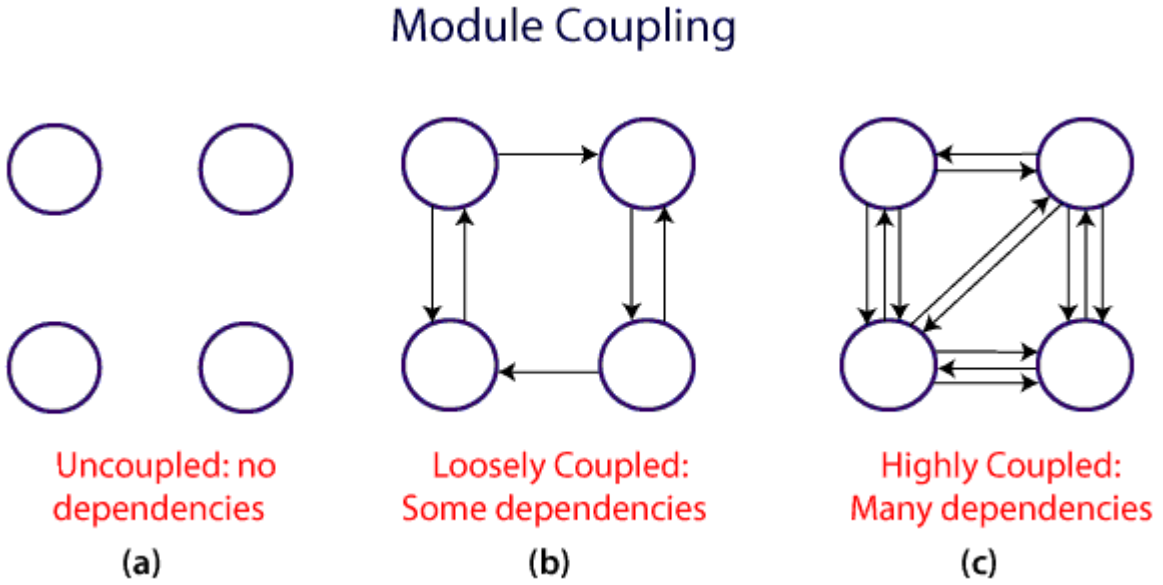
Coupling and Cohesion

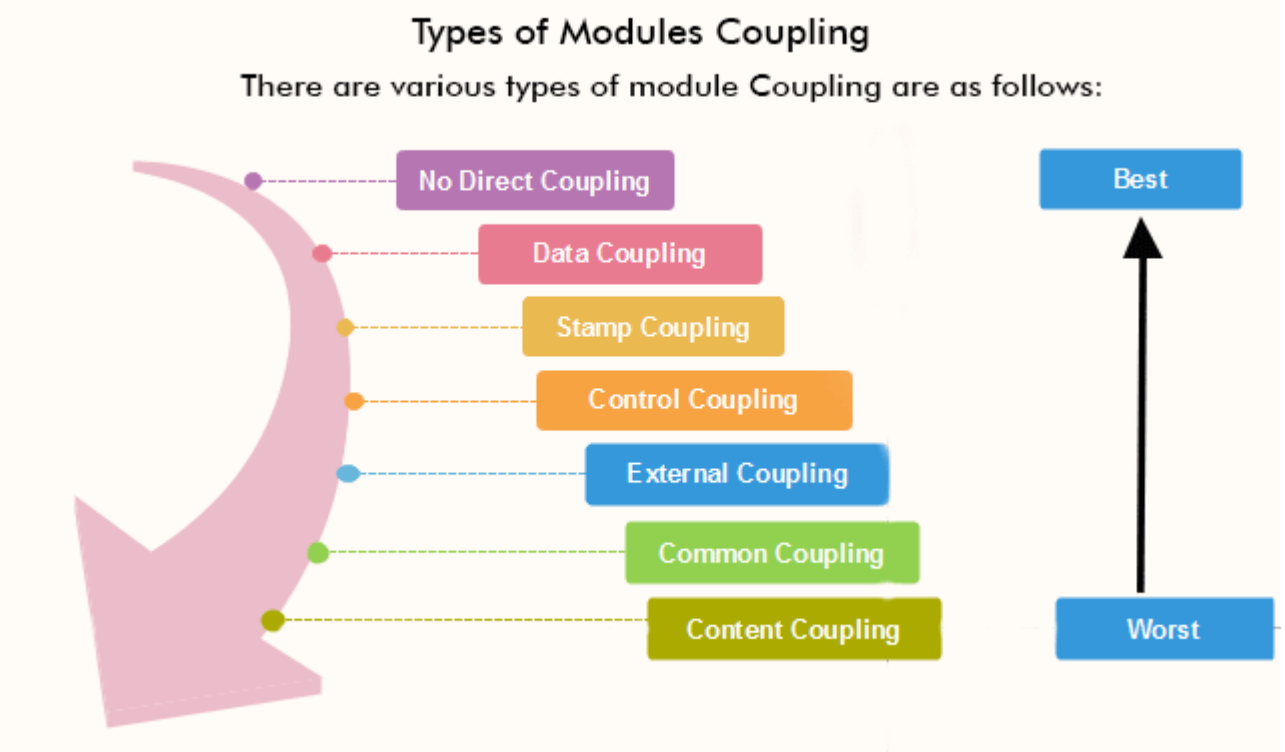
- Module Coupling

- In software engineering, the coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. **Uncoupled modules** have no interdependence at all within them.

