

Software design

- Software design is the process to transform the user requirements into some suitable form, which helps the programmer in software coding and implementation.
- During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document.
- Hence the aim of this phase is to transform the SRS document into the design document.

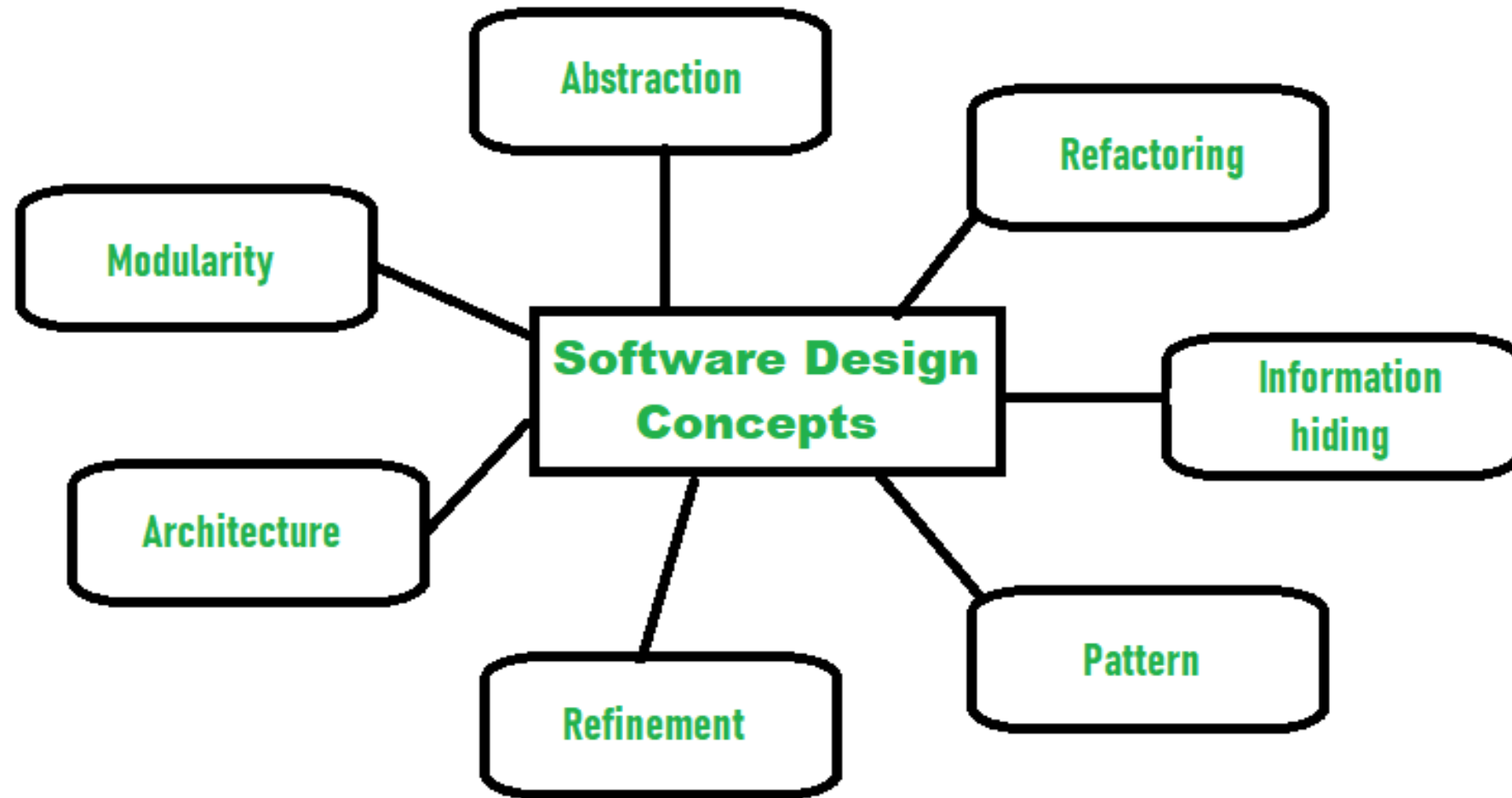
- The following items are designed and documented during the design phase
 - Different modules required.
 - Control relationships among modules.
 - Data structure among the different modules.
 - Algorithms required to implement among the individual modules.

Design concepts

- Concepts are defined as a **principal idea** or **invention** that comes into our mind or in thought to understand something.
- The **software design concept** simply means **the idea or principle** behind the design.
- It describes how you plan to solve the problem of designing software, the logic, or thinking behind how you will design software.
- It allows the software engineer to create the model of the system or software or product that is to be developed or built.
- The software design concept provides a supporting and essential structure or model for developing the right software.
- There are many concepts of software design and some of them are given below:

Design concept

- A set of fundamental software design concept has evolved over the history of software engineering
- Each helps you to answer the following questions
 - What criteria can be used to partition the software into individual components?
 - How is function or data structure detail separated from a conceptual representation of the software?
 - What uniform criteria define the quality of software design?



Abstraction

- Display only relevant attributes and hides the unnecessary details
- Many level of abstractions

Highest level of abstraction

- Solution is stated in broad terms using the language of the problem environment

Lower level of abstraction

- More detail level of solution is provided

Procedural abstraction

- Refers to a sequence of instructions and limit function

Data abstraction

- Data abstraction means hiding the details about the data

Control abstraction

- control abstraction means hiding the implementation details

Architecture

- Software architecture is the **structure or organization** of program components (modules), the manners in which these **components interacts**, and the structure of data that are that are used by these components.
- In a broader sense, however, components can be generalized to represent major system elements and their interactions
- Shaw and Garlan [SHA95a] describe a set of properties that should be specified as part of an architectural design:

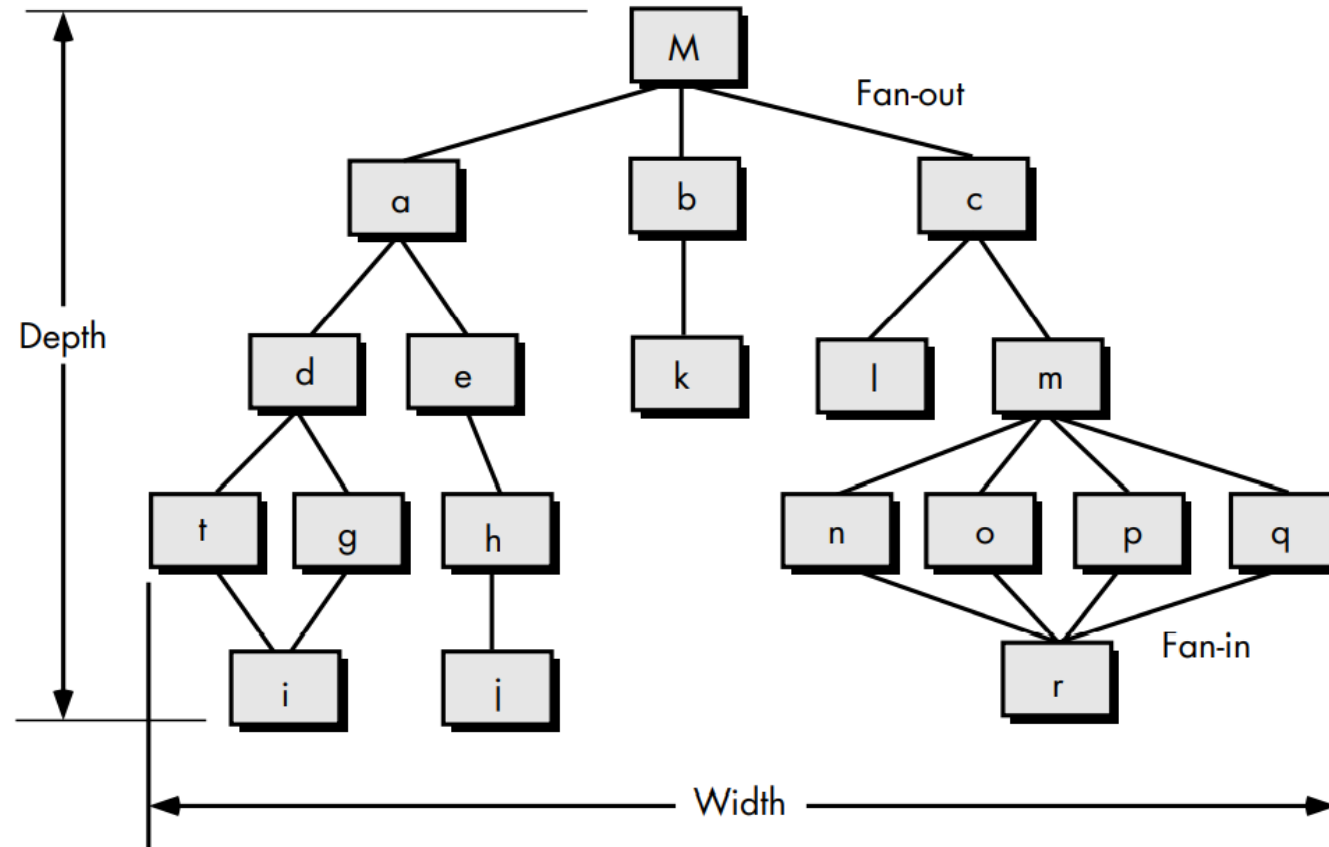
- **Structural properties.** This aspect of the architectural design representation defines the **components of a system** (e.g., modules, objects) and the manner in which those components are packaged and interact with one another.
- For example, objects are packaged to encapsulate both data and the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

