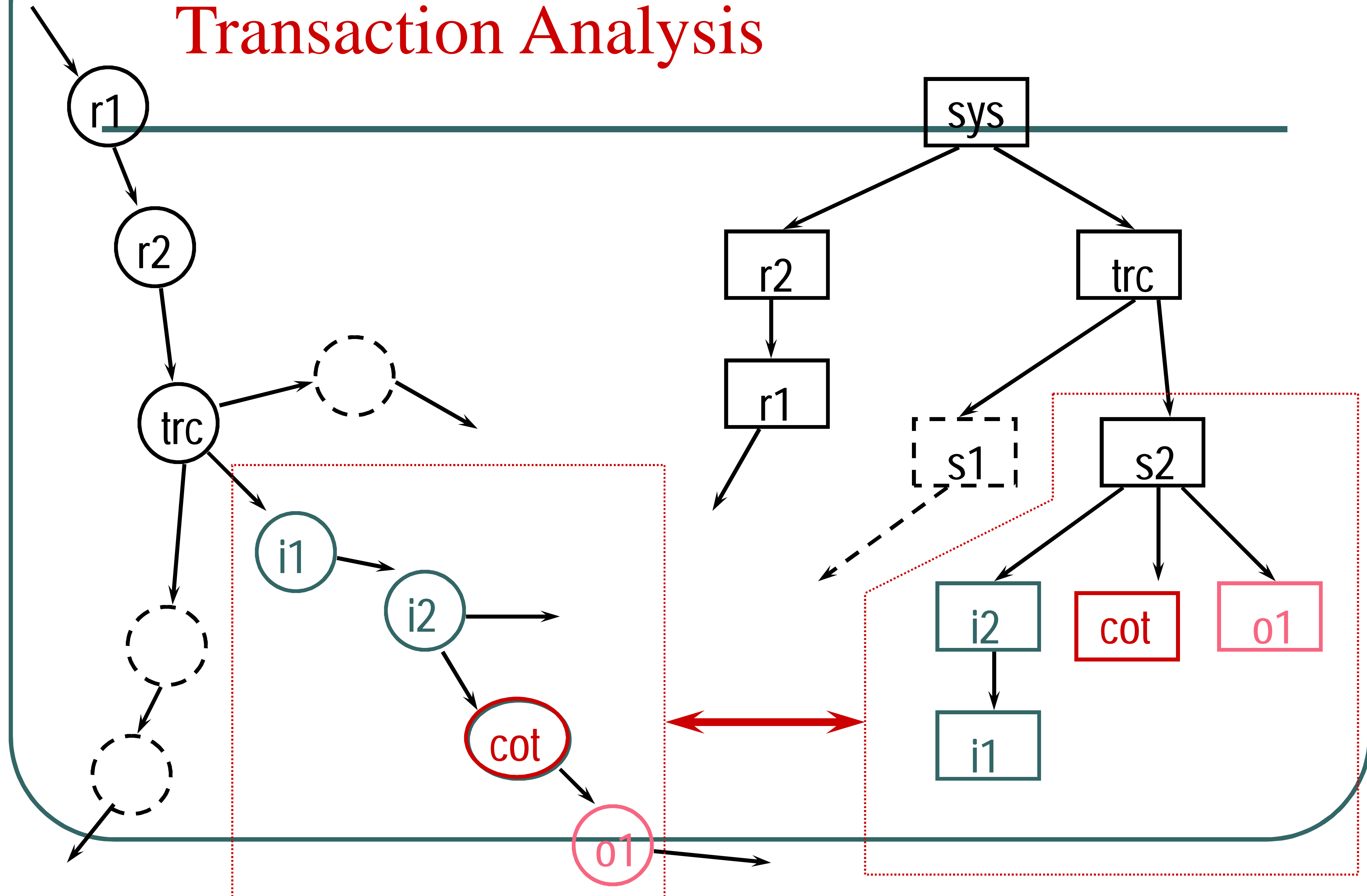
  (sub-)system with transform flow characteristics

Transaction Analysis: mapping heuristic



Transaction Analysis may include Transform Analysis as an element.

Transform Analysis as an element of Transaction Analysis

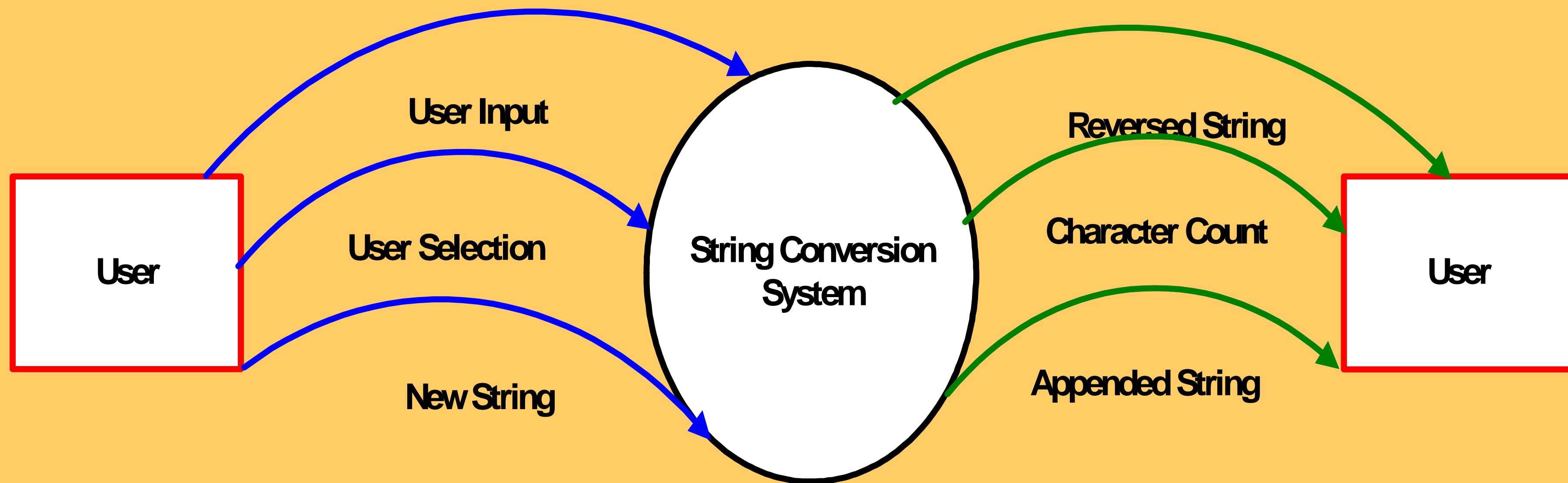


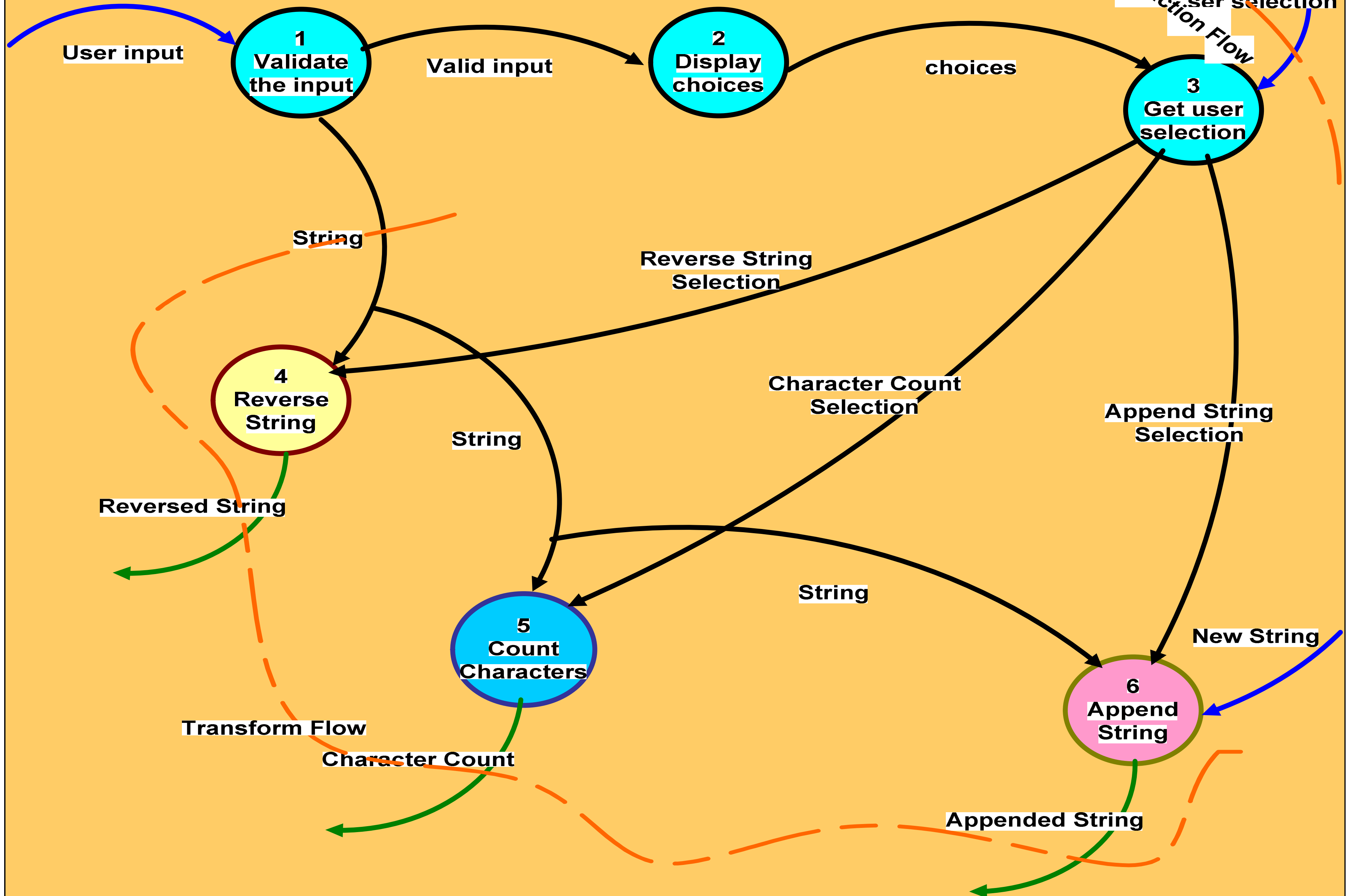
Transform Mapping (cont)

- **Design steps**
  - **Step 1. Review the fundamental system model.**
  - **Step 2. Review and refine data flow diagrams for the software.**
  - **Step 3. Determine whether DFD has transform or transaction flow characteristics.**

- in general---transform flow
- special case---transaction flow

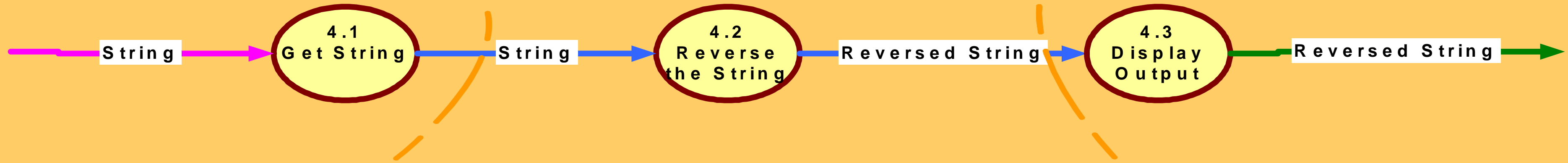
Context Level DFD



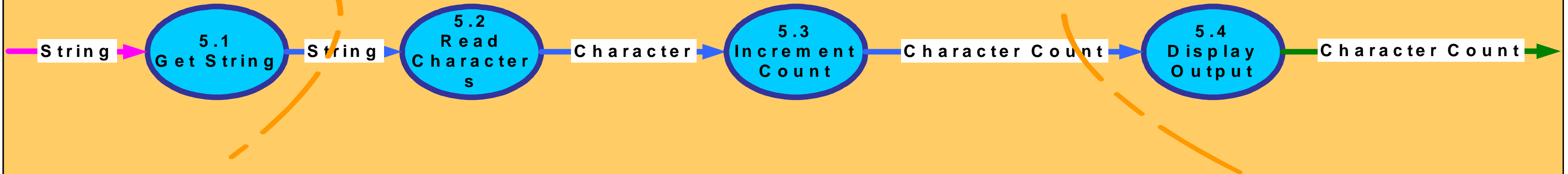


Level 1 DFD
User input
1
Validate
the input
Valid input
2
Display
choices
choices
Transaction Flow
ser selection
3
Get user
selection
String
Reverse String
Selection
4
Reverse
String
Character Count
Selection
Append String
Selection
String
Reversed String
String
New String
5
Count
Characters
6
Append
String
Transform Flow
Character Count
Appended String

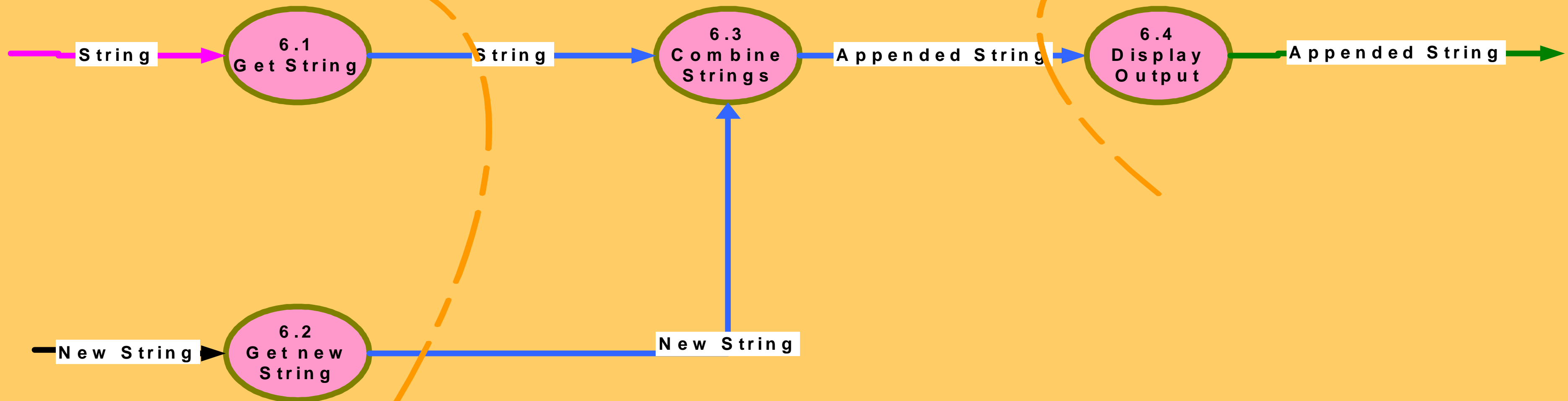
DFD Level-2 For REVERSE STRING - <Process # 4>