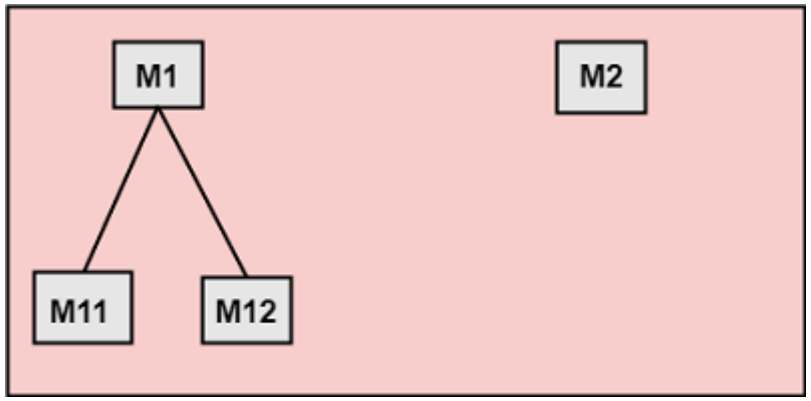
The various types of coupling techniques are shown in fig



- A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.

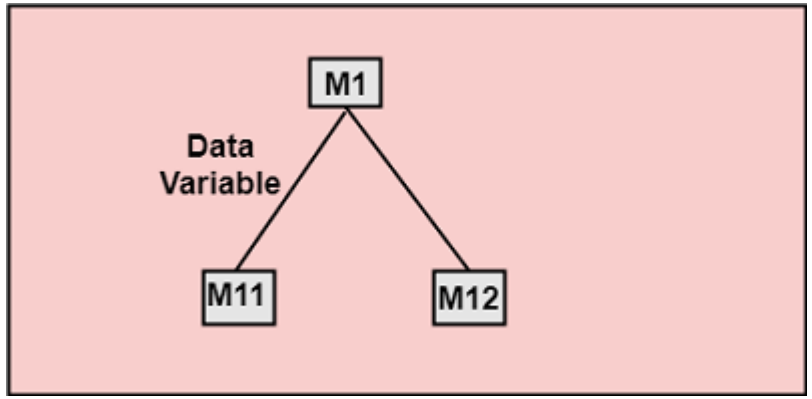


1. No Direct Coupling: There is no direct coupling between M1 and M2.

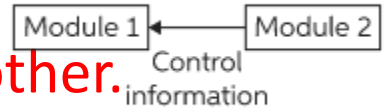


In this case, modules are subordinates to different modules. Therefore, no direct coupling.

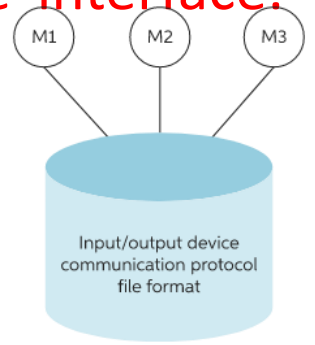
2. Data Coupling: When data of one module is passed to another module, this is called data coupling.



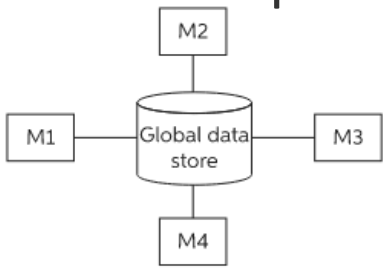
- **3.Stamp Coupling:** Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.
- **4. Control Coupling:** Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

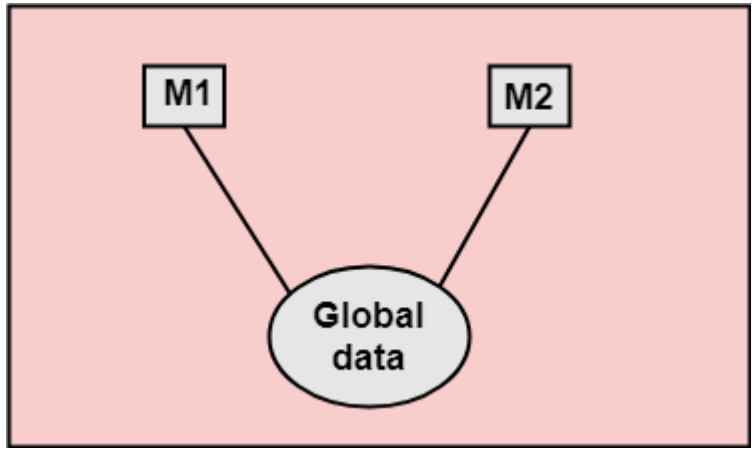


- **5. External Coupling:** External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.



- **6. Common Coupling:** Two modules are common coupled if they share information through some global data items.