- The architectural design can be represented using number of different models
- **Structural models**→represents architecture as an organized collection of program components
- **Framework model**→increase the level of design abstraction by attempting to identifying repeatable architectural design framework(patterns) that are encountered in similar type of applications
- **Dynamic models**→ addresses the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events

Call and return architectural style

FIGURE 13.3

Structure terminology for a call and return architectural style



Pattern

- Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of solution to that problem, in such a way that you can use this solution a million times over , without ever doing it the same way twice.
- Provides descriptions to enable a designer to determine the following
 - Whether the pattern is applicable to the current work
 - Whether the pattern can be reused
 - Whether the pattern can be serve as a guide for developing a similar but functionally or structurally different pattern

Modularity

- Divide the software into separately named and addressable components sometimes called modules
- The partitioning is done in order to reduce complexity to some degree.
- Based on divide and conquer strategy
- It is easier to solve a complex problem when broken into sub modules.
- Modules are integrated to satisfy the problem requirements

Modularity

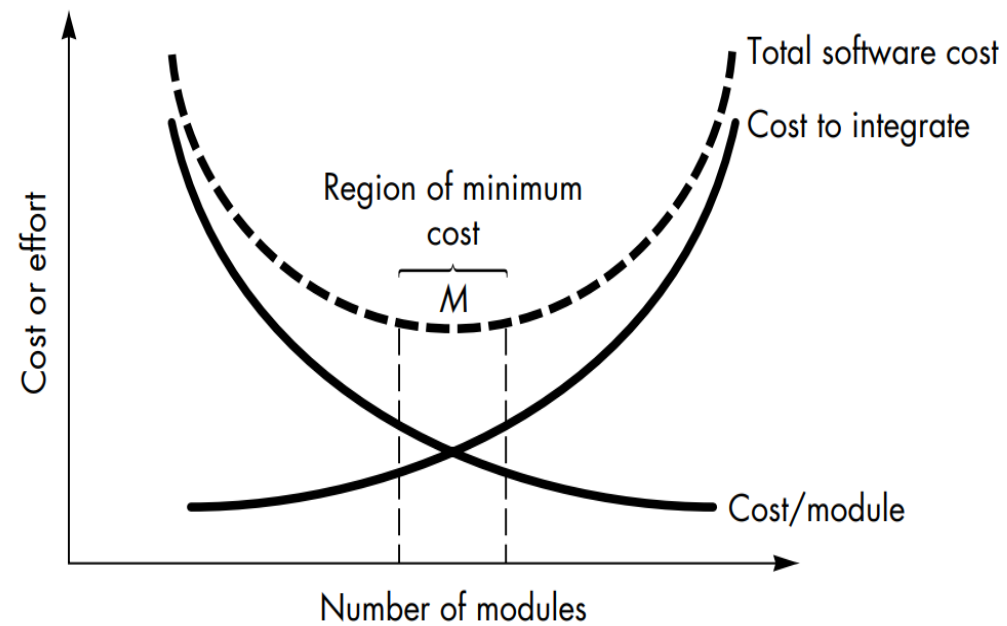
- software is divided into separately named and addressable components, often called modules, that are integrated to satisfy problem requirements
- Given the same set of requirements, more modules means smaller individual size.
- However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows.
- These characteristics lead to a total cost or effort curve shown in the figure

Modularity

CHAPTER 13 DESIGN CONCEPTS AND PRINCIPLES

FIGURE 13.2

Modularity
and software
cost



Modularity

- We should modularize, but care should be taken to stay in the vicinity of M.
- Undermodularity or overmodularity should be avoided.
- Don't overmodularize. The simplicity of each module will be overshadowed by the complexity of integration
- How can we **evaluate a design method to determine if it will lead to effective modularity?**
- Meyer [MEY88] defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

Methods for effective modularity

- **Modular decomposability.**

If a design method provides a **systematic mechanism for decomposing** the problem into subproblems, it will **reduce the complexity** of the overall problem, thereby achieving an effective modular solution.

- **Modular composability.**

If a design method enables existing (**reusable**) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

Methods for effective modularity

- **Modular understandability.**

If a module can be understood as a **standalone unit** (without reference to other modules), it will be easier to build and easier to change.

- **Modular continuity.**

If small changes to the system requirements result in **changes to individual modules**, rather than systemwide changes, the impact of change-induced side effects will be minimized.

- **Modular protection.**

If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

Information hiding

- Information contained within a module is inaccessible to **other module** who **do not** need such information
- Achieved by defining a set of independent modules that communicate with one another only that **information necessary** to achieve software functions.
- Provides the greatest benefits when modifications are required during testing and later.
- Errors introduced during modifications are less likely to propagate to other locations within the software.

Refinements

- Process of elaboration from high level abstractions to the lowest level abstractions
- High level abstraction begins with a statements or functions
- Refinements cause the designer to elaborate providing more and more details at successive levels of abstractions
- **Abstractions** and **refinements** are complementary concepts.

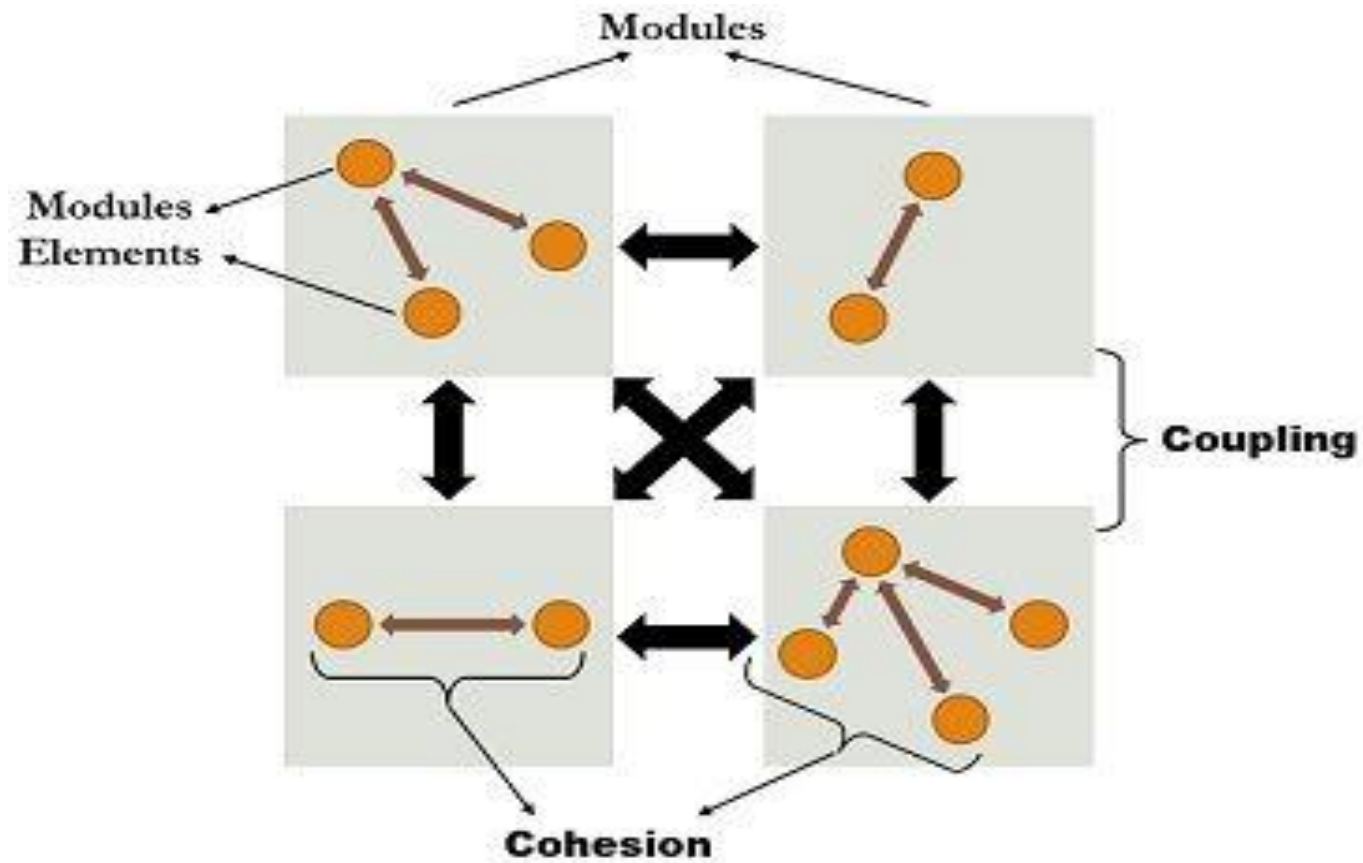
Refactoring

- Organization technique that simplifies the design of the components without changing its functions or behaviour
- Examines for redundancy, unused design elements, and inefficient or unnecessary algorithms
- Any design failure can be corrected.