DFD Level-2 For Count Characters - <Process # 5>



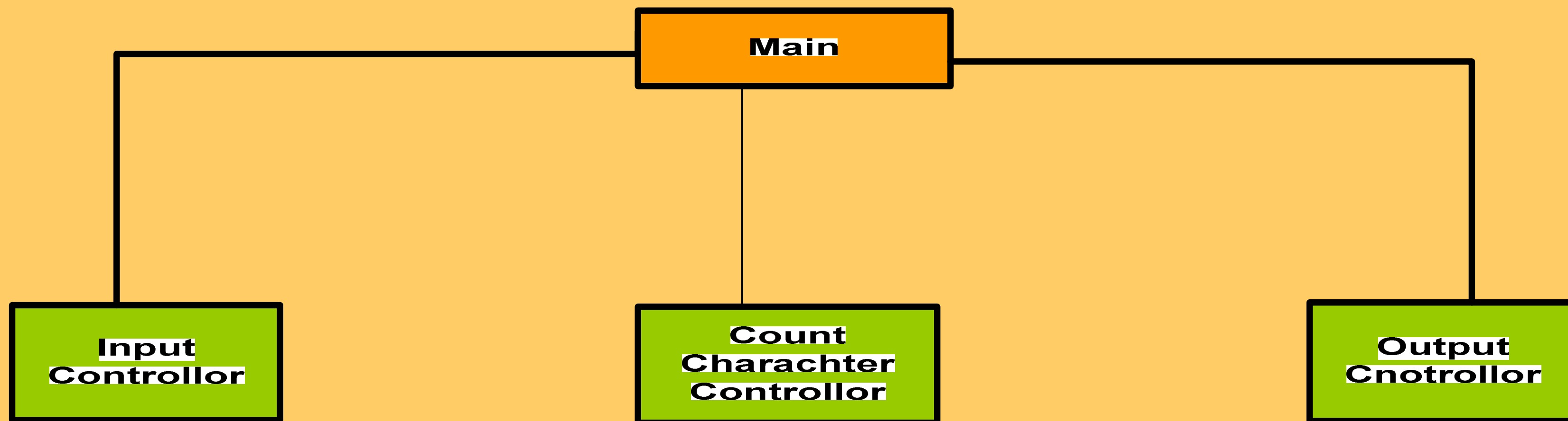
DFD Level-2 For Append STRING - <Process # 6>



Transform Mapping (cont)

- **Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries**
  - different designers may select slightly differently
  - transform center can contain more than one bubble.
- **Step 5. Perform "first-level factoring"**
  - program structure represent a top-down distribution control.
  - factoring results in a program structure(top-level, middle-level, low-level)
  - number of modules limited to minimum.

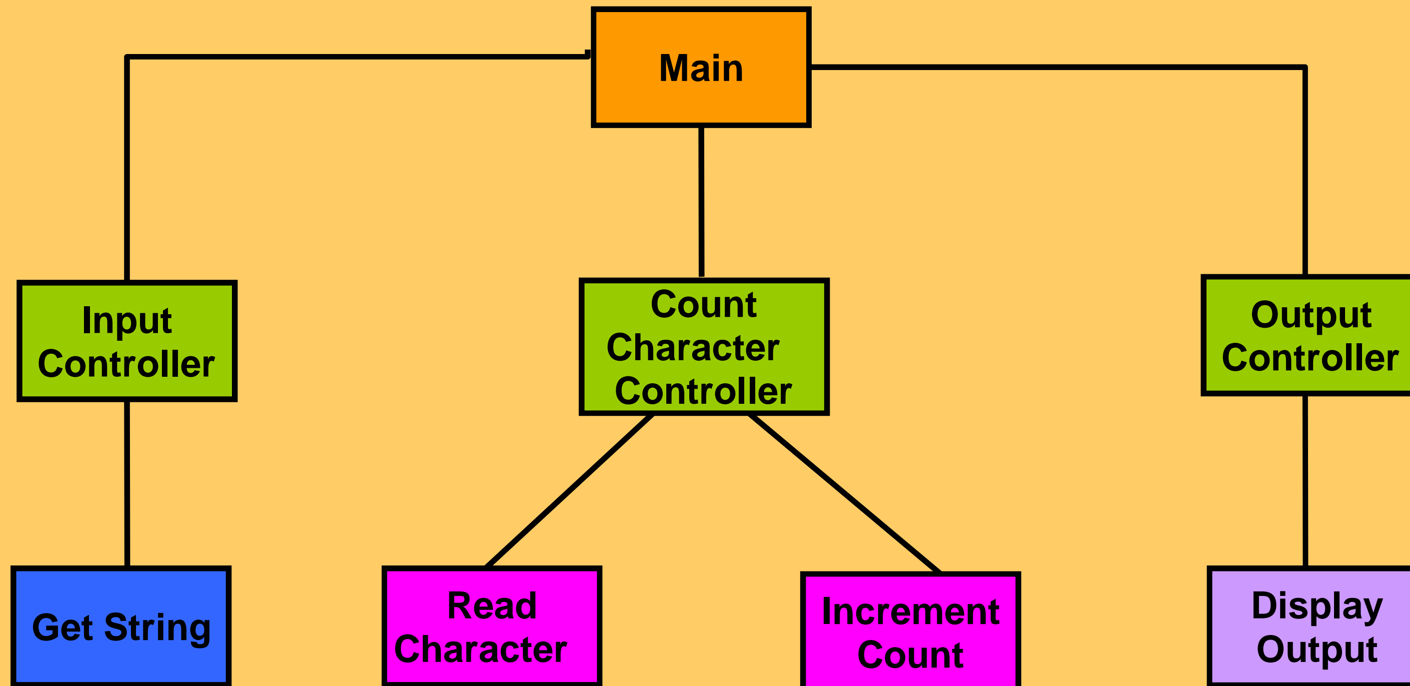
First Level Factoring



Transform Mapping (cont)

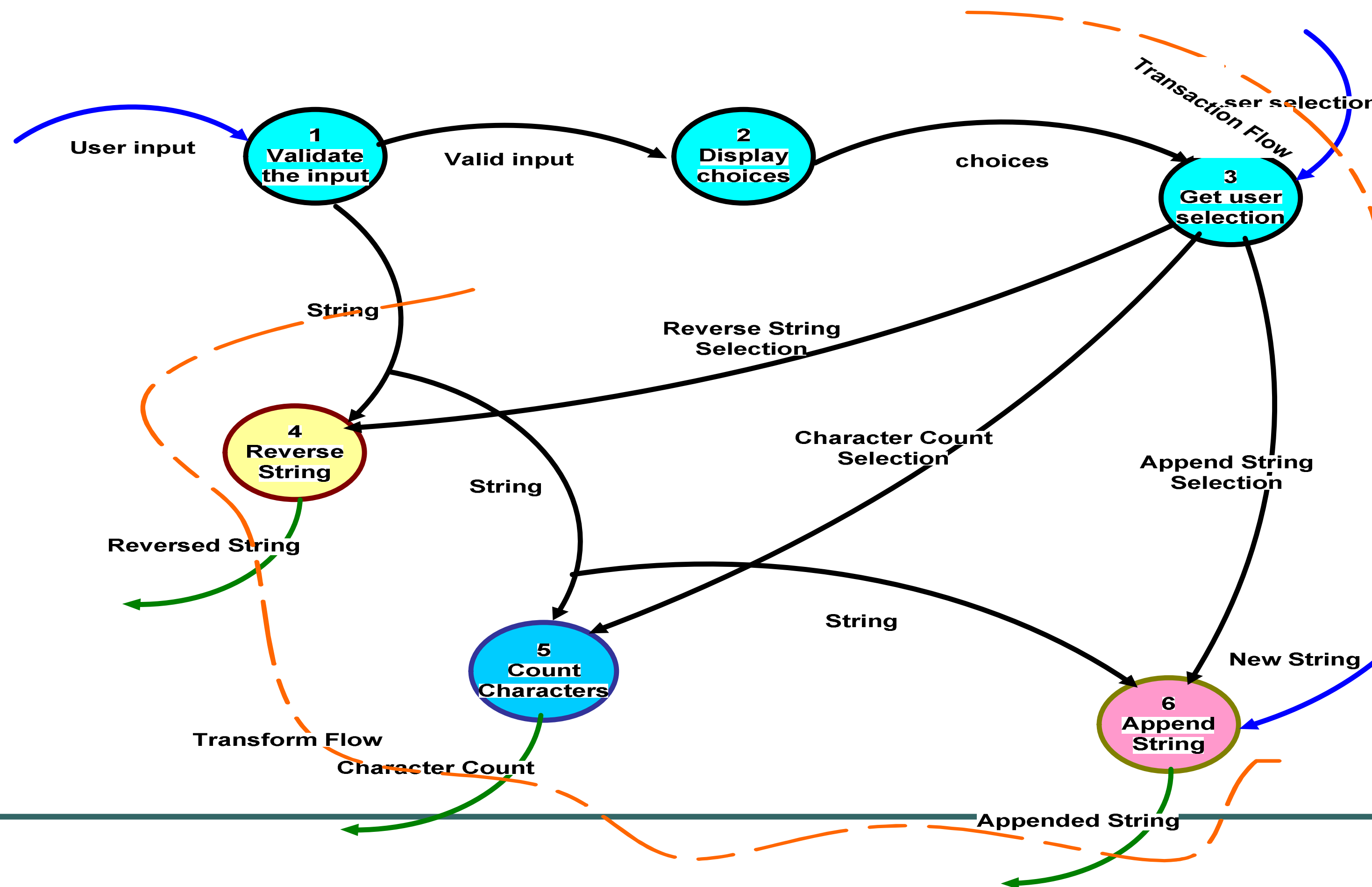
- **Step 6. Perform "second-level factoring"**
  - mapping individual transforms(bubbles) to appropriate modules.
  - factoring accomplished by moving outwards from transform center boundary.
- **Step 7. Refine the first iteration program structure using design heuristics for improved software quality.**

Second Level Factoring



Transaction Mapping

A single data item triggers one or more information flows



Transaction Mapping Design

- **Step 1.Review the fundamental system model.**
- **Step 2.Review and refine DFD for the software**
- **Step 3.Determine whether the DFD has transform or transaction flow characteristics**
- **Step 4. Identify the transaction center and flow characteristics along each of the action paths**
  - isolate incoming path and all action paths
  - each action path evaluated for its flow characteristic.

Transaction Mapping (cont)

- **Step 5. Map the DFD in a program structure amenable to transaction processing**
  - *incoming branch*
    - bubbles along this path map to modules
  - *dispatch branch*
    - dispatcher module controls all subordinate action modules
    - each action path mapped to corresponding structure