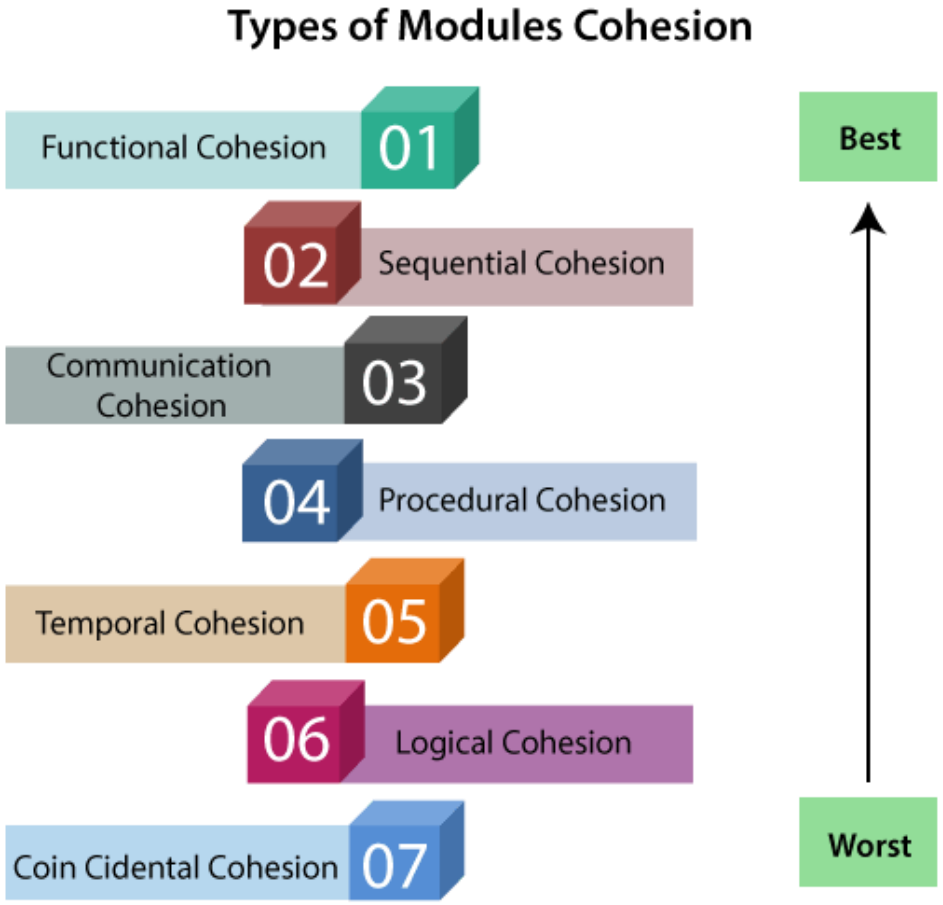


7. Content Coupling: Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

Module Cohesion

In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

Types of Modules Cohesion



1. Functional Cohesion: Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.

example of functional cohesion includes reading transaction records because all the elements related to single transaction are involved in it.

2. Sequential Cohesion: A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.

3.Communicational Cohesion: A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.

4.Procedural Cohesion: A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

5.Temporal Cohesion: When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.

6.Logical Cohesion: A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.

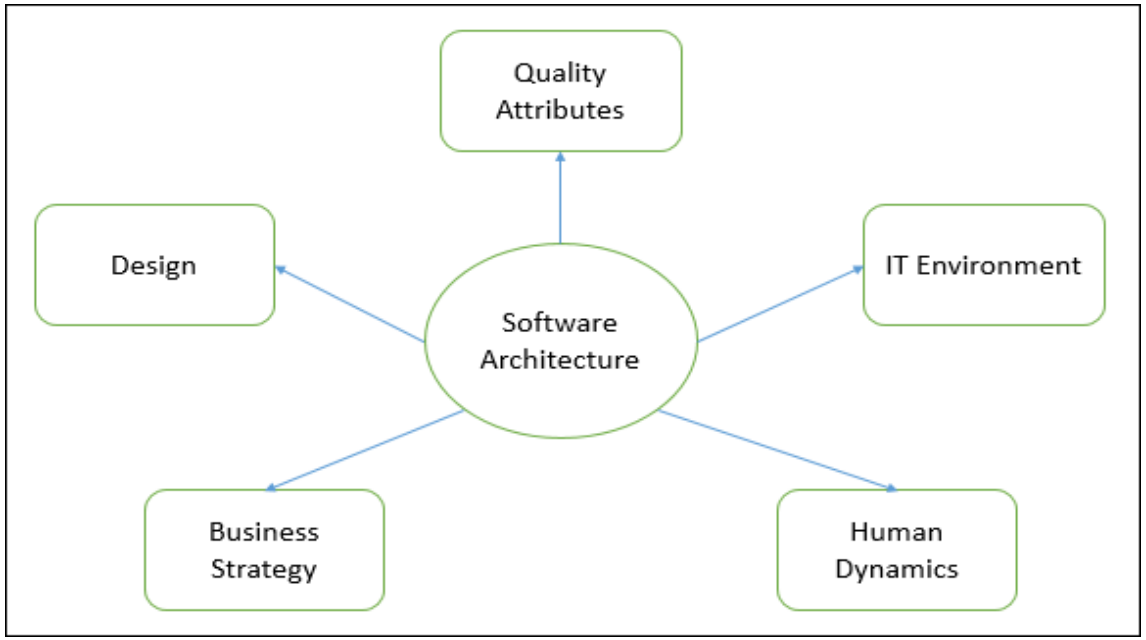
7.Coincidental Cohesion: A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely.