Functional/modular independence

- A direct outgrowth of modularity, abstraction, information hiding
- Divide software/problem into number of modules such that all modules independent to each other.
- Different module has different functionality
- Easier to develop and have simple interface
- Easier to maintain because secondary effect caused by design or code modifications are limited, error propagation is reduced and reusable modules are possible.
- Independence is accessed by two quantitative criteria
 - Cohesion
 - Coupling

Coupling and cohesion



Cohesion and coupling

- Coupling is the degree of interdependence between software modules;
- a measure of how closely connected two routines or modules are
- Strength of the relationship between the modules
- Coupling is usually contrasted with the cohesion.
- Low coupling is often a sign of a well structured computer system and good design.

- Module → module here refers to a subroutine → set of statements with own variables

- **Content coupling**
- When one module modifies or relies on the internal working of another module
- E.g accessing local data of another module

- **Common coupling**

- Occurs when two modules share a same global data/variable.
- Changing the shared resource might imply changing all the modules using it.

- **External coupling**

- External coupling occurs when two coupling share an externally imposed data format, communication protocol, device or interface.
- This is basically related to the communication to the external tools and devices

- **Control coupling**
- When one module controlling the flow of another, by passing it information on what to do (e.g.flag)

- **Stamp coupling/data structured coupling**
- Stamp coupling occurs when module shares a composite data structure and use only a part of it
- E.g passing a whole record to a function that only need one field of it
- In this situations a modifications in a field that a module doesnot need may lead to changing the way the module reads the record

- **Data coupling**
- Data coupling when a module share a data through, for example ,passing parameters

Disadvantages of coupling

- a tight coupled system have following disadvantages
- A change in one module usually forces a ripple effect of a change to other modules
- Assembly of modules might require more effort and time due to increase intermodular dependency
- A particular module is harder to reuse or test because dependent modules must be included.

Cohesion

- Cohesion refers to the degree to which the elements of a module belong together.
- Thus cohesion measure a strength of relationship between pieces of functionality within a given module.
- Highly cohesive systems functionality is strongly related.

- **Coincidental cohesion (worst)**
- Coincidental cohesion is when part of module are grouped arbitrarily
- The only relationship between the parts is that they have been grouped together.

- **Logical cohesion**
- Logical cohesion when parts of a module are grouped because they are logically categorized to do the same things even though they are different by nature
- E.g grouping all mouse and keyboard handling routine

- **Temporal cohesion**

- Temporal cohesion is when parts of a module are grouped by when they are processed – the parts are processed at a particular time in a program execution
- E.g a function which is called after catching an exception which closes open files, creates an error log, and notifies the user.

- **Procedural cohesion**

- Parts of module are grouped because they always follows a certain sequence of execution
- E.g function which check file permission and then open a file

- **Communicational /informational cohesion**
- Communicational cohesion is when a part of module are grouped because they operate on the same data

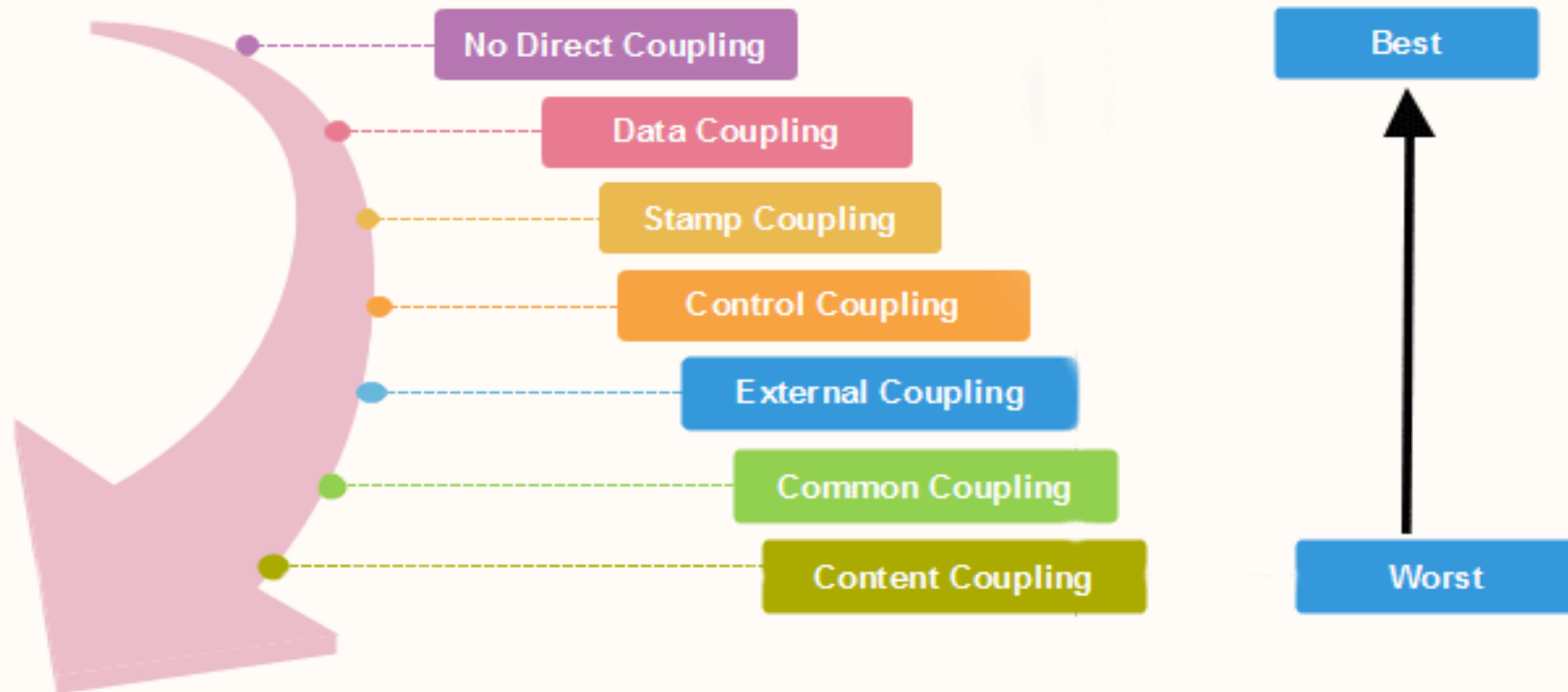
- **Sequential cohesion**
- When parts of a module are grouped because the output from one part is the input to another part
- **Functional cohesion(best)**
- Functional cohesion is when parts of module are grouped because they all contribute **to a single well defined task of the module.**
- Most desirable but may not be achievable.

Cohesion vs coupling

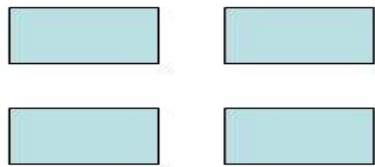
| | Cohesion | Coupling |
|---|---|---|
| 1 | Cohesion is the degree to which the elements inside a module belong together. | Coupling is the degree of interdependence between the modules. |
| 2 | A module with high cohesion contains elements that are tightly related to each other and united in their purpose. | Two modules have high coupling (or tight coupling) if they are closely connected and dependent on each other. |
| 3 | A module is said to have low cohesion if it contains unrelated elements. | Modules with low coupling among them work mostly independently of each other. |
| 4 | Highly cohesive modules reflect higher quality of software design | Loose coupling reflects the higher quality of software design |

Types of Modules Coupling

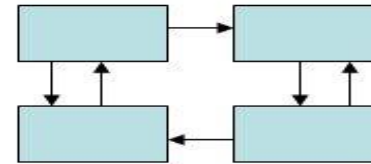
There are various types of module Coupling are as follows:



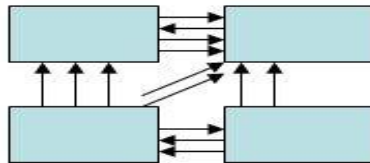
Coupling: Degree of dependence among components



No dependencies



Loosely coupled-some dependencies



Highly coupled-many dependencies

High coupling makes modifying parts of the system difficult, e.g., modifying a component affects all the components to which the component is connected.

Architectural Design

- 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. Rather, it is a representation that enables a software engineer to
 - analyze the effectiveness of the design in meeting its stated requirements,
 - consider architectural alternatives at a stage when making design changes is still relatively easy, and
 - reducing the risks associated with the construction of the software

Why Is Architecture Important?