Transaction Mapping

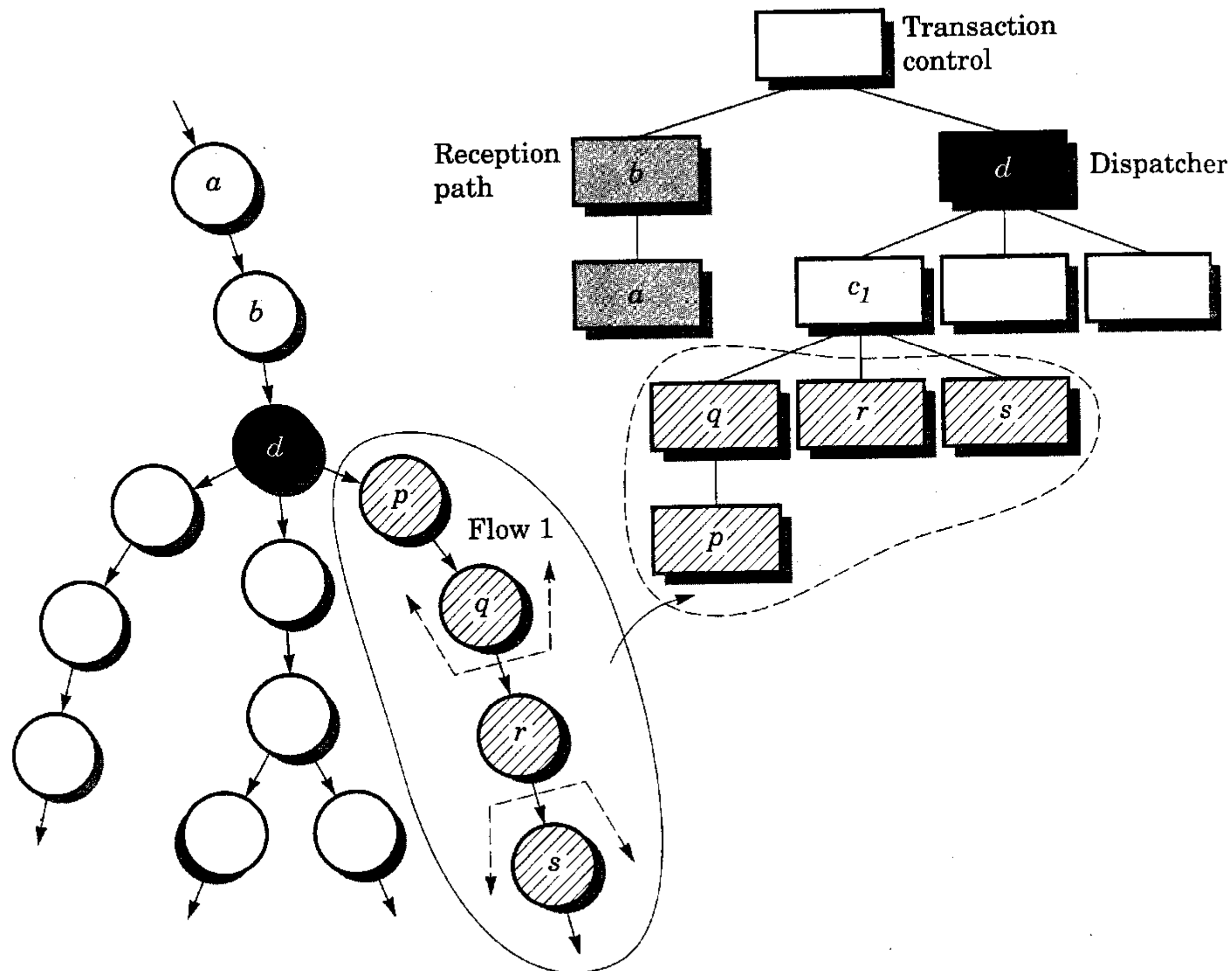
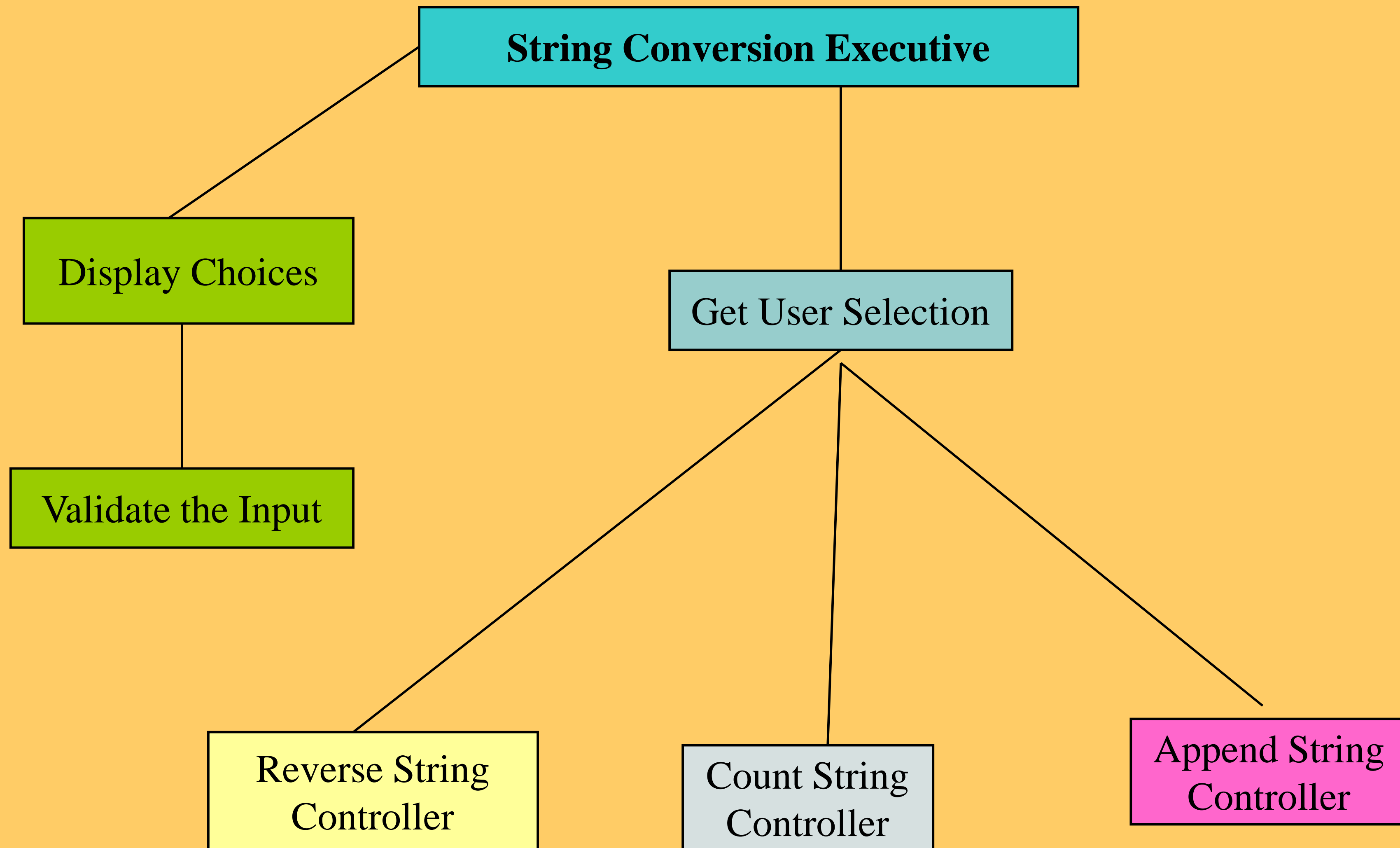
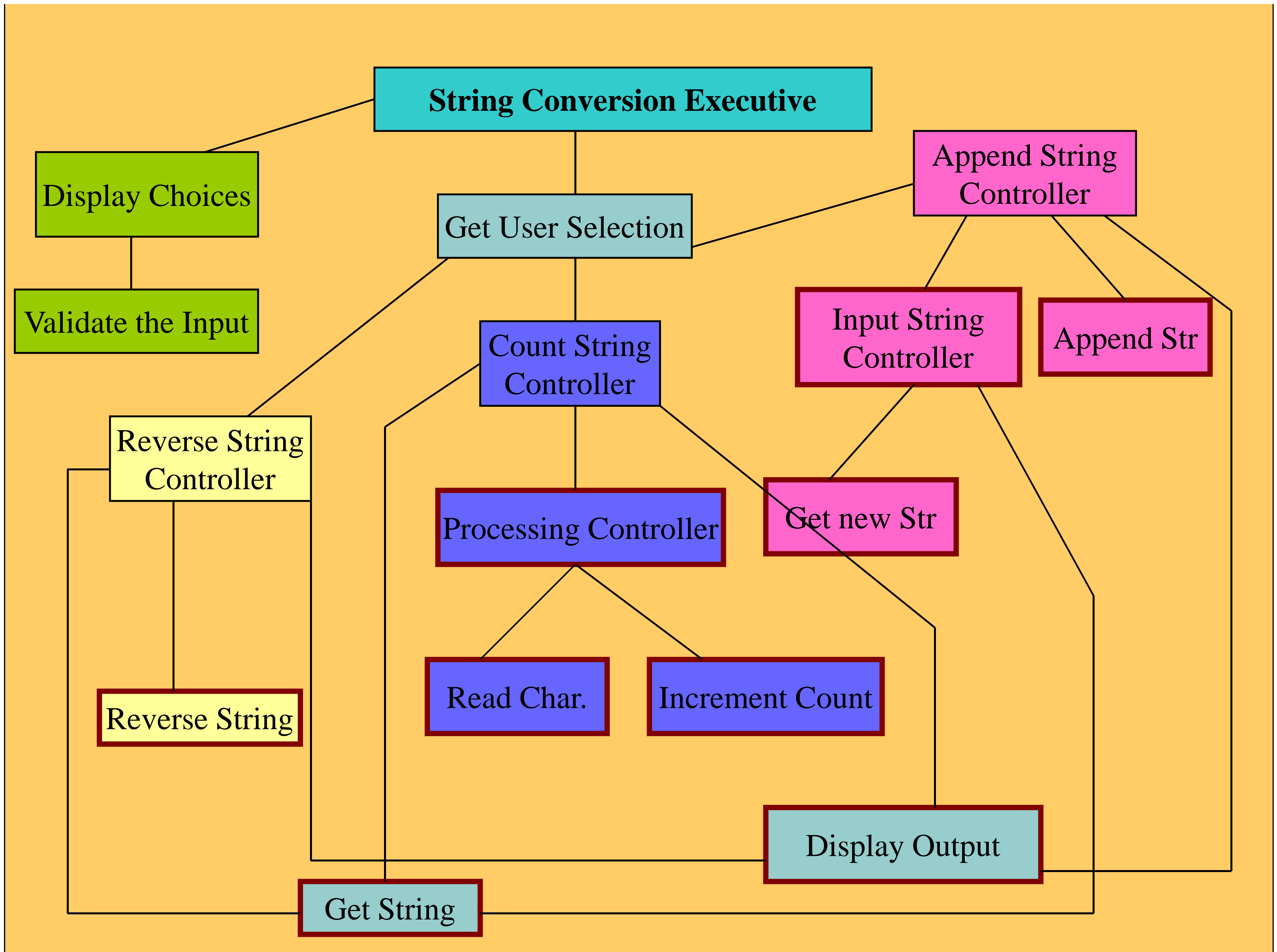


FIGURE 14.12. Transaction mapping

First Level Factoring




String Conversion Executive
Display Choices
Get User Selection
Append String Controller
Validate the Input
Input String Controller
Append Str
Count String Controller
Reverse String Controller
Processing Controller
Get new Str
Read Char.
Increment Count
Reverse String
Display Output
Get String

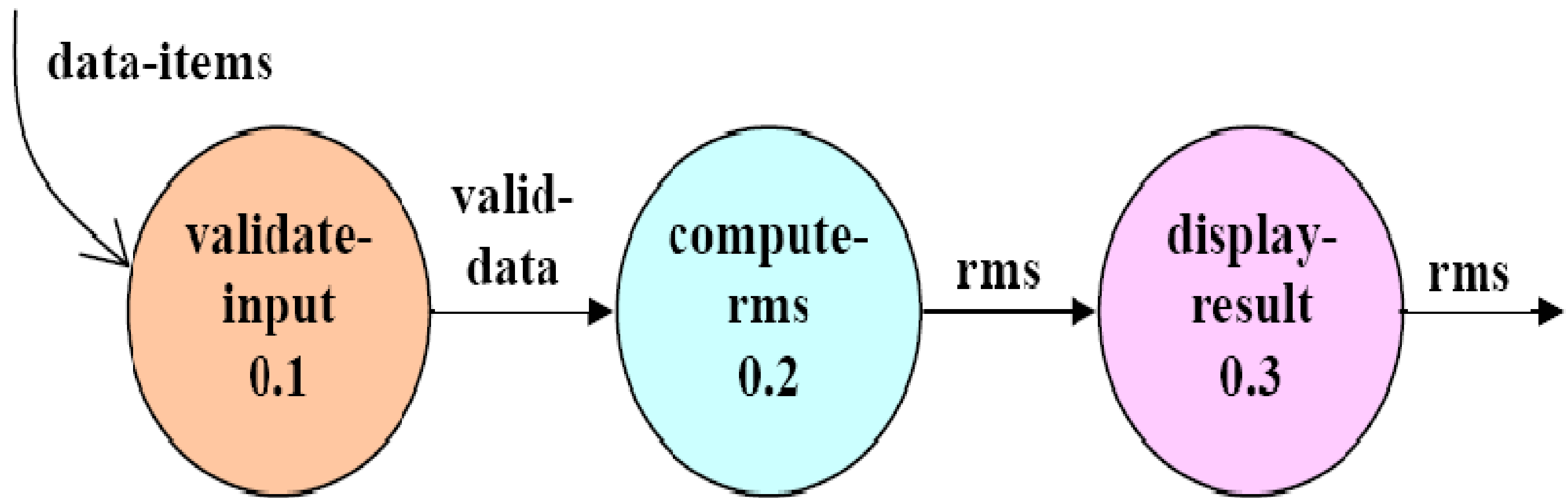
Transaction Mapping (cont)

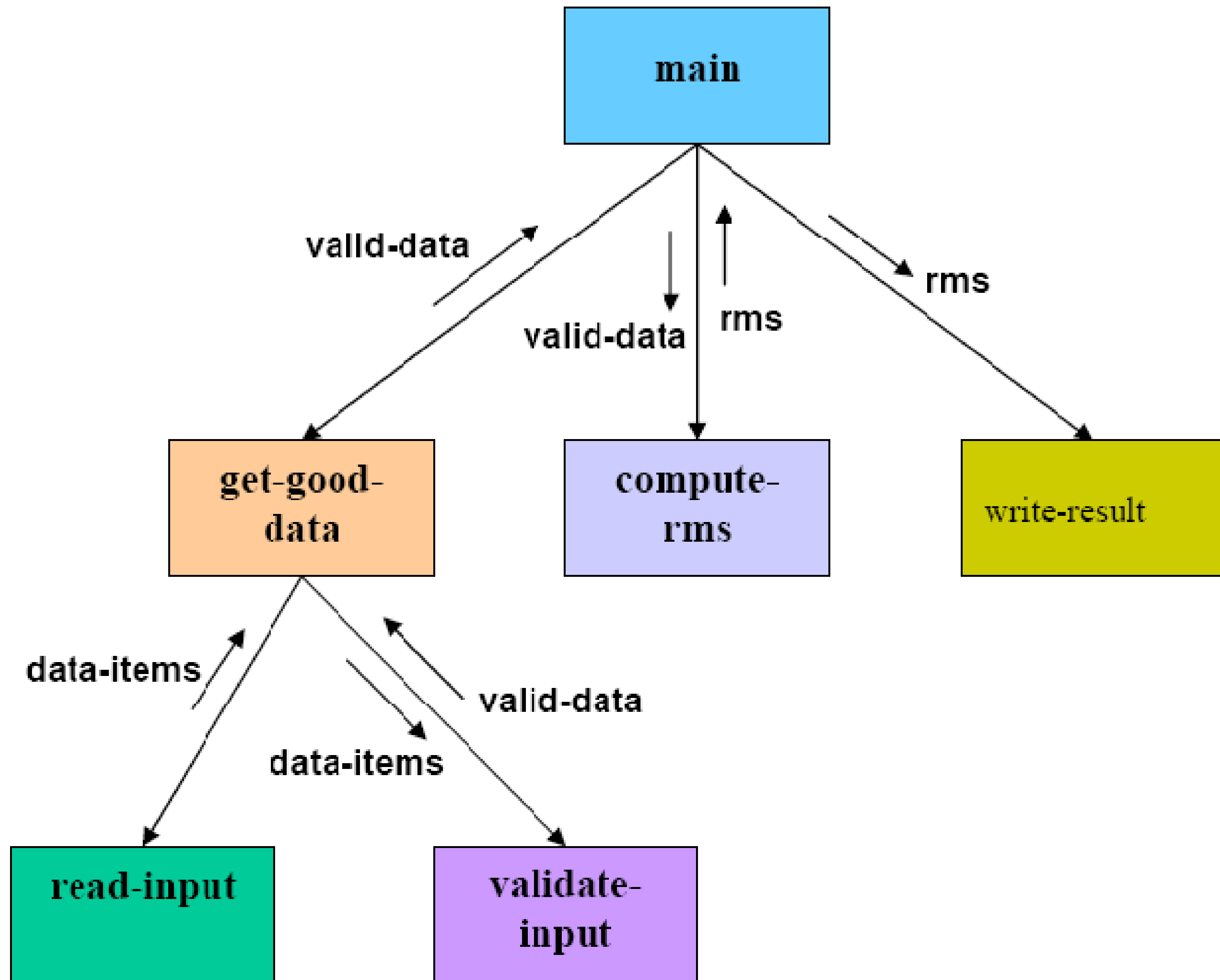
- **Step 6. Factor and refine the transaction structure and the structure of each action path**