- Differentiate between Coupling and Cohesion

Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative **independence** between the modules.	Cohesion shows the module's relative **functional** strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

Software Architecture Design

- The architecture of a system describes its major components, their relationships (structures), and how they interact with each other.
- Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



- Software Architecture and Design divided into two distinct phases: Software Architecture and Software Design.
- In **Architecture**, nonfunctional decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished.

Software Architecture

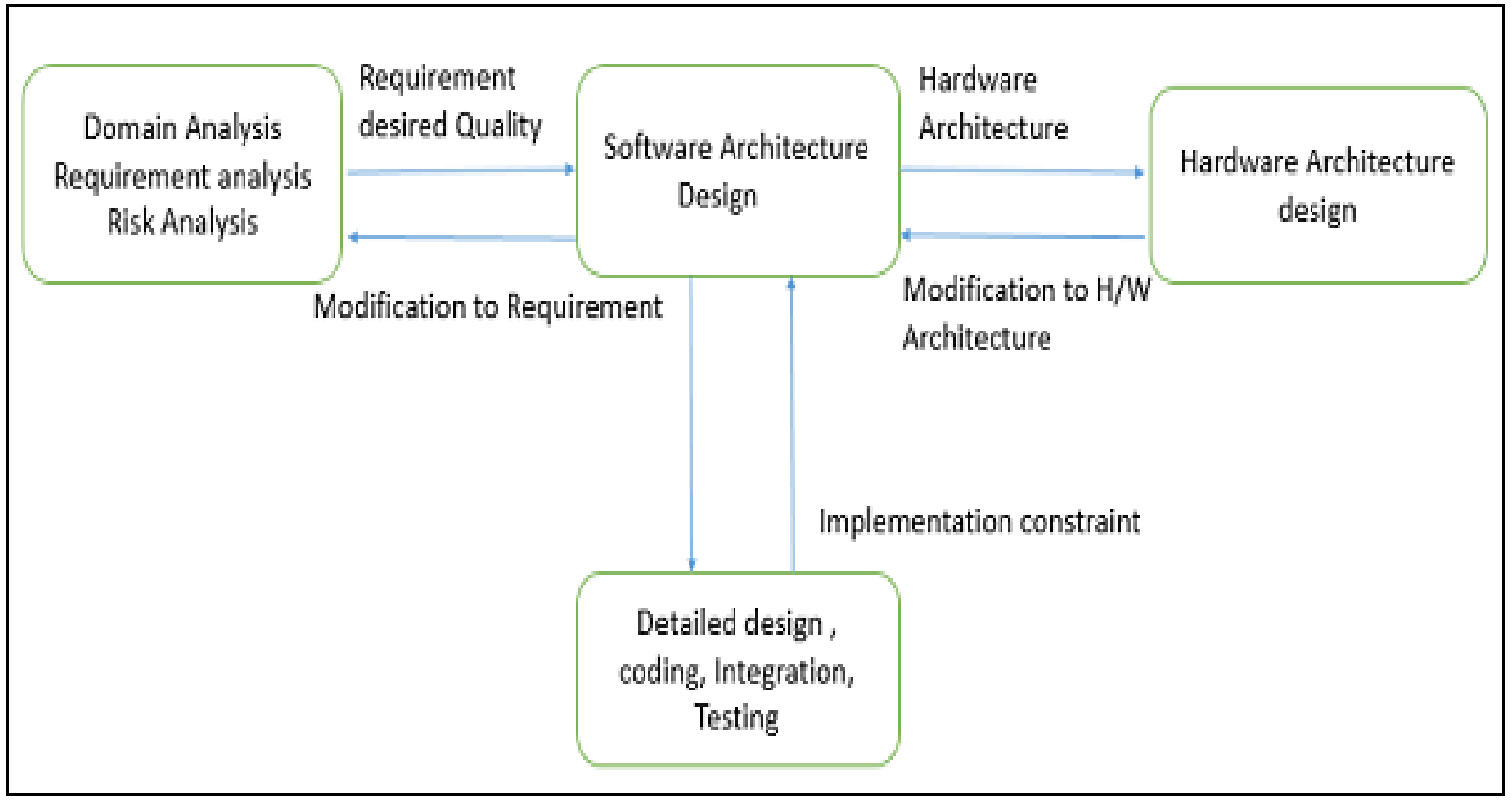
Architecture serves as a **blueprint for a system**. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.

- Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of –
  - Selection of structural elements and their interfaces by which the system is composed.
  - Behavior as specified in collaborations among those elements.
  - Composition of these structural and behavioral elements into large subsystem.
  - Architectural decisions align with business objectives.
  - Architectural styles guide the organization.

Software Design

- Software design provides a **design plan** that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system. The objectives of having a design plan are as follows –
- To negotiate system requirements, and to set expectations with customers, marketing, and management personnel.
- Act as a blueprint during the development process.
- Guide the implementation tasks, including detailed design, coding, integration, and testing.
- It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.


Domain Analysis
Requirement analysis
RIsk Analysis
Requirement
desired Quality
Software Architecture
Design
Hardware
Architecture
Hardware Architecture
design
Modification to Requirement
Modification to H/W
Architecture
Implementation constraint
Detailed design ,
coding, Integration,
Testing

Goals of Architecture

- The primary goal of the architecture is to identify requirements that affect the structure of the application. A well-laid architecture reduces the business risks associated with building a technical solution and builds a bridge between business and technical requirements.