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together”

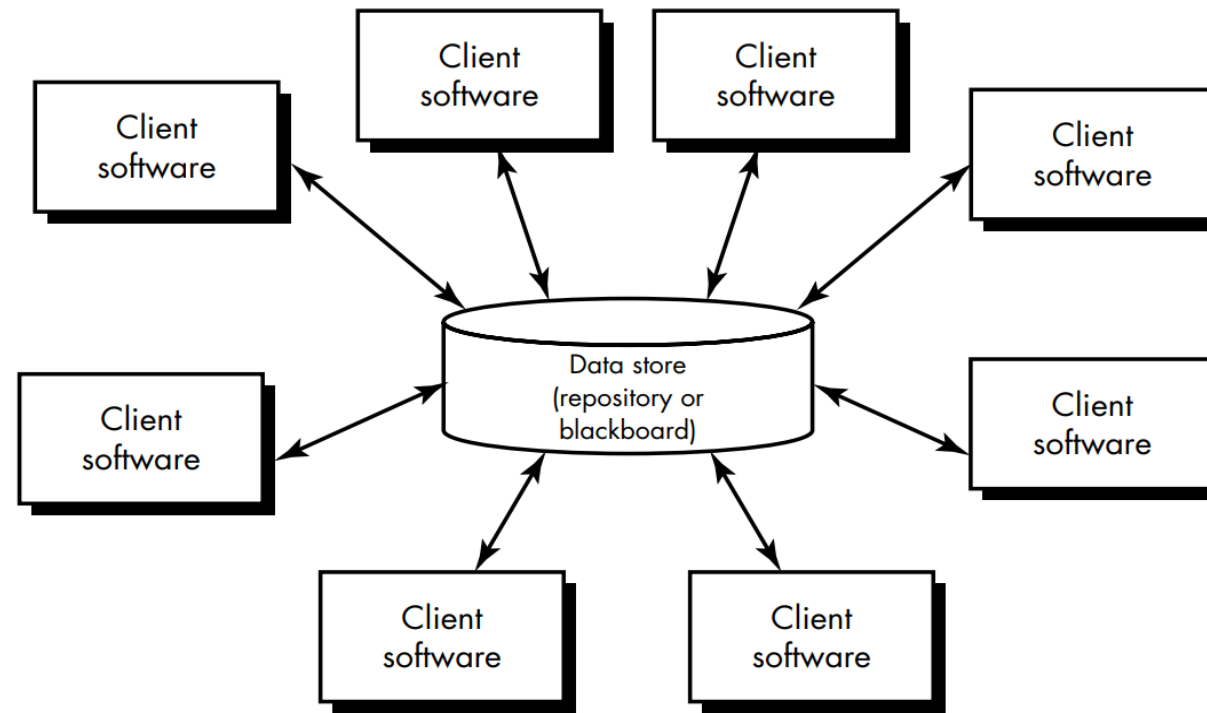
Architectural styles

- The software that is built for computer-based systems also exhibits one of many 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, coordinations 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.
- Client server architecture
- Data-flow architectures.
- Call and return architectures.
- Layered architectures.

Data centered Architecture

FIGURE 14.1
Data-centered
architecture



Data centered architecture

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- Client software accesses a central repository

Data centered architecture

- Data-centered architectures promote integrability. That is, existing components can be changed and new client components can be added to the architecture without concern about other clients (because the client components operate independently).
- In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes

Data centered architecture

| | |
|---------------|---|
| When used | You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool. |
| Advantages | Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place. |
| Disadvantages | The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult. |

Client server model

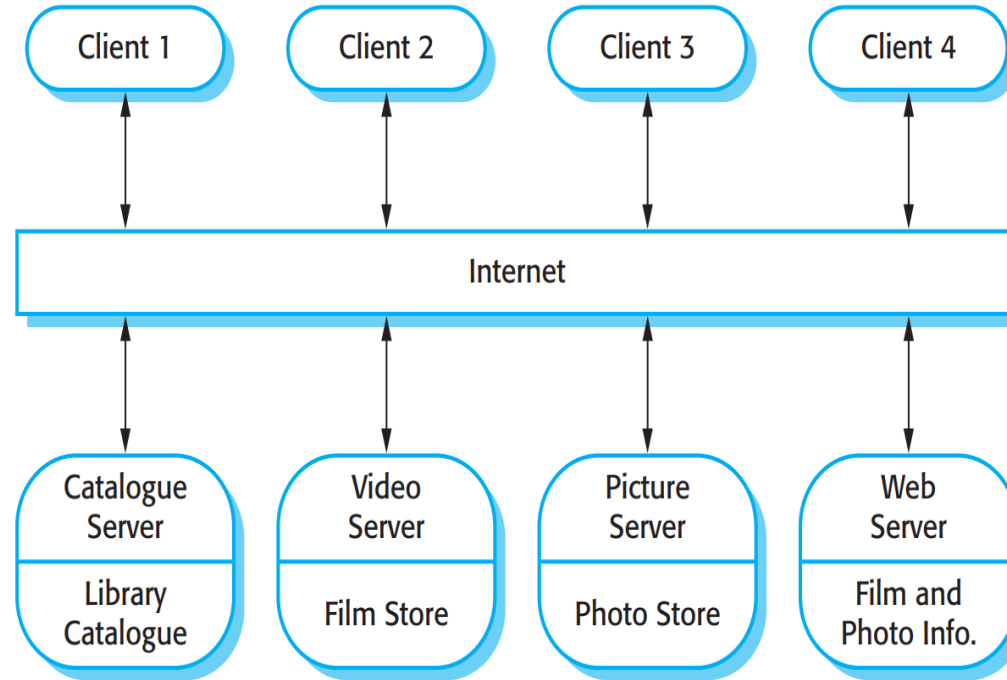


Figure 6.11 A client–server architecture for a film library

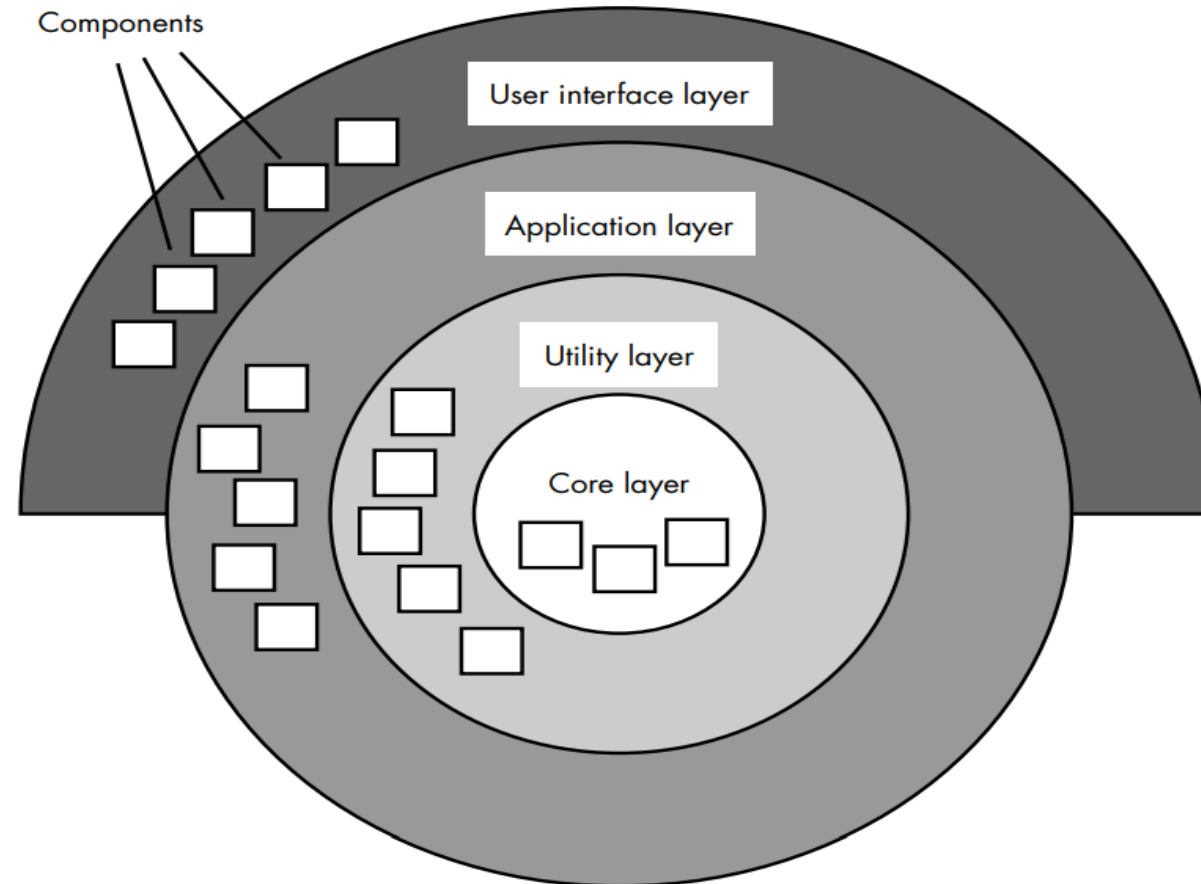
Client server model

- Figure 6.11 is an example of a system that is based on the client–server model.
- This is a multi-user, web-based system for providing a film and photograph library.
- In this system, several servers manage and display the different types of media.
- Video frames need to be transmitted quickly and in synchrony but at relatively low resolution.
- They may be compressed in a store, so the video server can handle video compression and decompression in different formats.
- Still pictures, however, must be maintained at a high resolution, so it is appropriate to maintain them on a separate server.
- The catalog must be able to deal with a variety of queries and provide links into the web information system that includes data about the film and video clips,
- THE client program is simply an integrated user interface, constructed using a web browser, to access these services

| Name | Client-server |
|---------------|--|
| Description | In a client-server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them. |
| Example | Figure 6.11 is an example of a film and video/DVD library organized as a client-server system. |
| When used | Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable. |
| Advantages | The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services. |
| Disadvantages | Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations. |