- **Step 7. Refine the first iteration program structure using design heuristics for improved software quality**

Design Post processing

- A processing narrative must be developed for each module
- An interface description is provided for each module
- Local and global data structures are defined
- All design restrictions/limitations are noted
- A design review is conducted
- “Optimization” is considered (if required and justified)


data-items
validate-
input
0.1
valid-
data
compute-
rms
0.2
rms
display-
result
0.3
rms



main
valid-data
valid-data
rms
rms
get-good-data
compute-rms
write-result
data-items
valid-data
data-items
read-input
validate-input

Software Design

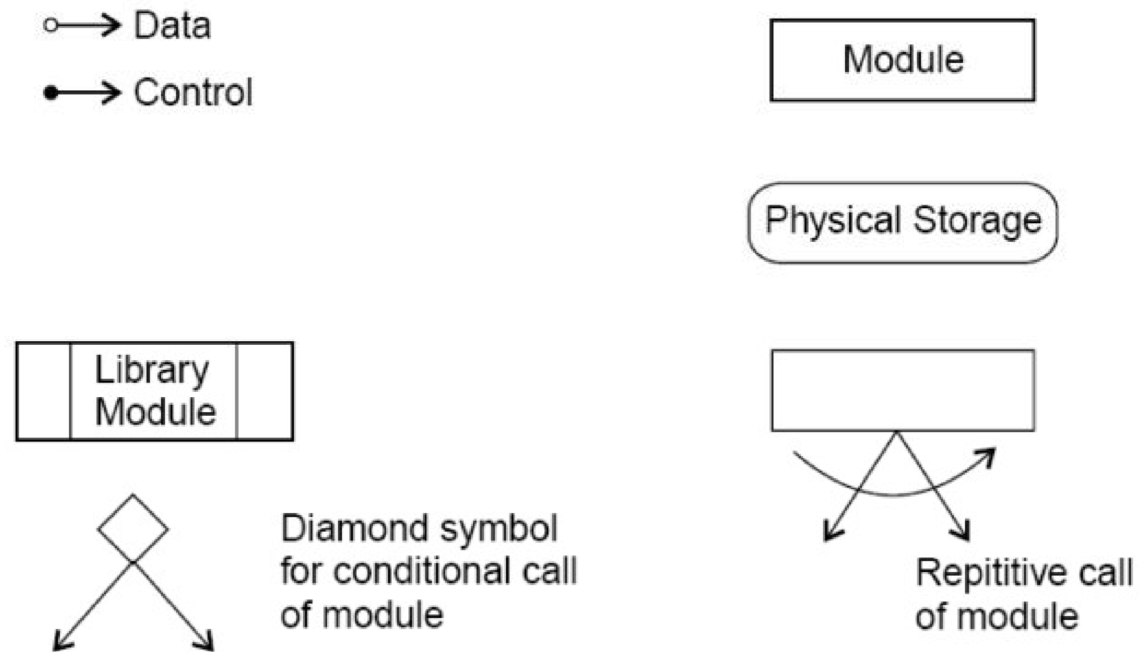


Fig. 17 : Structure chart notations

Software Design

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

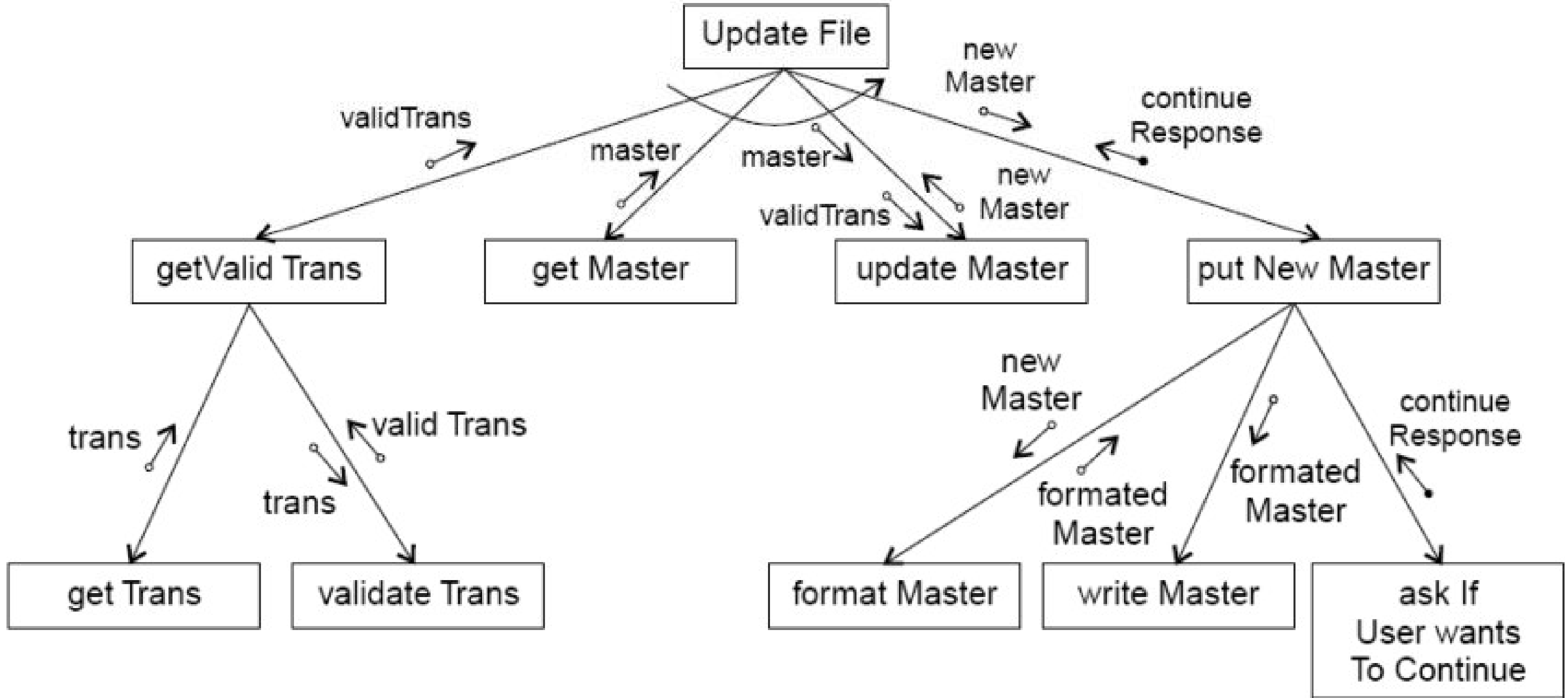


Fig. 18 : Update file

Software Design

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

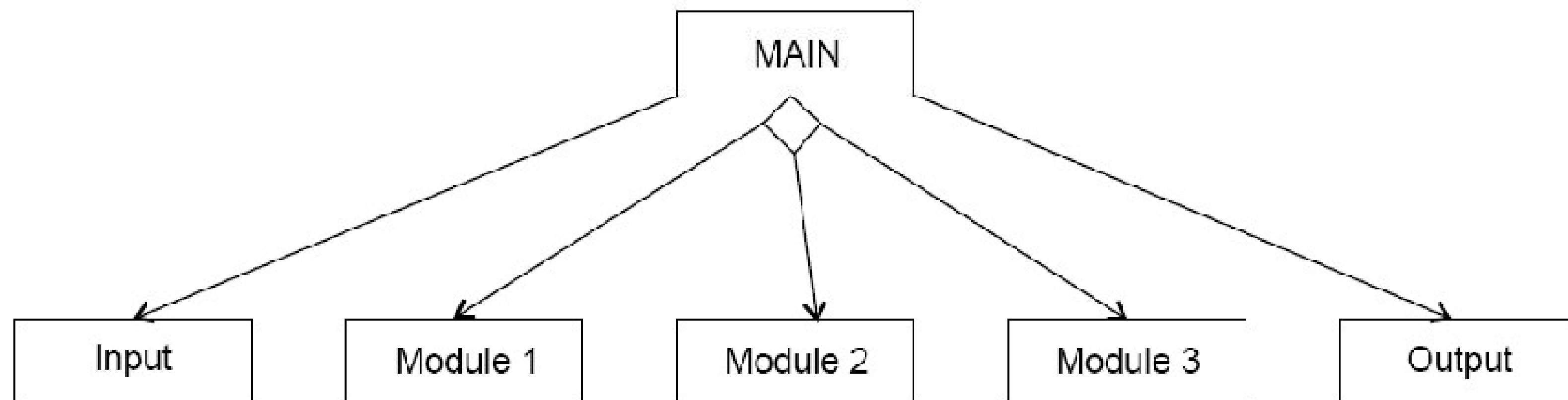


Fig. 19 : Transaction-centered structure

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

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

Software Design

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

Software Design

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.)

Software Design

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.

Software Design

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.

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.

Design Description Information Content

Introduction

Design entities

Design entity attributes

Software Design

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

Software Design

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.

Software Design

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 decompostion
     - 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…

Software Design

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

Software Design

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

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 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.

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. 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.

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

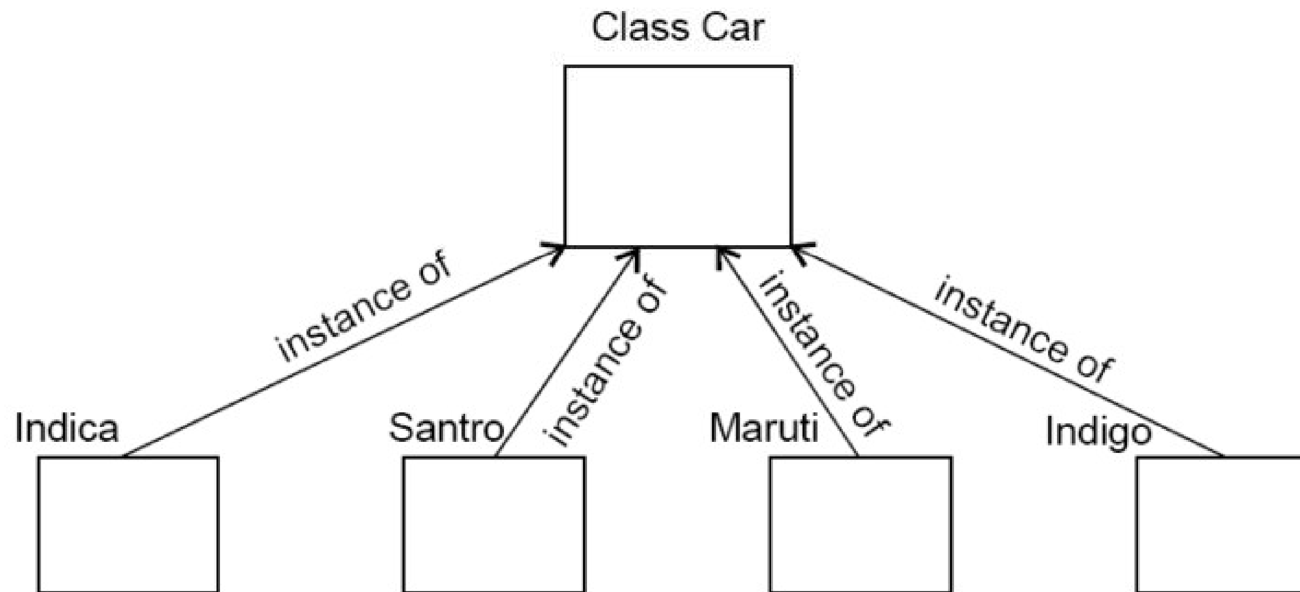


Fig.20: Indica, Santro, Maruti, Indigo are all instances of the class “car”

Software Design

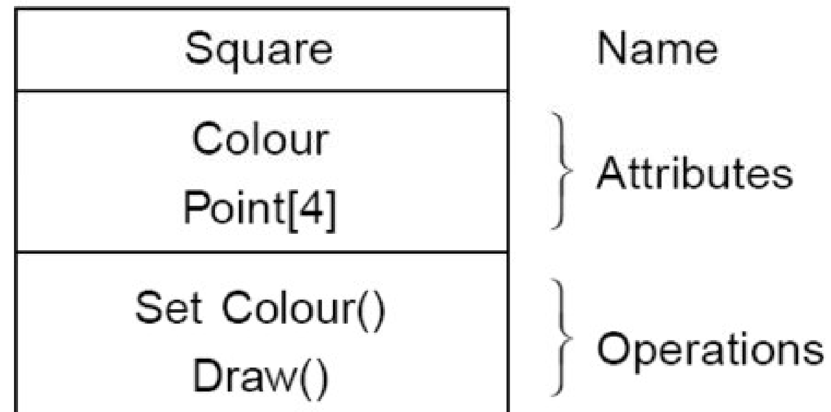


Fig. 21: The square class

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.