Some of the other goals are as follows –

- Expose the structure of the system, but hide its implementation details.
- Realize all the use-cases and scenarios.
- Try to address the requirements of various stakeholders.
- Handle both functional and quality requirements.
- Reduce the goal of ownership and improve the organization's market position.
- Improve quality and functionality offered by the system.
- Improve external confidence in either the organization or system.

Limitations

Software architecture is still an emerging discipline within software engineering. It has the following limitations –

- Lack of tools and standardized ways to represent architecture.
- Lack of analysis methods to predict whether architecture will result in an implementation that meets the requirements.
- Lack of awareness of the importance of architectural design to software development.
- Lack of understanding of the role of software architect and poor communication among stakeholders.
- Lack of understanding of the design process, design experience and evaluation of design.

Quality Attributes

- Quality is a measure of excellence or the state of being free from deficiencies or defects. Quality attributes are the system properties that are separate from the functionality of the system.
- Implementing quality attributes makes it easier to differentiate a good system from a bad one. Attributes are overall factors that affect runtime behavior, system design, and user experience.

They can be classified as –

Static Quality Attributes

- Reflect the structure of a system and organization, directly related to architecture, design, and source code. They are invisible to end-user, but affect the development and maintenance cost, e.g.: modularity, testability, maintainability, etc.

Dynamic Quality Attributes

- Reflect the behavior of the system during its execution. They are directly related to system's architecture, design, source code, configuration, deployment parameters, environment, and platform.
- They are visible to the end-user and exist at runtime, e.g. throughput, robustness, scalability, etc.

Quality Scenarios

- Quality scenarios specify how to prevent a fault from becoming a failure. They can be divided into six parts based on their attribute specifications –
- **Source** – An internal or external entity such as people, hardware, software, or physical infrastructure that generate the stimulus.
- **Stimulus** – A condition that needs to be considered when it arrives on a system.
- **Environment** – The stimulus occurs within certain conditions.
- **Artifact** – A whole system or some part of it such as processors, communication channels, persistent storage, processes etc.
- **Response** – An activity undertaken after the arrival of stimulus such as detect faults, recover from fault, disable event source etc.
- **Response measure** – Should measure the occurred responses so that the requirements can be tested.

Architectural Style

- The **architectural style**, also called as **architectural pattern**, is a set of principles which shapes an application. It defines an abstract framework for a family of system in terms of the pattern of structural organization.

The architectural style is responsible to –

- Provide a lexicon of components and connectors with rules on how they can be combined.
- Improve partitioning and allow the reuse of design by giving solutions to frequently occurring problems.
- Describe a particular way to configure a collection of components (a module with well-defined interfaces, reusable, and replaceable) and connectors (communication link between modules).

- The software that is built for computer-based systems exhibit one of many architectural styles. Each style describes a system category that encompasses –
- A set of component types which perform a required function by the system.
- A set of connectors (subroutine call, remote procedure call, data stream, and socket) that enable communication, coordination, and cooperation among different components.
- Semantic constraints which define how components can be integrated to form the system.
- A topological layout of the components indicating their runtime interrelationships.

Types of Architecture

- There are four types of architecture from the viewpoint of an enterprise and collectively, these architectures are referred to as **enterprise architecture**.
- **Business architecture** – Defines the strategy of business, governance, organization, and key business processes within an enterprise and focuses on the analysis and design of business processes.
- **Application (software) architecture** – Serves as the blueprint for individual application systems, their interactions, and their relationships to the business processes of the organization.

- **Information architecture** – Defines the logical and physical data assets and data management resources.
- **Information technology (IT) architecture** – Defines the hardware and software building blocks that make up the overall information system of the organization.

Architecture Design Process

- The architecture design process focuses on the decomposition of a system into different components and their interactions to satisfy functional and nonfunctional requirements. The key inputs to software architecture design are –
- The requirements produced by the analysis tasks.

- The hardware architecture (the software architect in turn provides requirements to the system architect, who configures the hardware architecture).
- The result or output of the architecture design process is an **architectural description**. The basic architecture design process is composed of the following steps –

Understand the Problem

- This is the most crucial step because it affects the quality of the design that follows.
- Without a clear understanding of the problem, it is not possible to create an effective solution.

- Many software projects and products are considered failures because they did not actually solve a valid business problem or have a recognizable return on investment (ROI).

Identify Design Elements and their Relationships

- In this phase, build a baseline for defining the boundaries and context of the system.
- Decomposition of the system into its main components based on functional requirements. The decomposition can be modeled using a design structure matrix (DSM), which shows the dependencies between design elements without specifying the granularity of the elements.
- In this step, the first validation of the architecture is done by describing a number of system instances and this step is referred as functionality based architectural design.

Evaluate the Architecture Design

- Each quality attribute is given an estimate so in order to gather qualitative measures or quantitative data, the design is evaluated.
- It involves evaluating the architecture for conformance to architectural quality attributes requirements.
- If all estimated quality attributes are as per the required standard, the architectural design process is finished.
- If not, the third phase of software architecture design is entered: architecture transformation. If the observed quality attribute does not meet its requirements, then a new design must be created.