Layered architecture

FIGURE 14.3
Layered
architecture



- The basic structure of a layered architecture is illustrated in Figure 14.3. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing
- Intermediate layers provide utility services and application software functions
- These architectural styles are only a small subset of those available to the software designer. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural pattern (style) or combination of patterns (styles) that best fits those characteristics and constraints can be chosen.
- In many cases, more than one pattern might be appropriate and alternative architectural styles might be designed and evaluated.

Layered architecture – next example

168 Chapter 6 ■ Architectural design

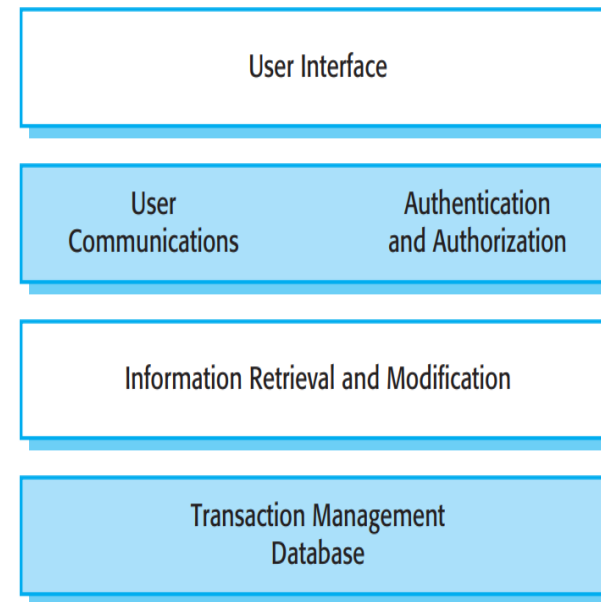
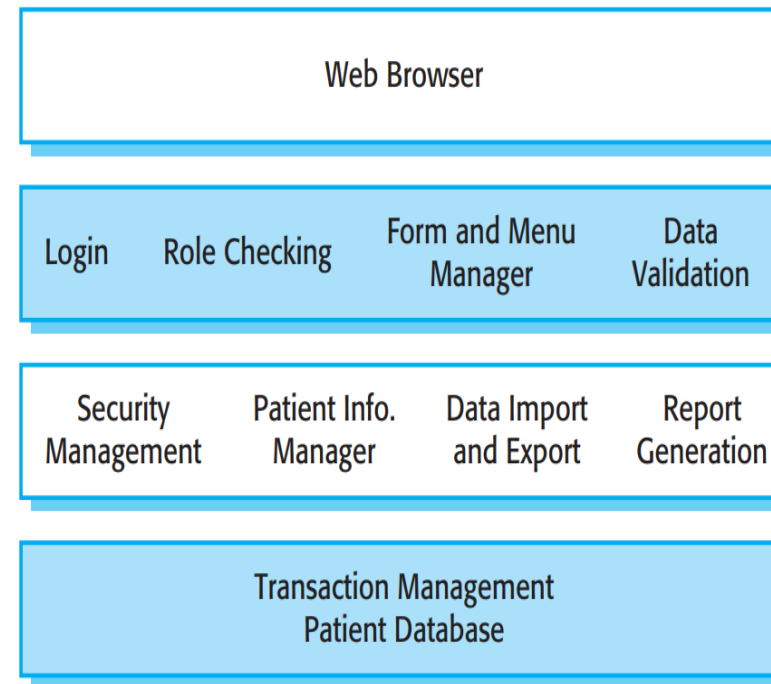


Figure 6.16 Layered information system architecture

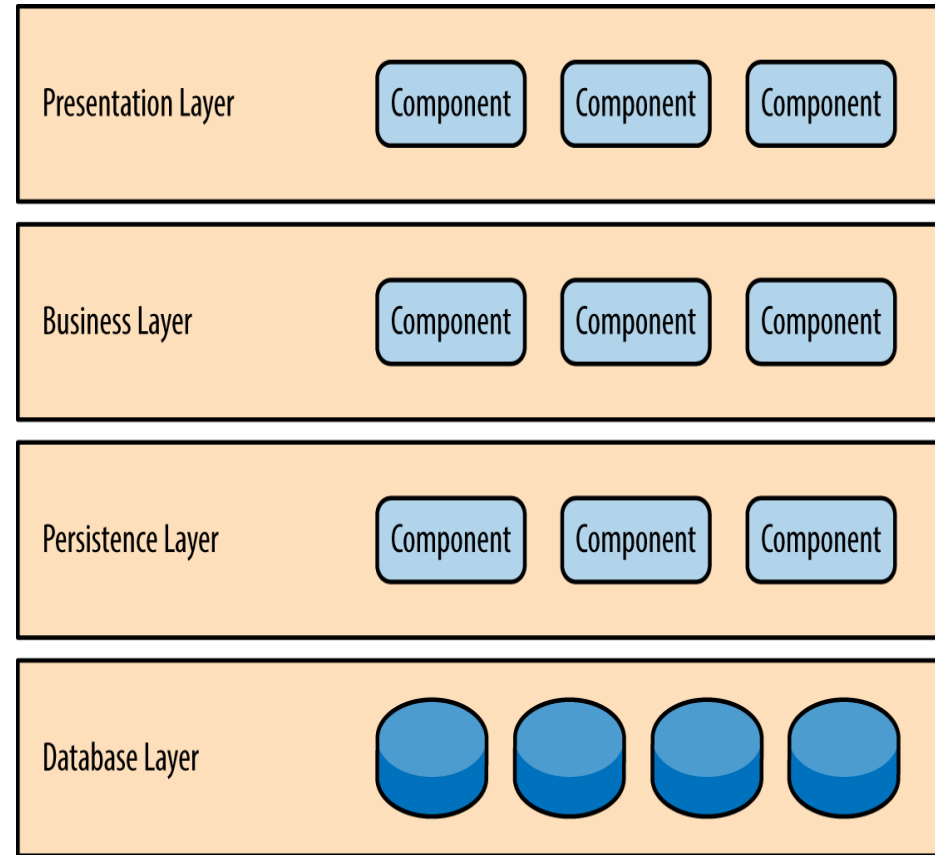
- The top layer is responsible for implementing the user interface. In this case, the UI has been implemented using a web browser.
- The second layer provides the user interface functionality that is delivered through the web browser.
- It includes components to allow users to log in to the system and checking components that ensure that the operations they use are allowed by their role.
- This layer includes form and menu management components that present information to users, and data validation components that check information consistency

- The third layer implements the functionality of the system and provides components that implement system security, patient information creation and updating, import and export of patient data from other databases, and report generators that create management reports.
- Finally, the lowest layer, which is built using a commercial database management system, provides transaction management and persistent data storage.

Figure 6.17 The architecture of the MHC-PMS



Example -3; figure :: Layered architecture



- Each layer of the layered architecture pattern has a specific role and responsibility within the application.
- For example, a presentation layer would be responsible for handling all user interface and browser communication logic,
- whereas a business layer would be responsible for executing specific business rules associated with the request.
- Each layer in the architecture forms an abstraction around the work that needs to be done to satisfy a particular business request.
- For example, the presentation layer doesn't need to know or worry about *how* to get customer data;
- it only needs to display that information on a screen in particular format.
- Similarly, the business layer doesn't need to be concerned about how to format customer data for display on a screen or even where the customer data is coming from;
- it only needs to get the data from the persistence layer, perform business logic against the data (e.g., calculate values or aggregate data), and pass that information up to the presentation layer.