vii. Inheritance

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

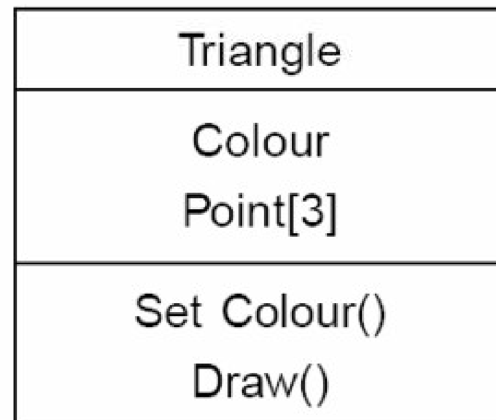


Fig. 22: The triangle class

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

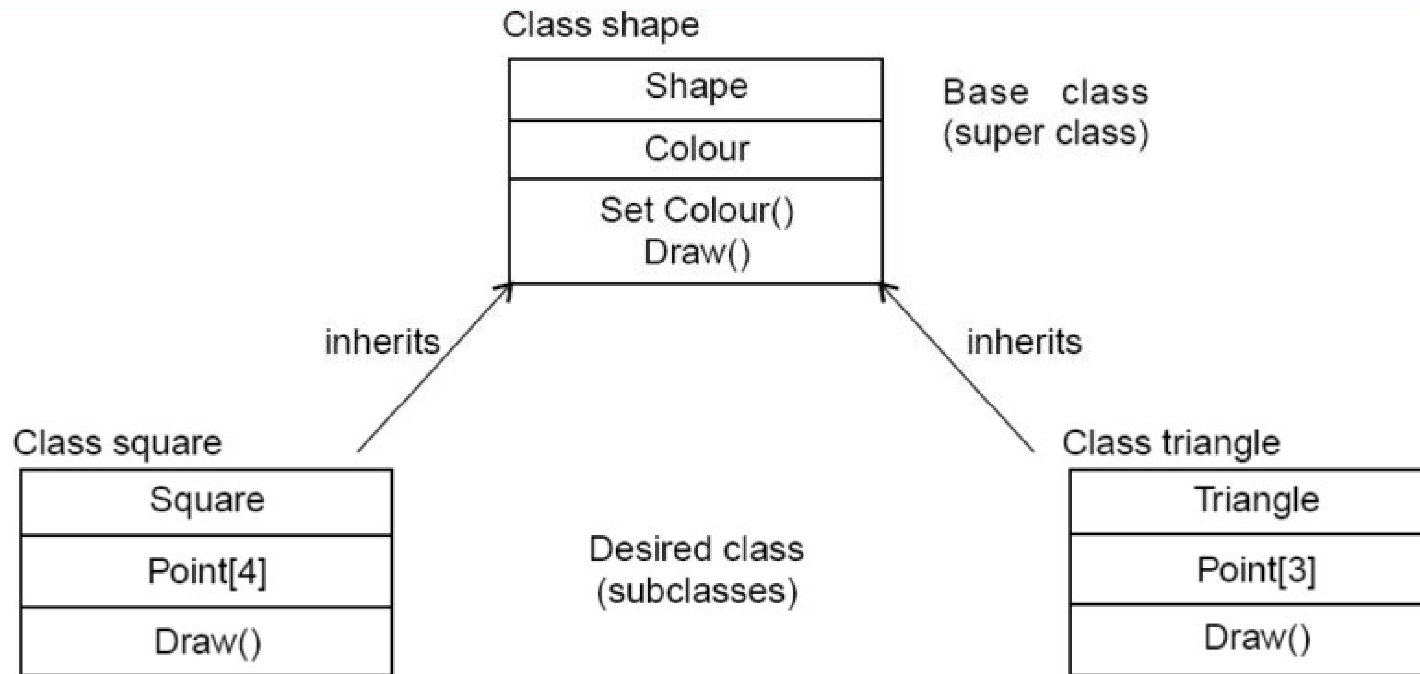


Fig. 23: Abstracting common features in a new class

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

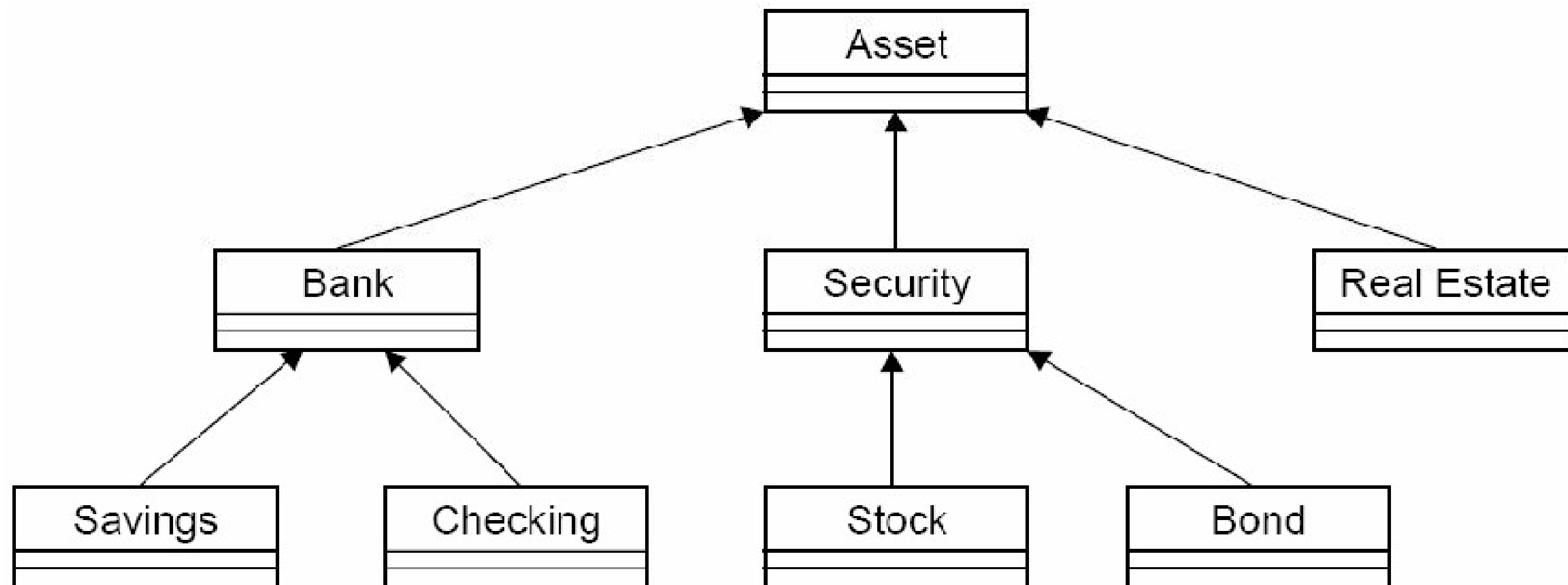


Fig. 24: Hierarchy

Software Design

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

Software Design

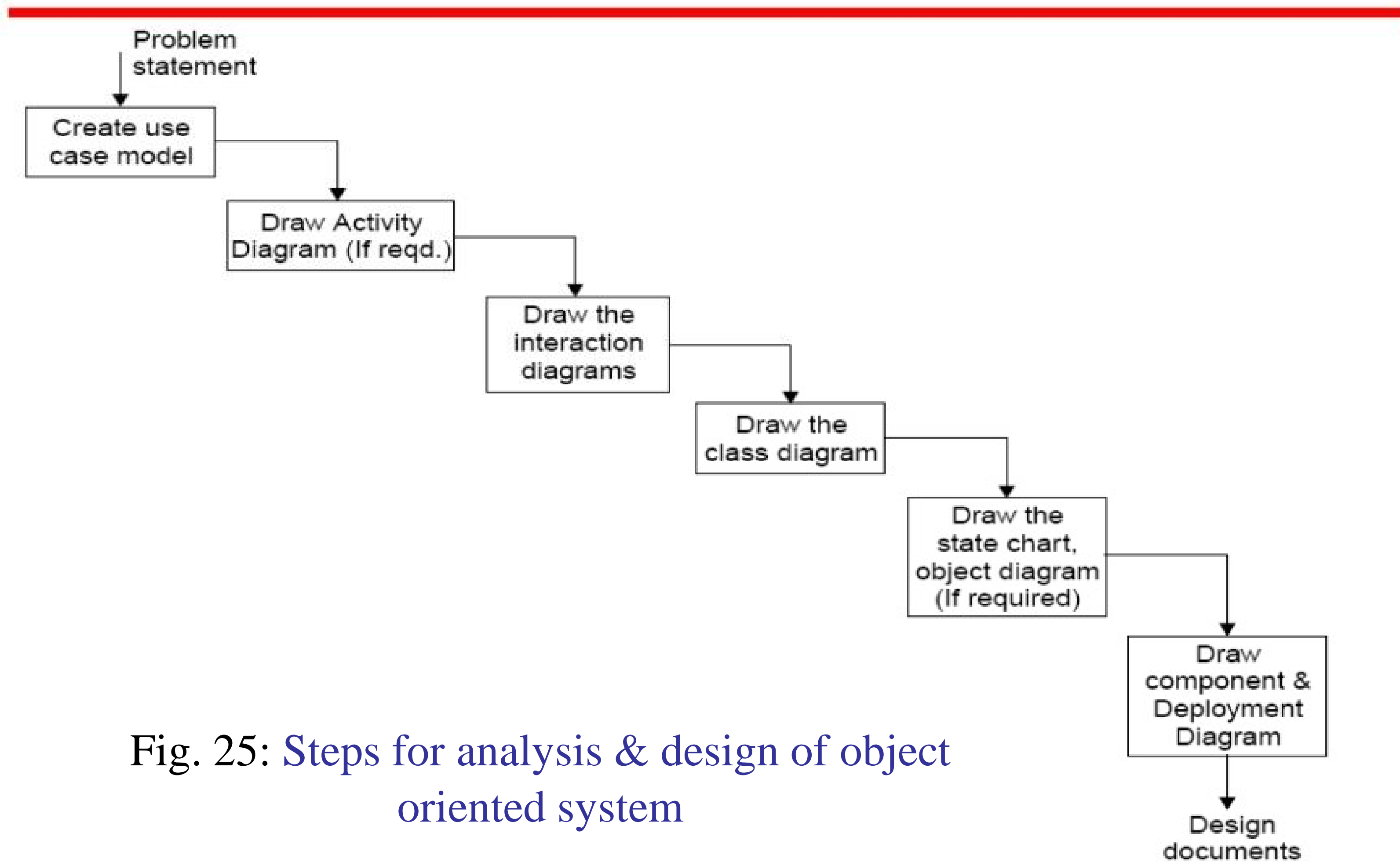


Fig. 25: Steps for analysis & design of object oriented system

Software Design

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 form 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.

Software Design

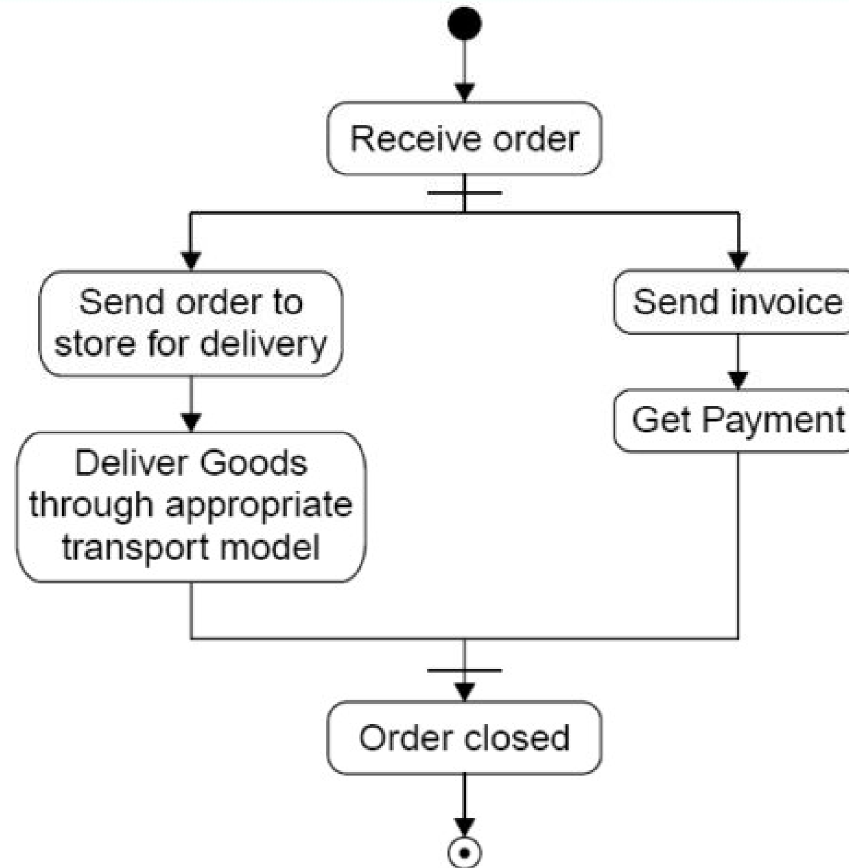


Fig. 26: Activity diagram

Software Design

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 draws interaction diagrams are as under:

a) Firstly, we should identify that the objects with respects to every use case.

b) We draw the sequence diagrams for every use case.

d) We draw the collaboration diagrams for every use case.