Transform the Architecture Design

- This step is performed after an evaluation of the architectural design. The architectural design must be changed until it completely satisfies the quality attribute requirements.
- It is concerned with selecting design solutions to improve the quality attributes while preserving the domain functionality.
- A design is transformed by applying design operators, styles, or patterns. For transformation, take the existing design and apply design operator such as decomposition, replication, compression, abstraction, and resource sharing.
- The design is again evaluated and the same process is repeated multiple times if necessary and even performed recursively.
- The transformations (i.e. quality attribute optimizing solutions) generally improve one or some quality attributes while they affect others negatively

Key Architecture Principles

Following are the key principles to be considered while designing an architecture –

- Build to Change Instead of Building to Last
- Consider how the application may need to change over time to address new requirements and challenges, and build in the flexibility to support this.
- Reduce Risk and Model to Analyze
- Use design tools, visualizations, modeling systems such as UML to capture requirements and design decisions. The impacts can also be analyzed. Do not formalize the model to the extent that it suppresses the capability to iterate and adapt the design easily.
- Use Models and Visualizations as a Communication and Collaboration Tool
- Efficient communication of the design, the decisions, and ongoing changes to the design is critical to good architecture.
- Identify and understand key engineering decisions and areas where mistakes are most often made.

- Use an Incremental and Iterative Approach
- Start with baseline architecture and Iteratively add details to the design over multiple passes to get the big or right picture and then focus on the details.

Key Design Principles

Following are the design principles to be considered for minimizing cost, maintenance requirements, and maximizing extendibility, usability of architecture

Separation of Concerns

- Divide the components of system into specific features so that there is no overlapping among the components functionality. This will provide high cohesion and low coupling. This approach avoids the interdependency among components of system which helps in maintaining the system easy.

Single Responsibility Principle

- Each and every module of a system should have one specific responsibility, which helps the user to clearly understand the system. It should also help with integration of the component with other components.

Principle of Least Knowledge

- Any component or object should not have the knowledge about internal details of other components. This approach avoids interdependency and helps maintainability.

Minimize Large Design Upfront

- Minimize large design upfront if the requirements of an application are unclear. If there is a possibility of modifying requirements, then avoid making a large design for whole system.

Do not Repeat the Functionality

- Do not repeat functionality specifies that functionality of components should not to be repeated and hence a piece of code should be implemented in one component only. Duplication of functionality within an application can make it difficult to implement changes, decrease clarity, and introduce potential inconsistencies.

Prefer Composition over Inheritance while Reusing the Functionality

- Inheritance creates dependency between children and parent classes and hence it blocks the free use of the child classes. In contrast, the composition provides a great level of freedom and reduces the inheritance hierarchies.

Identify Components and Group them in Logical Layers

- Identity components and the area of concern that are needed in system to satisfy the requirements. Then group these related components in a logical layer, which will help the user to understand the structure of the system at a high level. Avoid mixing components of different type of concerns in same layer.

Define the Communication Protocol between Layers

- Understand how components will communicate with each other which requires a complete knowledge of deployment scenarios and the production environment.

Define Data Format for a Layer

- Various components will interact with each other through data format. Do not mix the data formats so that applications are easy to implement, extend, and maintain.

System Service Components should be Abstract

- Code related to security, communications, or system services like logging, profiling, and configuration should be abstracted in the separate components. Do not mix this code with business logic, as it is easy to extend design and maintain it.

Design Exceptions and Exception Handling Mechanism

- Defining exceptions in advance, helps the components to manage errors or unwanted situation in an elegant manner. The exception management will be same throughout the system.