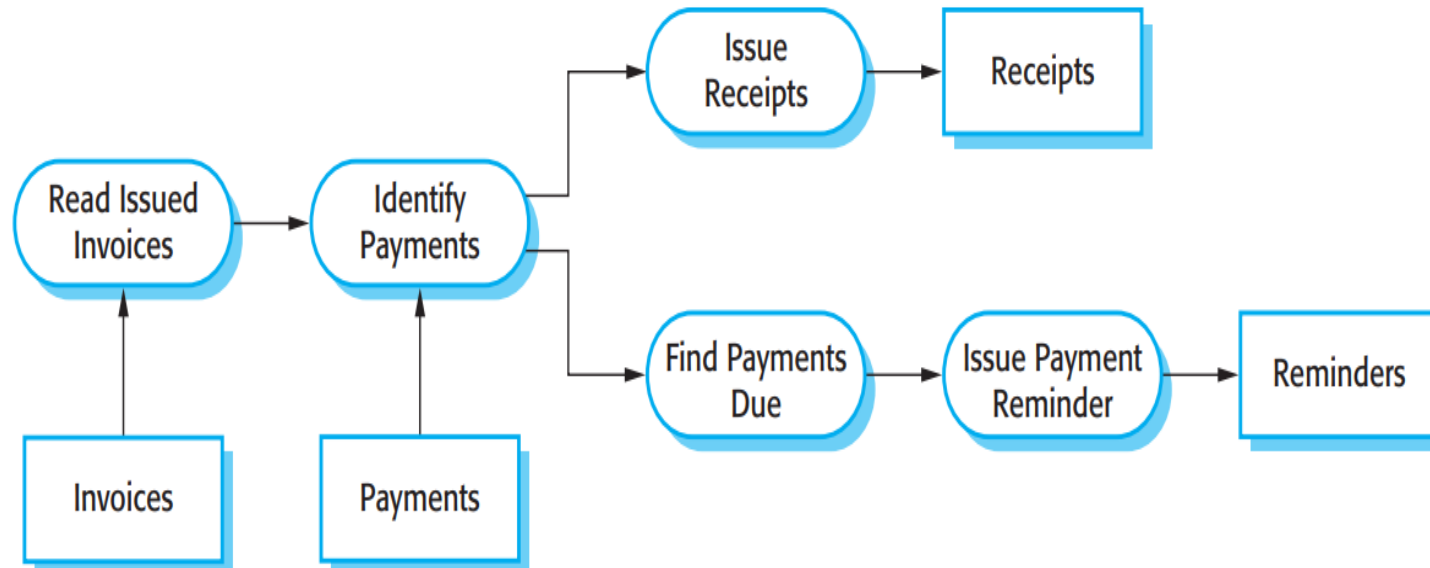
- One of the powerful features of the layered architecture pattern is the *separation of concerns* among components.
- Components within a specific layer deal only with logic that pertains to that layer.
- For example, components in the presentation layer deal only with presentation logic, whereas components residing in the business layer deal only with business logic.
- This type of component classification makes it easy to build effective roles and responsibility models into your architecture, and also makes it easy to develop, test, govern, and maintain applications using this architecture pattern due to well-defined component interfaces and limited component scope.

MVC

- MVC stands for Model View Controller
- MVC divides a software application into three parts
 - Model
 - View
 - controller

Pipes and filter

- Data flows from one to another and is transformed as it moves through the sequence.
- Each processing step is implemented as a transform.
- Input data flows through these transforms until converted to output.
- The transformations may execute sequentially or in parallel.
- The data can be processed by each transform item by item or in a single batch.



- An example of this type of system architecture, used in a batch processing application, is shown in Figure above.
- An organization has issued invoices to customers.
- Once a week, payments that have been made are reconciled/settle with the invoices.
- For those invoices that have been paid, a receipt is issued.
- For those invoices that have not been paid within the allowed payment time, a reminder is issued

User interface Design principles- golden rules

- Theo Mandel [MAN97] coins three “golden rules”:
 - 1. Place the user in control.
 - 2. Reduce the user’s memory load.
 - 3. Make the interface consistent.

Design principles- golden rules

How do we design interfaces that allow the user to maintain control?

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions
- Provide for flexible interaction
- Allow user interaction to be interruptible and undoable
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user
- Design for direct interaction with objects that appear on the screen.

Design principles- golden rules

Reduce the User's Memory Load

- Reduce demand on short-term memory
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real world metaphor
- Disclose information in a progressive fashion.

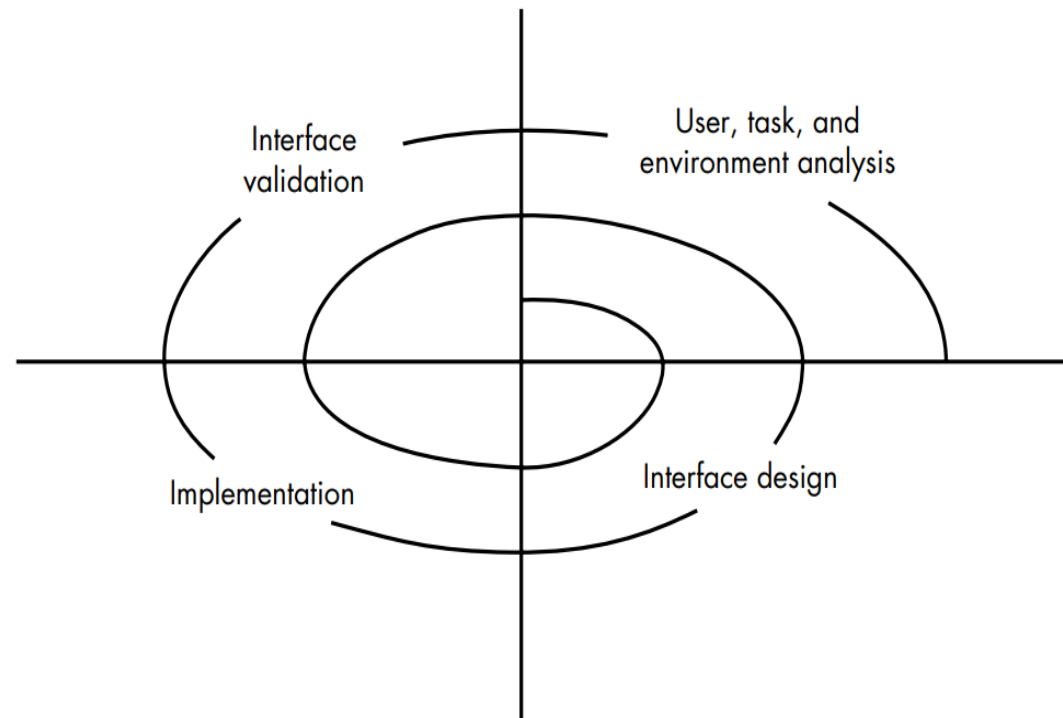
Make the Interface Consistent

- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

USER INTERFACE DESIGN process

FIGURE 15.1

The user interface design process



- The design process for user interfaces is **iterative** and can be represented using a spiral model
 - 1. User, task, and environment analysis
 - 2. Interface design
 - 3. Interface construction
 - 4. Interface validation

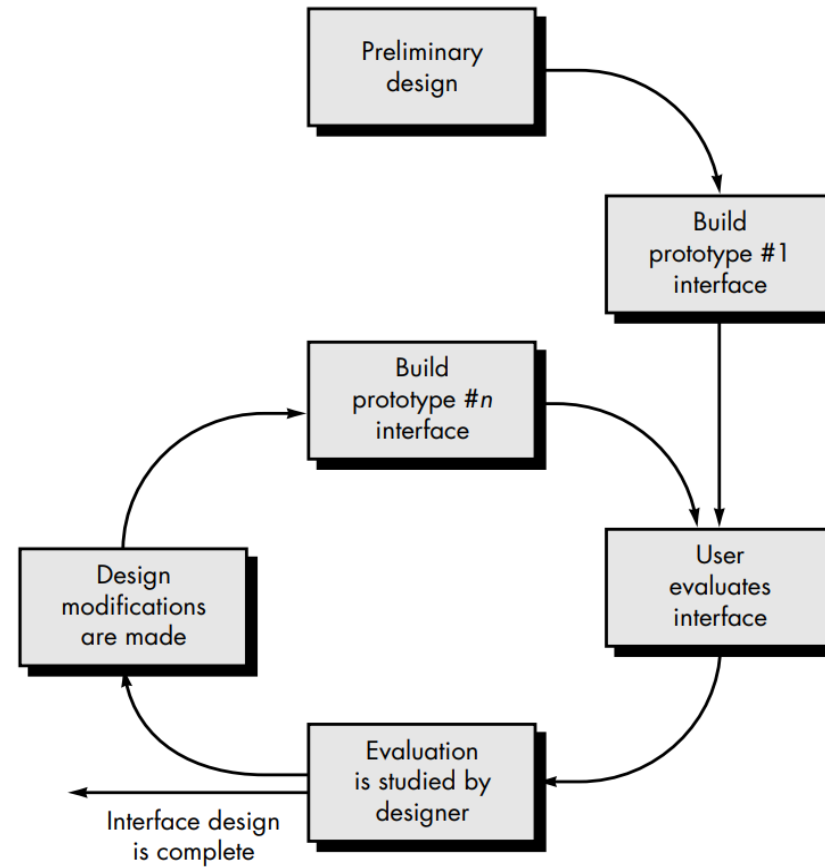
- The initial analysis activity focuses on the profile of the users who will interact with the system.
- Skill level, business understanding, and general receptiveness to the new system are recorded;
- and different user categories are defined. For each user category, requirements are elicited.
- The analysis of the user environment focuses on the physical work environment.