Software Design

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

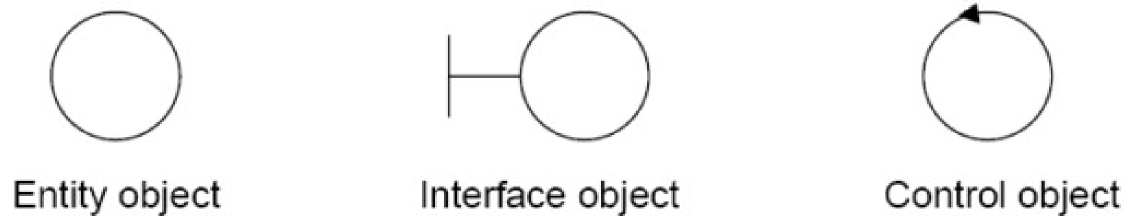


Fig. 27: Object types

Software Design

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.

Software Design

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

c) **Aggregations** are 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.

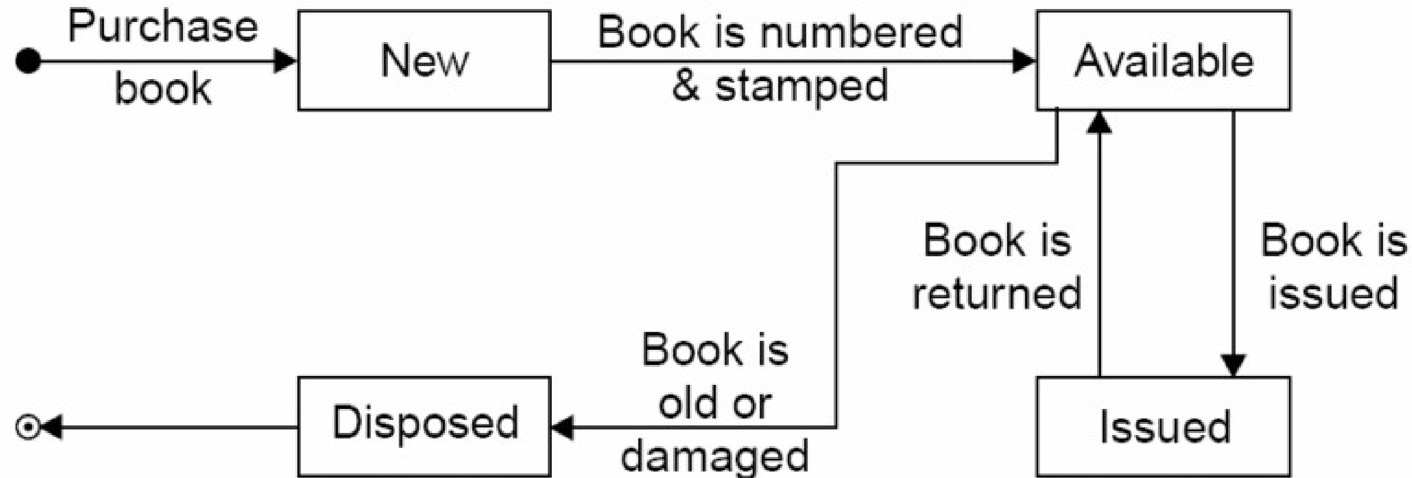


Fig. 28: Transition chart for "book" in a library system.

Software Design

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.

Software Design

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.

Software Design

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.

Software Design

Query Processing:

The system should be able to provide information like:

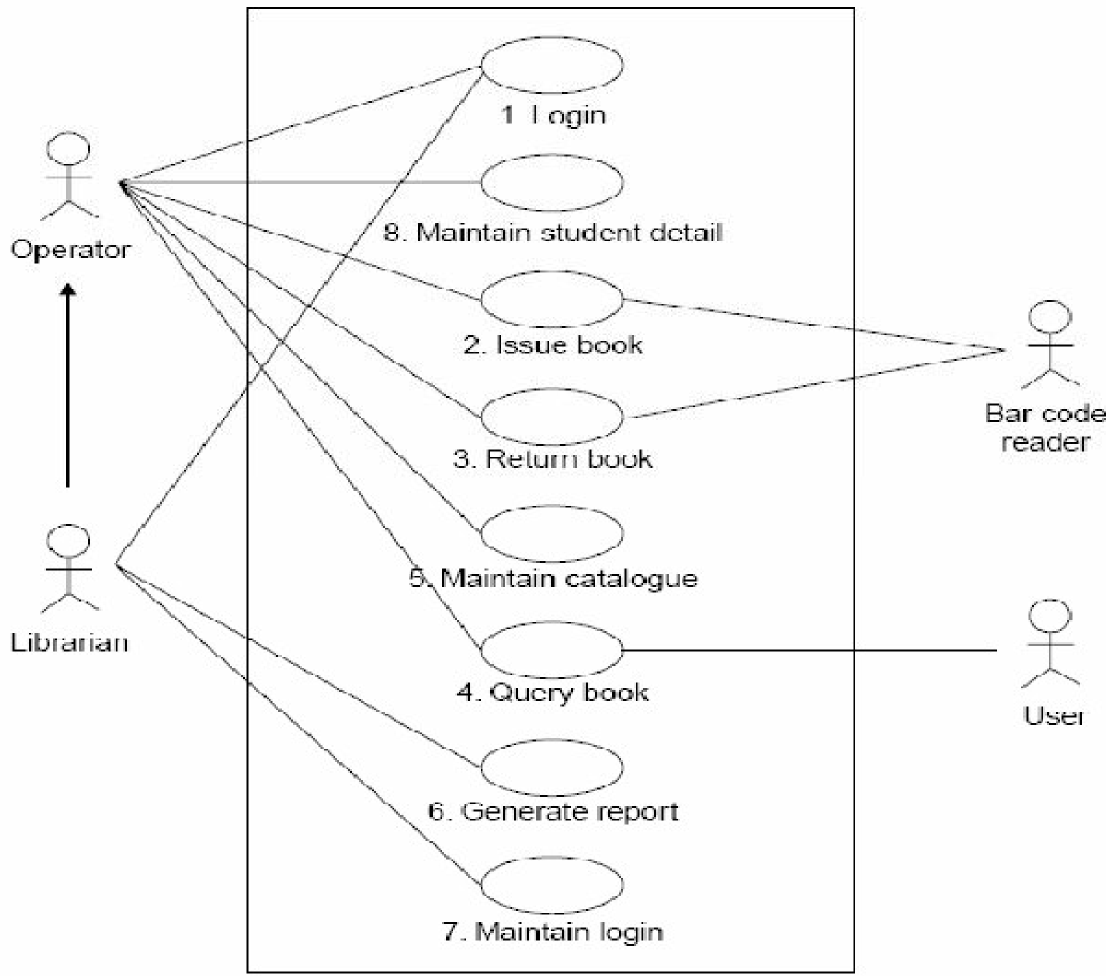
Availability of a particular book.

Availability of book of any particular author.

Number of copies available of the desired book.

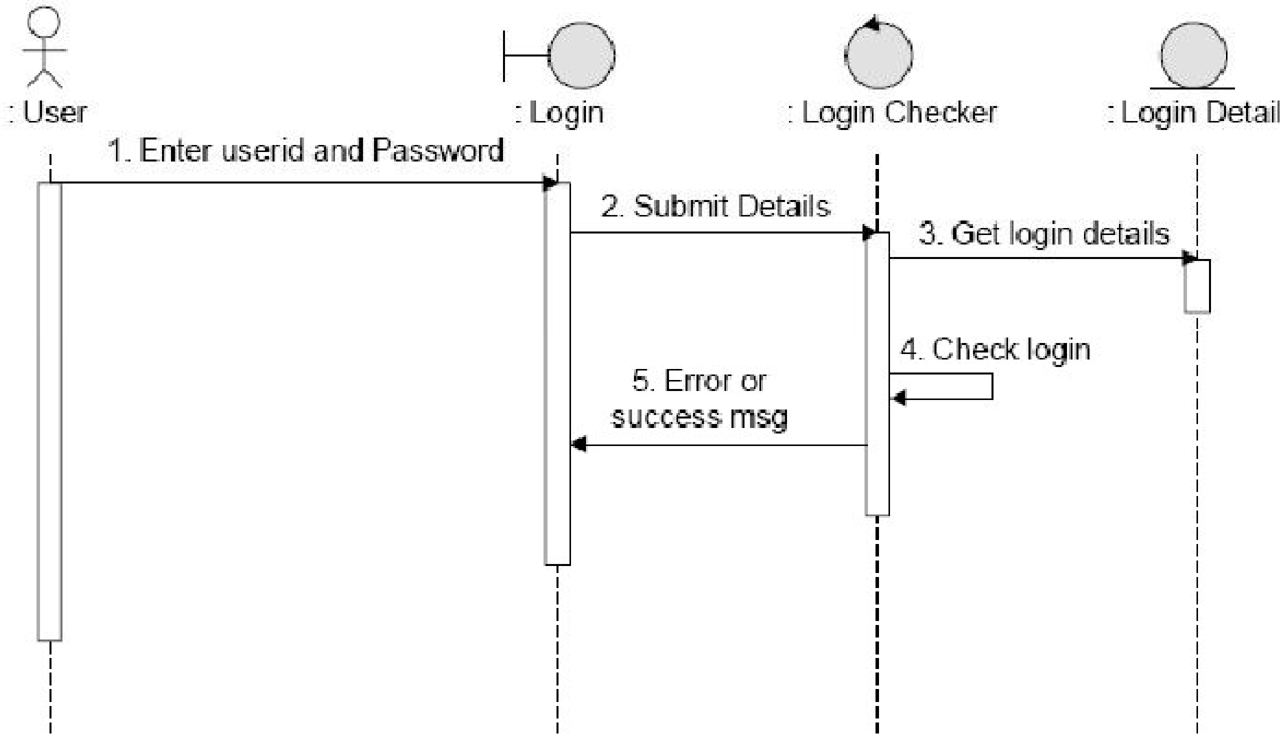
The system should also be able to generate reports regarding the details of the books available in the library at any given time. The corresponding printouts for each entry(issue/return) made in the system should be generated. Security provisions like the‘login authenticity should be provided. Each user 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.

Software Design



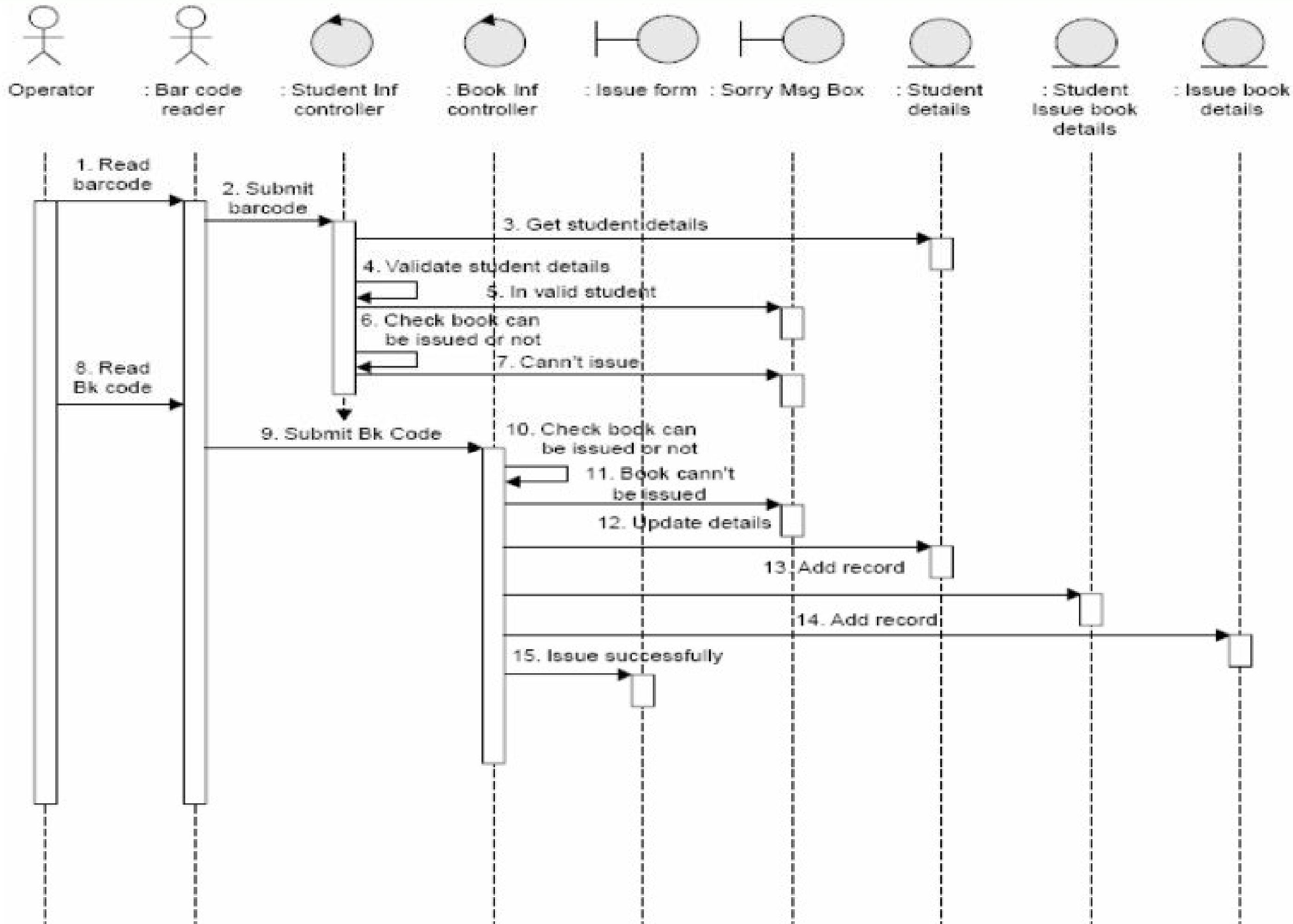
Use case diagram for library management system