Naming Conventions

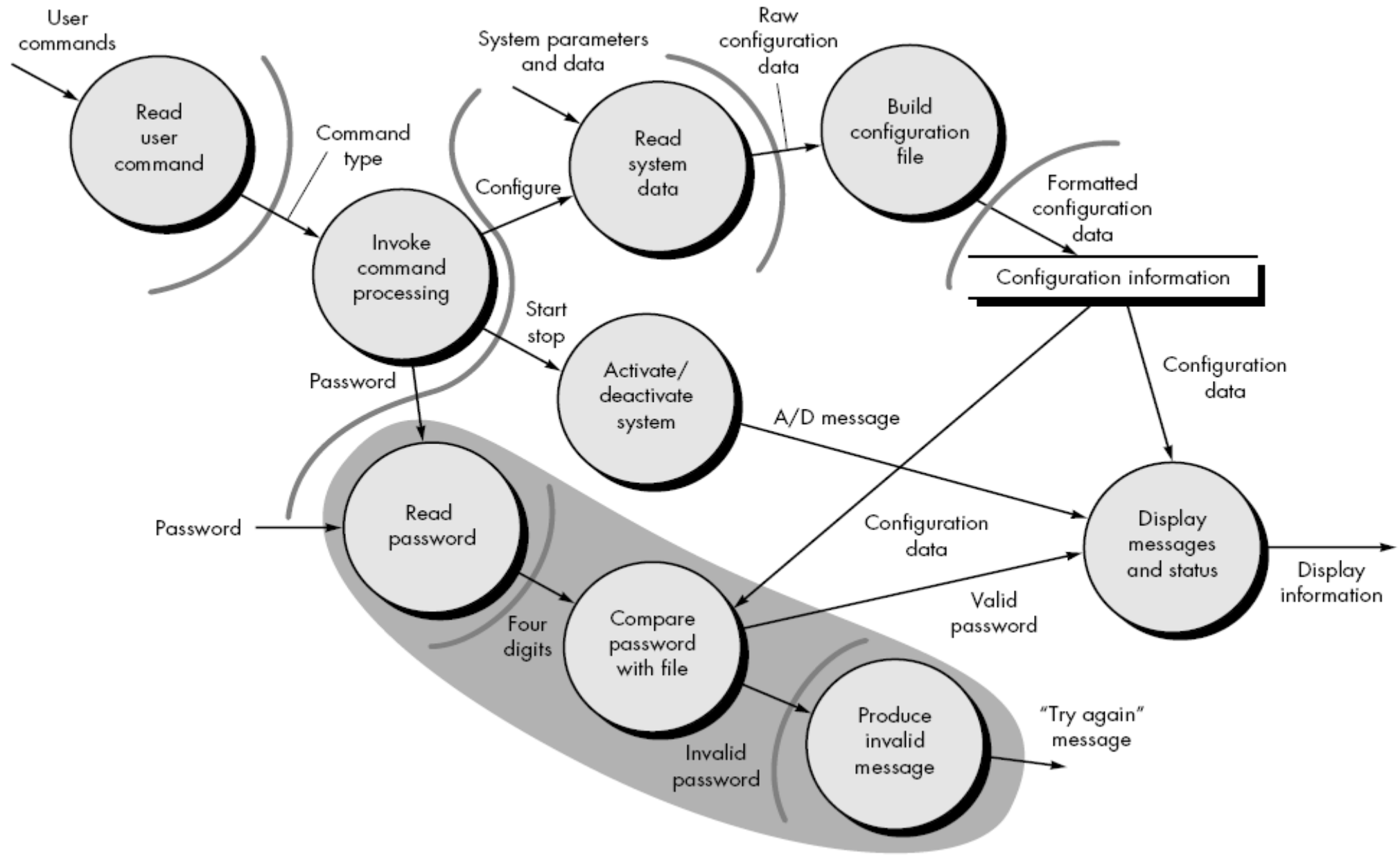
- Naming conventions should be defined in advance. They provide a consistent model that helps the users to understand the system easily. It is easier for team members to validate code written by others, and hence will increase the maintainability.

Transaction Mapping

In many software applications, a single data item triggers one or a number of information flows that effect a function implied by the triggering data item. The data item, called a transaction, and its corresponding flow characteristics . In this section we consider design steps used to treat transaction flow.

An Example

Transaction mapping will be illustrated by considering the user interaction subsystem of the SafeHome software.


User commands
Read user command
Command type
Invoke command processing
Configure
System parameters and data
Read system data
Raw configuration data
Build configuration file
Formatted configuration data
Configuration information
Start stop
Activate/ deactivate system
A/D message
Password
Password
Read password
Four digits
Compare password with file
Invalid password
Produce invalid message
"Try again" message
Configuration data
Valid password
Configuration data
Display messages and status
Display information

- As shown in the figure, user commands flows into the system and results in additional information flow along one of three action paths. A single data item, command type, causes the data flow to fan outward from a hub. Therefore, the overall data flow characteristic is transaction oriented.

It should be noted that information flow along two of the three action paths accommodate additional incoming flow (e.g., system parameters and data are input on the "configure" action path). Each action path flows into a single transform, display messages and status.

Design Steps

- The design steps for transaction mapping are similar and in some cases identical to steps for transform mapping . A major difference lies in the mapping of DFD to software structure.

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

Steps 1, 2, and 3 are identical to corresponding steps in transform mapping.

Step 4.