- The information gathered as part of the analysis activity is used to create an analysis model for the interface.
- Using this model as a basis, the design activity commence
- Once task analysis has been completed, all tasks (or objects/icons) and actions) required by the end-user have been identified in detail and the interface design activity commences.
- Implementation - user interface development tools can be used. Called **user-interface toolkits** or **user-interface development systems (UIDS)**, these tools provide components or objects that facilitate creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment.

Design evolution

FIGURE 15.3

The interface design evaluation cycle



Design Evaluation

- Once an operational user interface prototype has been created, it must be evaluated to determine whether it meets the needs of the user.
- Evaluation can span a formality spectrum that ranges from an informal "test drive," in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end-users.

Human computer interaction

- HCI is concerned with methods and tools for the development of human-computer interfaces, assessing the usability of computer systems and with broader issues about how people interact with computers.
- It is based on theories about how humans process information and interact with computers
- HCI helps to make interfaces that **increase productivity, enhance user experience,**
- Poorly designed machines lead to many unexpected problems, sometimes just user frustration, but sometimes, chaotic disasters.

- HCI is a **broad field** that reaches almost every industry. It often overlaps with areas like **user-centered design (UCD)**, **user interface (UI) design**.
- The designer of the user interface to a computer mainly focuses on
 - How the information from the user be provided to the computer system?
 - How can the information from the computer system be presented to the user?

Basic methods of user interactions

- Direct manipulations
 - User interacts directly with the objects on the screen
 - Fast better interactions
 - May be hard to implement
- Menu selection
 - User selects a command from the list of possible menu
 - E.g to delete a file , user selects the file and then selects the delete command.

- Form fill in
 - User can fill up certain data , perform certain actions to the filled data through menu.
- Command language
 - Issue special command
 - Execute certain code
 - E.g interactions through command prompt
- Natural language
 - Speech recognition
 - Voice command

How computer presents information to user?

- Text based presentation
- Visual representations such as pictures, diagrams, animations
- Graphical highlighting
- Audio/video information
- abstract visualizations using links (in case of large documents)

Procedural design (component level design)

- **Data, architectural, and interface design** must be translated into operational software.
- To accomplish this, the **design must be represented** at a level of abstraction that is **close to code**.
- **Component-level design** establishes the algorithmic detail required to manipulate **data structures**, effect communication between software components via their interfaces, and **implement the processing algorithms allocated to each component**.
- A software engineer performs component-level design.

Graphical Design Notation

- "A picture is worth a thousand words," but it's rather important to know which picture and which 1000 words.
- There is no question that graphical tools, such as the flowchart or box diagram, provide useful pictorial patterns that readily depict procedural detail.
- However, if graphical tools are misused, the wrong picture may lead to the wrong software

Why is it important?

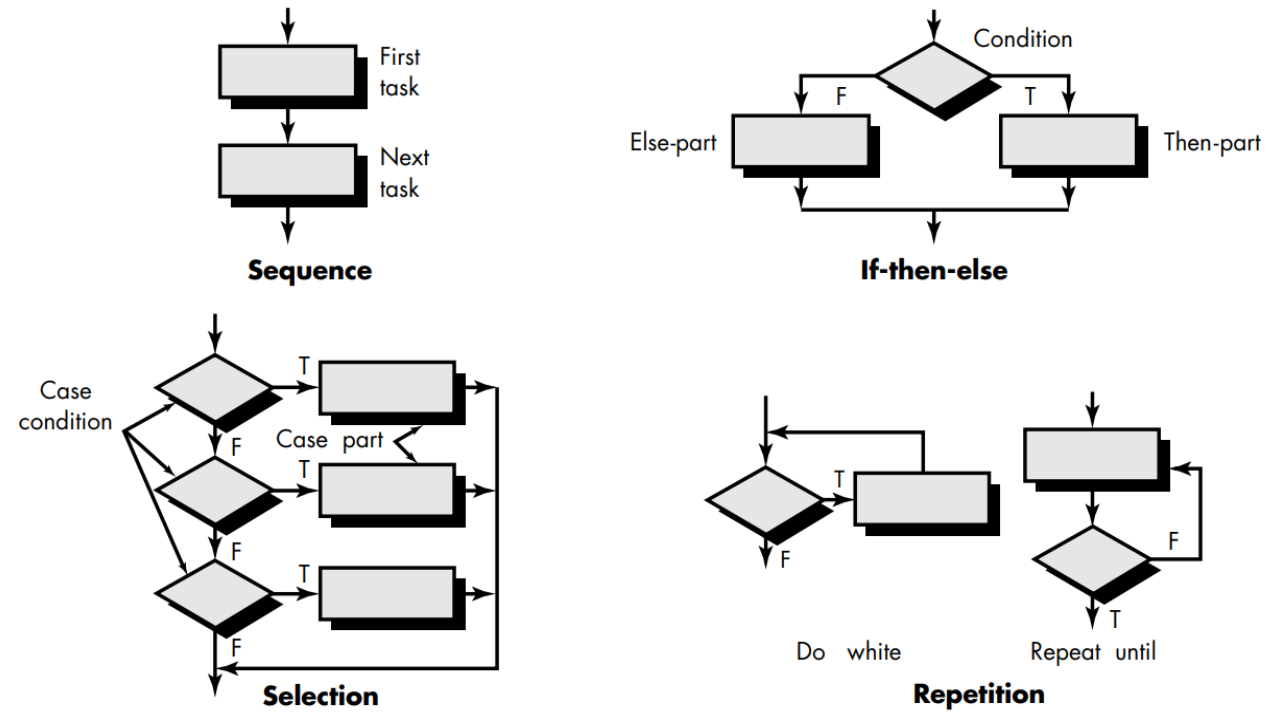
- You have to be able to determine whether the program will work before you build it.
- The component-level design represents the software in a way that allows you to review the details of the design for correctness and consistency with earlier design representations (i.e., the data, architectural, and interface designs).

What are the steps?

- Design representations of data, architecture, and interfaces form the foundation for component-level design
- The processing narrative for **each component is translated** into a **procedural design** model **using a set of structured programming constructs**.
- **Graphical, tabular, or text-based notation** is used to represent the design

- The **constructs** are **sequence**, **condition**, and **repetition**.
- **Sequence** implements **processing steps** that are essential in the specification of any algorithm.
- **Condition** provides the **facility for selected processing** based on some logical occurrence,
- and **repetition** allows for **looping**. These three constructs are fundamental to structured programming—an important component-level design technique