Software Design



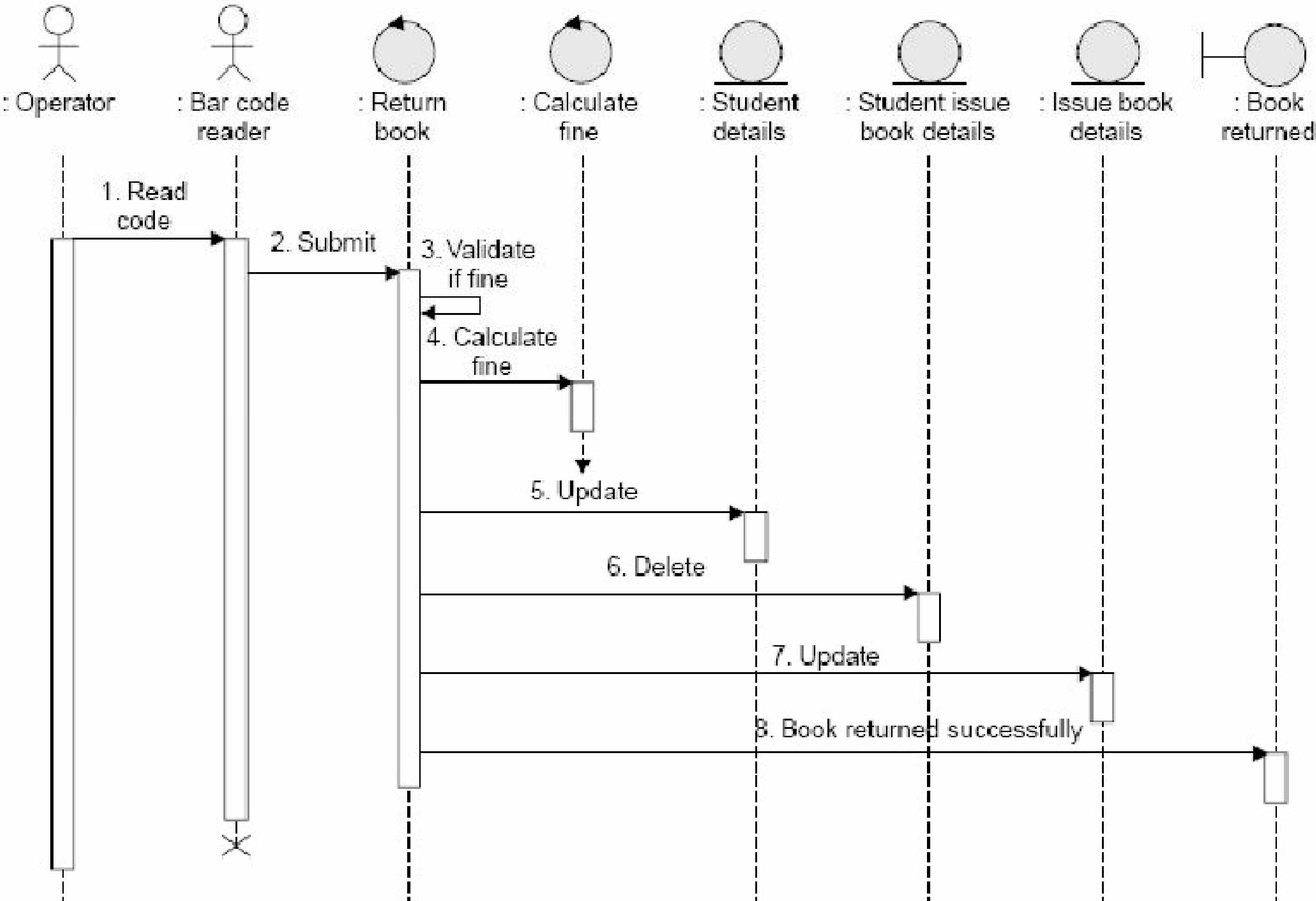
Sequence diagram—Login

Software Design



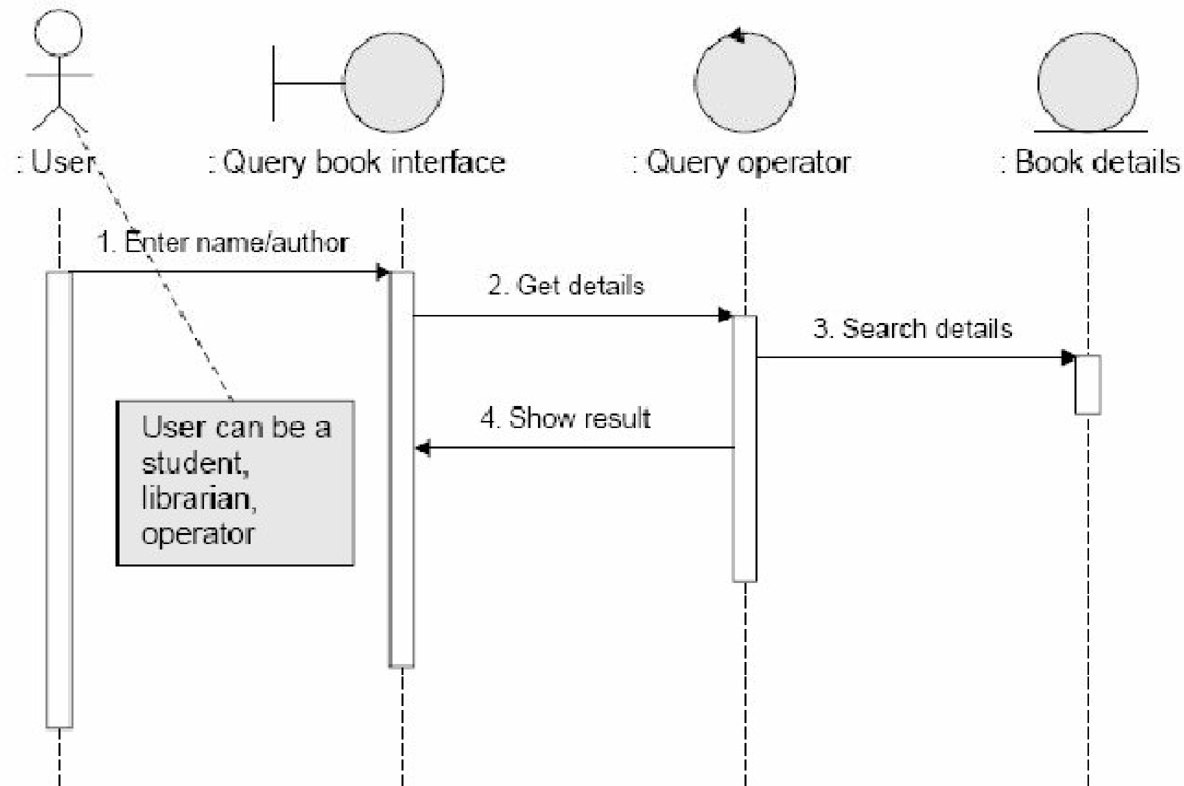
Sequence diagram—issue book

Software Design



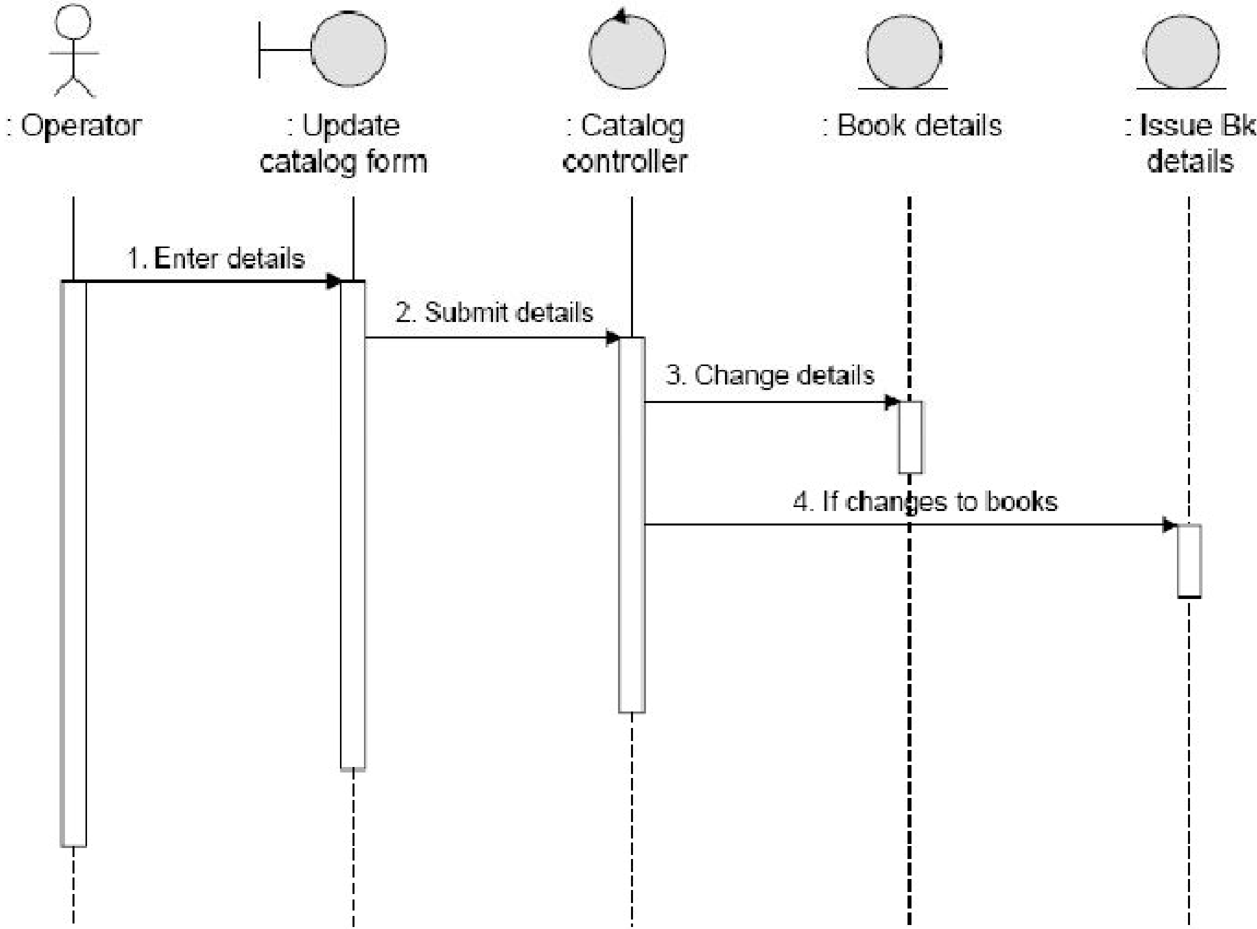
Sequence diagram—return book

Software Design



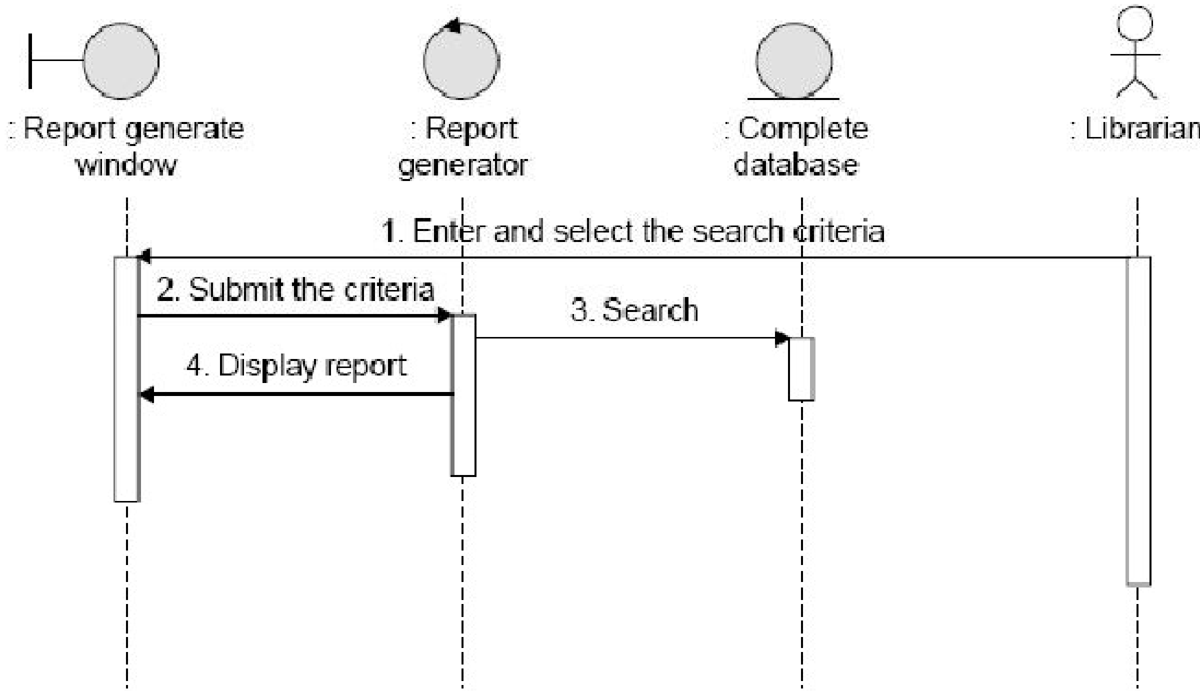
Sequence diagram—query book

Software Design



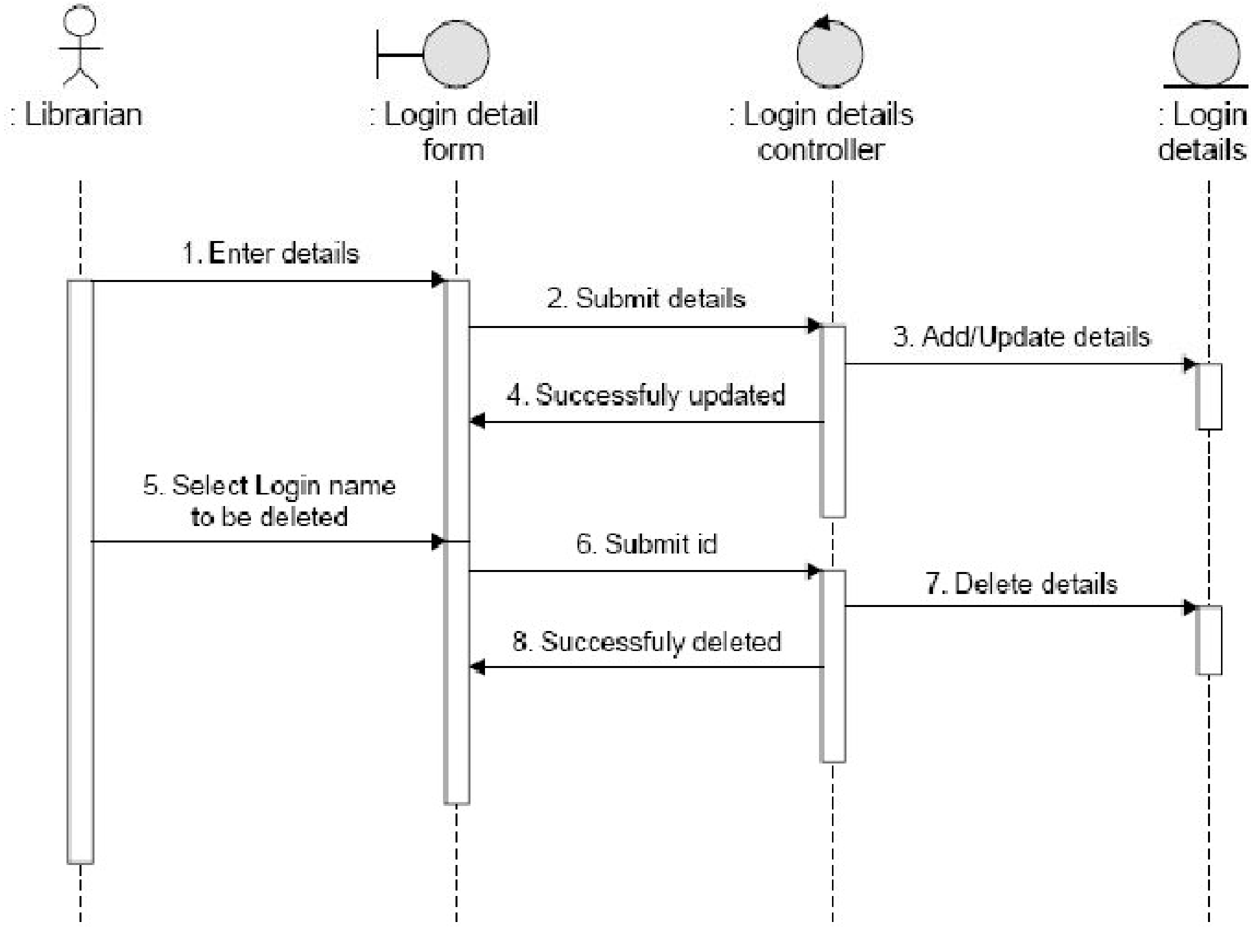
Sequence diagram—maintain catalog

Software Design



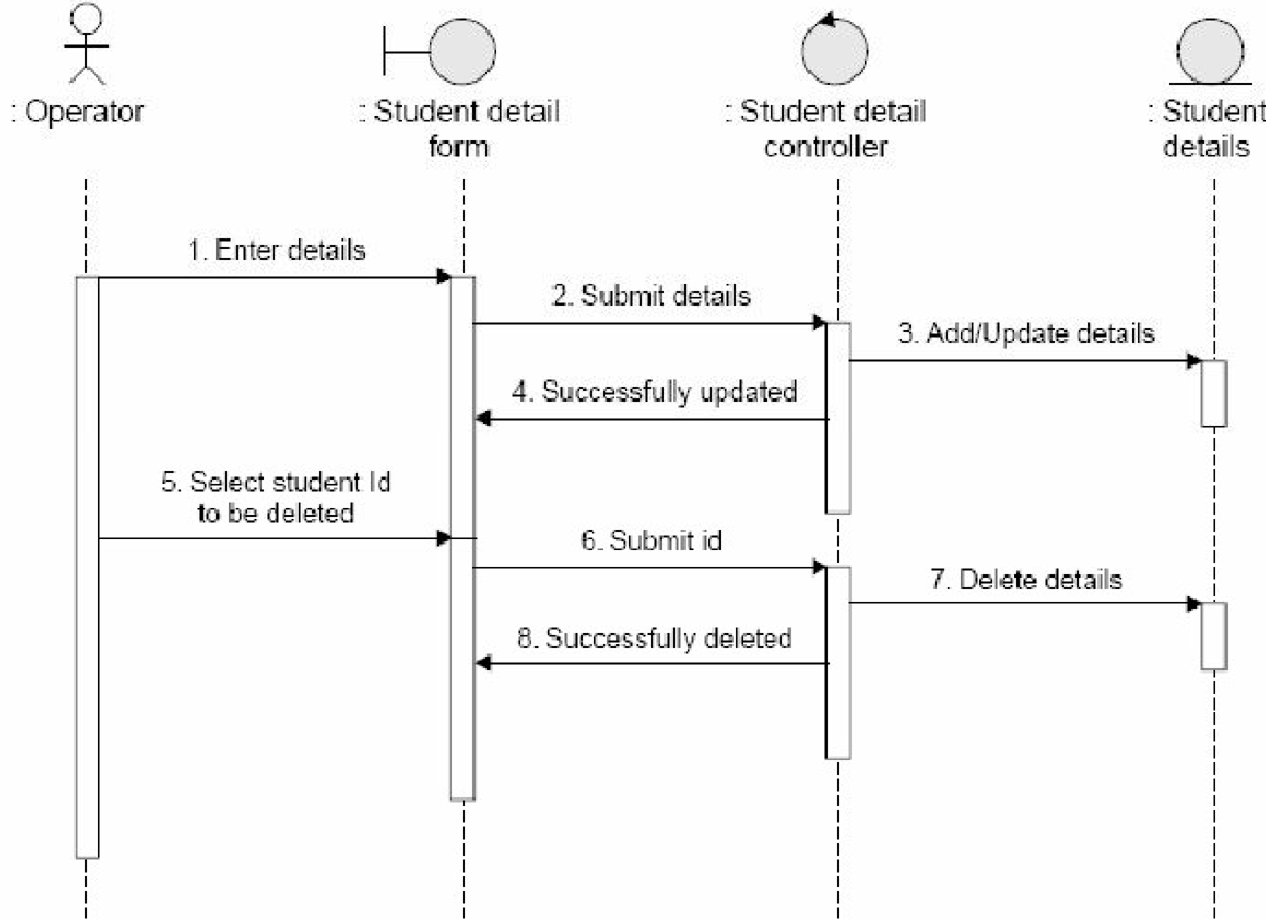
Sequence diagram—generate reports

Software Design



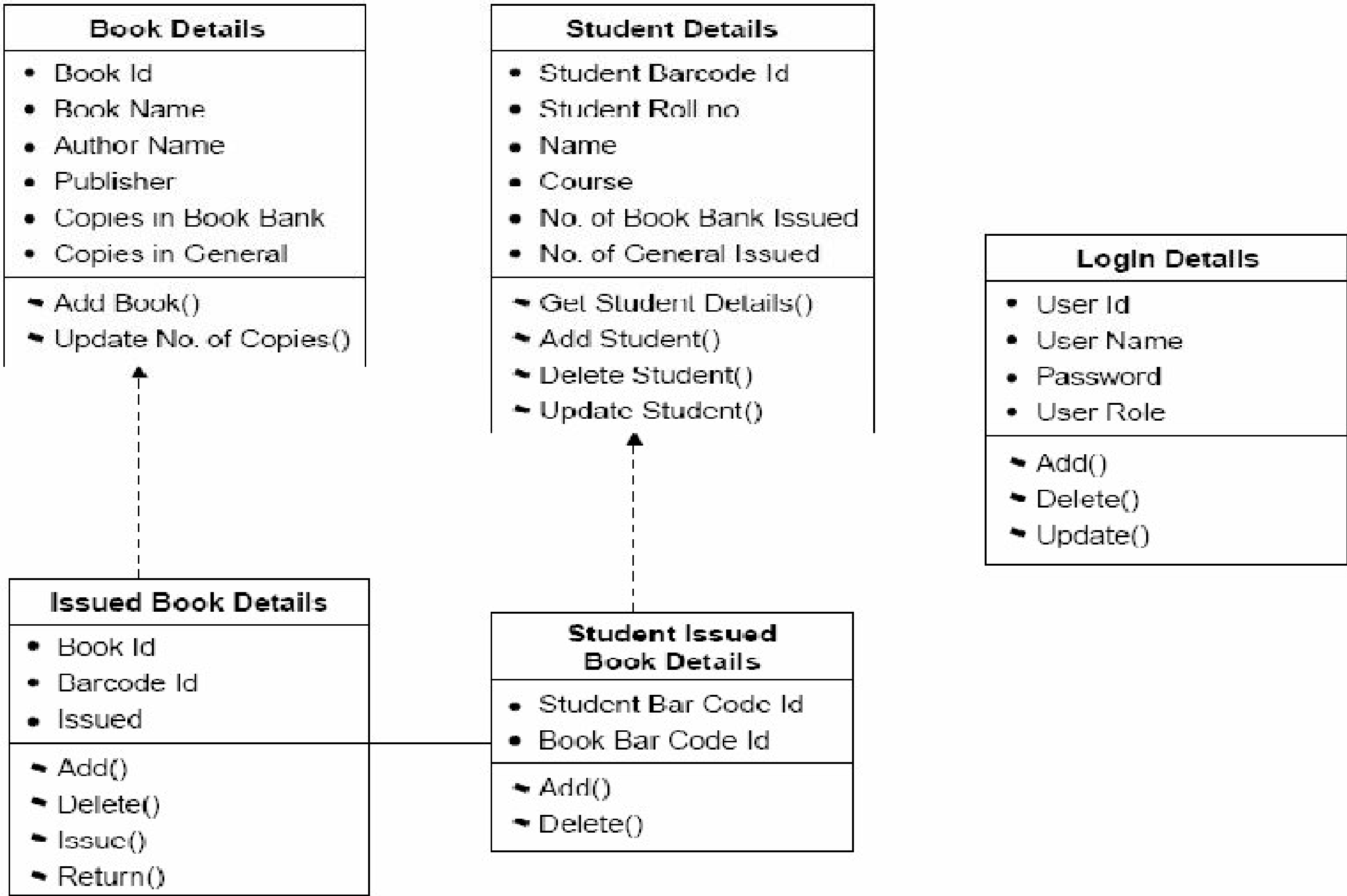
Sequence diagram—maintain login

Software Design



Sequence diagram—maintain student details

Software Design



Class diagram of entity classes

Multiple Choice Questions

Note: Choose most appropriate answer of the following questions:

5.1 The most desirable form of coupling is
(a) Control Coupling (b) Data Coupling
(c) Common Coupling (d) Content Coupling

5.2 The worst type of coupling is
(a) Content coupling (b) Common coupling
(c) External coupling (d) Data coupling

5.3 The most desirable form of cohesion is
(a) Logical cohesion (b) Procedural cohesion
(c) Functional cohesion (d) Temporal cohesion

5.4 The worst type of cohesion is
(a) Temporal cohesion (b) Coincidental cohesion
(c) Logical cohesion (d) Sequential cohesion

5.5 Which one is not a strategy for design?
(a) Bottom up design (b) Top down design
(c) Embedded design (d) Hybrid design

Multiple Choice Questions

5.6 Temporal cohesion means

(a) Cohesion between temporary variables
(b) Cohesion between local variable
(c) Cohesion with respect to time
(d) Coincidental cohesion

5.7 Functional cohesion means

(a) Operations are part of single functional task and are placed in same procedures
(b) Operations are part of single functional task and are placed in multiple procedures
(c) Operations are part of multiple tasks
(d) None of the above

5.8 When two modules refer to the same global data area, they are related as

(a) External coupled (b) Data coupled
(c) Content coupled (d) Common coupled

5.9 The module in which instructions are related through flow of control is

(a) Temporal cohesion (b) Logical cohesion
(c) Procedural cohesion (d) Functional cohesion

Multiple Choice Questions

5.10 The relationship of data elements in a module is called

(a) Coupling
(b) Cohesion
(c) Modularity
(d) None of the above