FIGURE 16.1
Flowchart
constructs

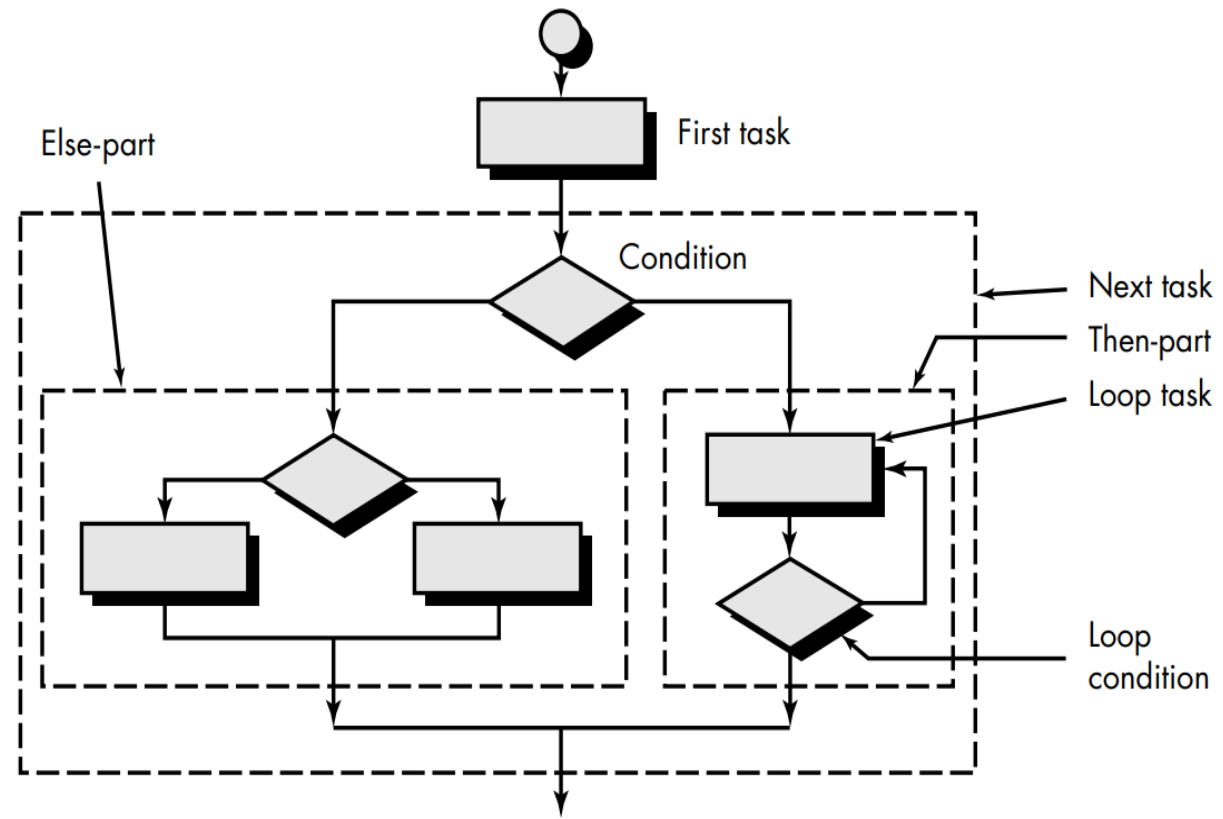


- A flowchart is quite simple pictorially.
 - A box is used to indicate a processing step.
 - A diamond represents a logical condition,
 - and arrows show the flow of control.
-
- The sequence is represented as two processing boxes connected by an line (arrow) of control
 - Condition, also called if then-else, is depicted as a decision diamond that if true, causes then-part processing to occur, and if false, invokes else-part processing.

- Repetition is represented using two slightly different forms.
- The do while tests a condition and executes a loop task repetitively as long as the condition holds true.
- A repeat until executes the loop task first, then tests a condition and repeats the task until the condition fails
- The selection (or select-case) construct shown in the figure is actually an extension of the if-else part

FIGURE 16.2

Nesting
constructs

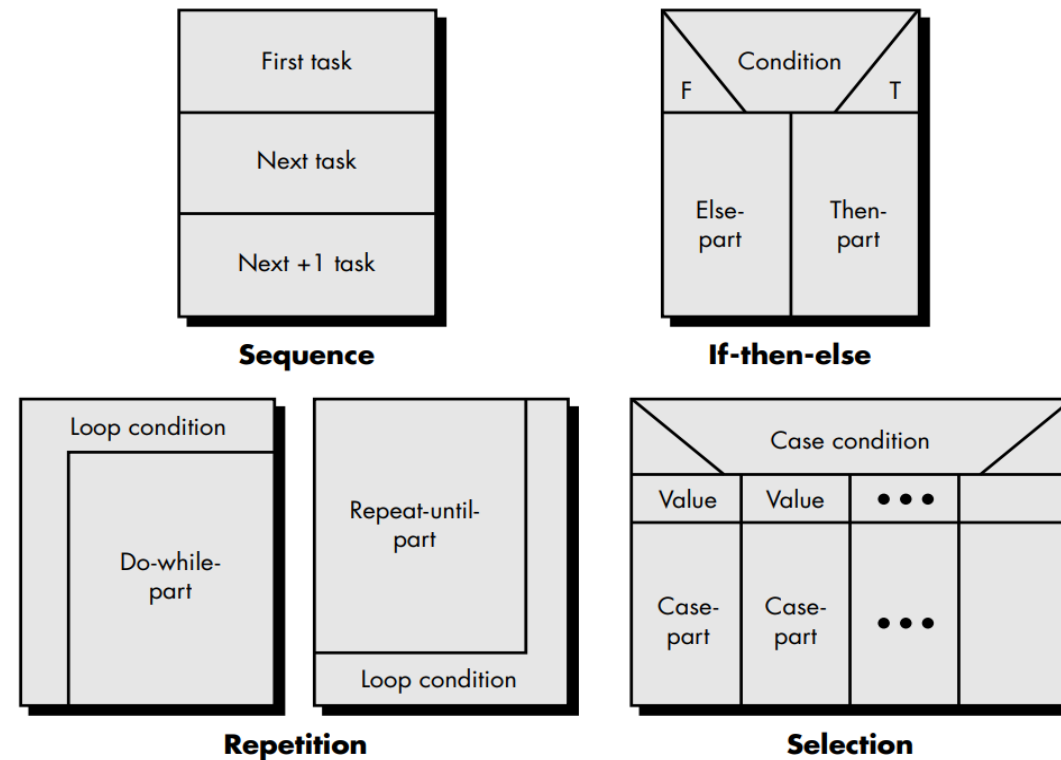


- Any program, regardless of application area or technical complexity, can be designed and implemented using only the three structured constructs.

Box diagram

- Another graphical design tool, the box diagram

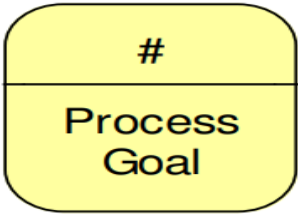

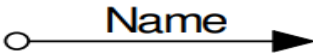
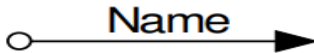
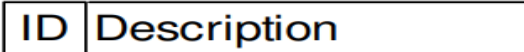
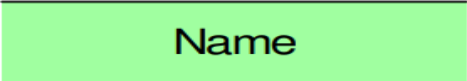


FIGURE 16.3
Box diagram
constructs



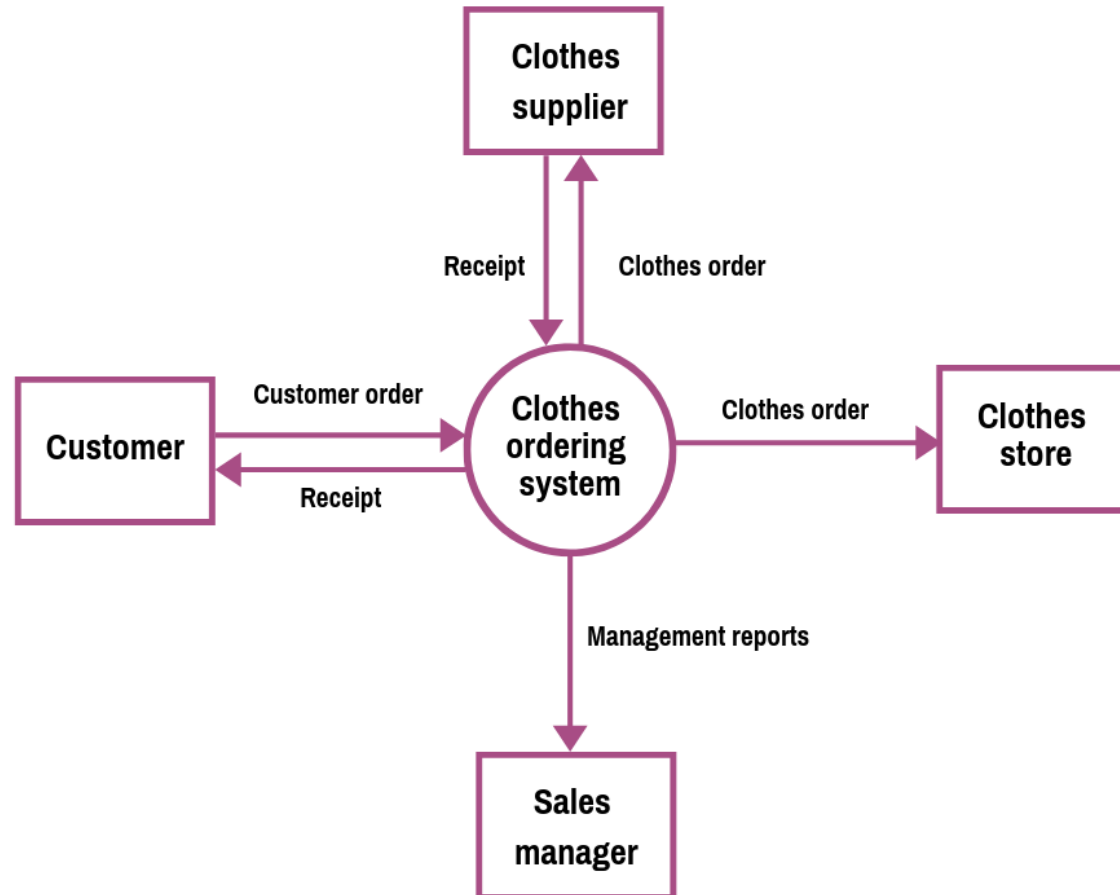
Program Design Language (pseudocode)

- Program design language (PDL), also called structured English or pseudocode
- The difference between PDL and a real programming language lies in the use of narrative text

Data flow diagrams

| Data Flow Diagram Symbols (showing the two major symbol sets) | | | |
|---|---|---|--|
| Name | Gane and Sarson Symbol | Yourdon Symbol | Description |
| Process |  |  | A major task that the program must perform |
| Data Flow |  |  | Data that flows into and out of each process |
| Data Store |  |  | An internal data structure that holds data during processing |
| External Entity |  |  | Devices or humans which input data and to which data is output |

Context 0



Context 1