5.11 A system that does not interact with external environment is called

(a) Closed system
(b) Logical system
(c) Open system
(d) Hierarchal system

5.12 The extent to which different modules are dependent upon each other is called

(a) Coupling
(b) Cohesion
(c) Modularity
(d) Stability

Exercises

5.1 What is design? Describe the difference between conceptual design and technical design.

5.2 Discuss the objectives of software design. How do we transform an informal design to a detailed design?

5.3 Do we design software when we "write" a program? What makes software design different from coding?

5.4 What is modularity? List the important properties of a modular system.

5.5 Define module coupling and explain different types of coupling.

5.6 Define module cohesion and explain different types of cohesion.

5.7 Discuss the objectives of modular software design. What are the effects of module coupling and cohesion?

5.8 If a module has logical cohesion, what kind of coupling is this module likely to have with others?

5.9 What problems are likely to arise if two modules have high coupling?

Exercises

5.10 What problems are likely to arise if a module has low cohesion?

5.11 Describe the various strategies of design. Which design strategy is most popular and practical?

5.12 If some existing modules are to be re-used in building a new system, which design strategy is used and why?

5.13 What is the difference between a flow chart and a structure chart?

5.14 Explain why it is important to use different notations to describe software designs.

5.15 List a few well-established function oriented software design techniques.

5.16 Define the following terms: Objects, Message, Abstraction, Class, Inheritance and Polymorphism.

5.17 What is the relationship between abstract data types and classes?

Exercises

5.18 Can we have inheritance without polymorphism? Explain.

5.19 Discuss the reasons for improvement using object-oriented design.

5.20 Explain the design guidelines that can be used to produce "good quality" classes or reusable classes.

5.21 List the points of a simplified design process.

5.22 Discuss the differences between object oriented and function oriented design.

5.23 What documents should be produced on completion of the design phase?

5.24 Can a system ever be completely "decoupled"? That is, can the degree of coupling be reduced so much that there is no coupling between modules?