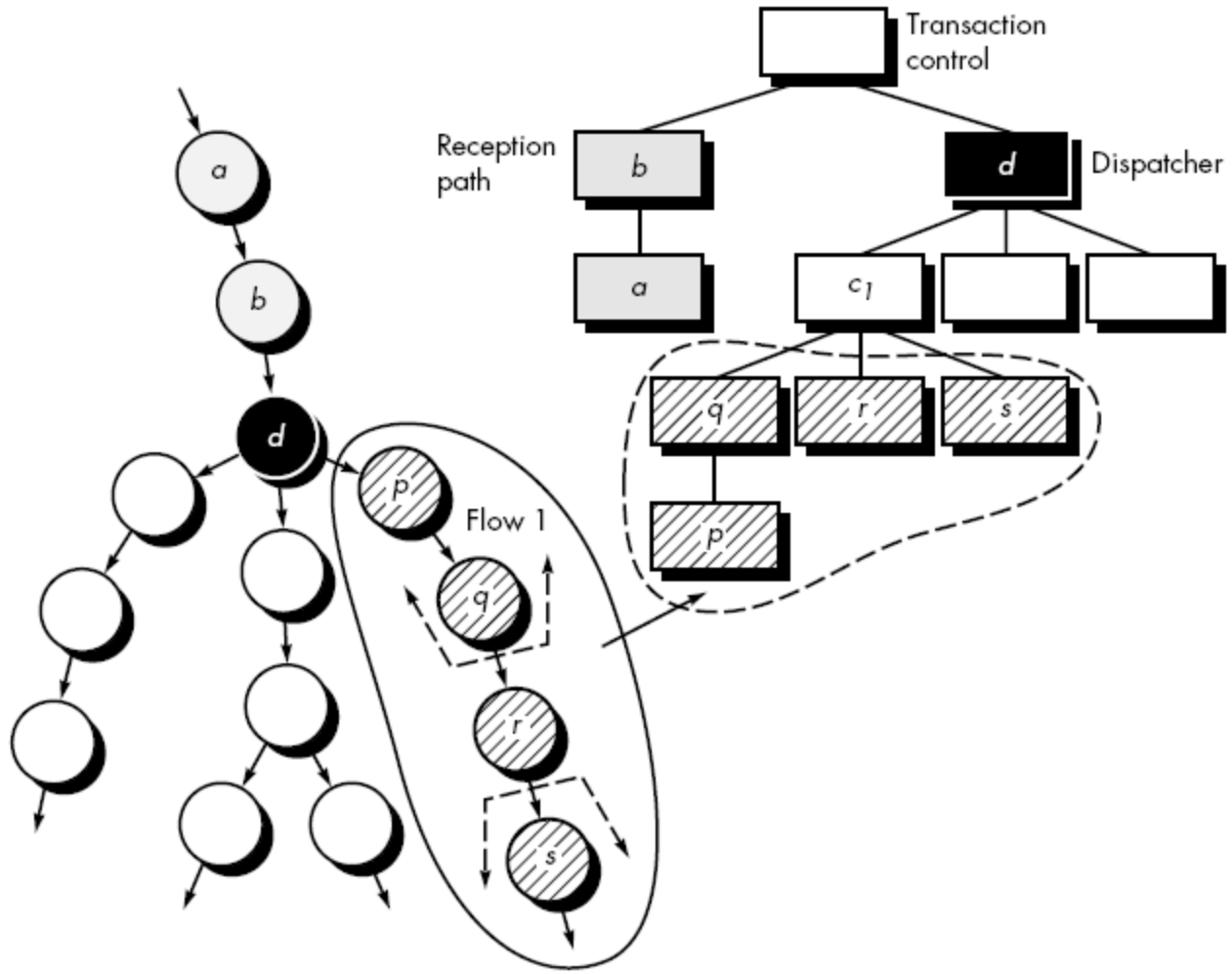
Identify the transaction center and the flow characteristics along each of the action paths. The location of the transaction center can be immediately discerned from the DFD.

The incoming path (i.e., the flow path along which a transaction is received) and all action paths must also be isolated. Boundaries that define a reception path and action paths are also shown in the figure.

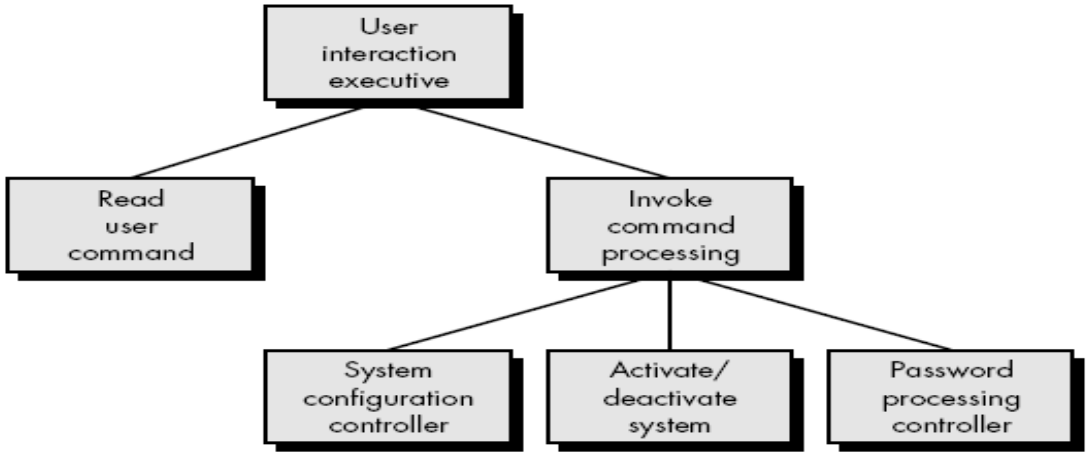
Each action path must be evaluated for its individual flow characteristic.

For example, the "password" path has transform characteristics. Incoming, transform, and outgoing flow are indicated with boundaries.

- **Step 5.** Map the DFD in a program structure amenable to transaction processing.
- Transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch.
- Starting at the transaction center, bubbles along the incoming path are mapped into modules.
- The structure of the dispatch branch contains a dispatcher module that controls all subordinate action modules.
- Each action flow path of the DFD is mapped to a structure that corresponds to its specific flow characteristics. This process is illustrated schematically in figure below.


Transaction control
Reception path
b
d
Dispatcher
a
c_1
q
r
s
p
a
b
d
p
Flow 1
q
r
s

- Considering the user interaction subsystem data flow, first-level factoring for step 5 is shown in below figure.



The bubbles read user command and activate/deactivate system map directly into the architecture without the need for intermediate control modules. The transaction center, invoke command processing, maps directly into a dispatcher module of the same name. Controllers for system configuration and password processing are created as illustrated in figure.

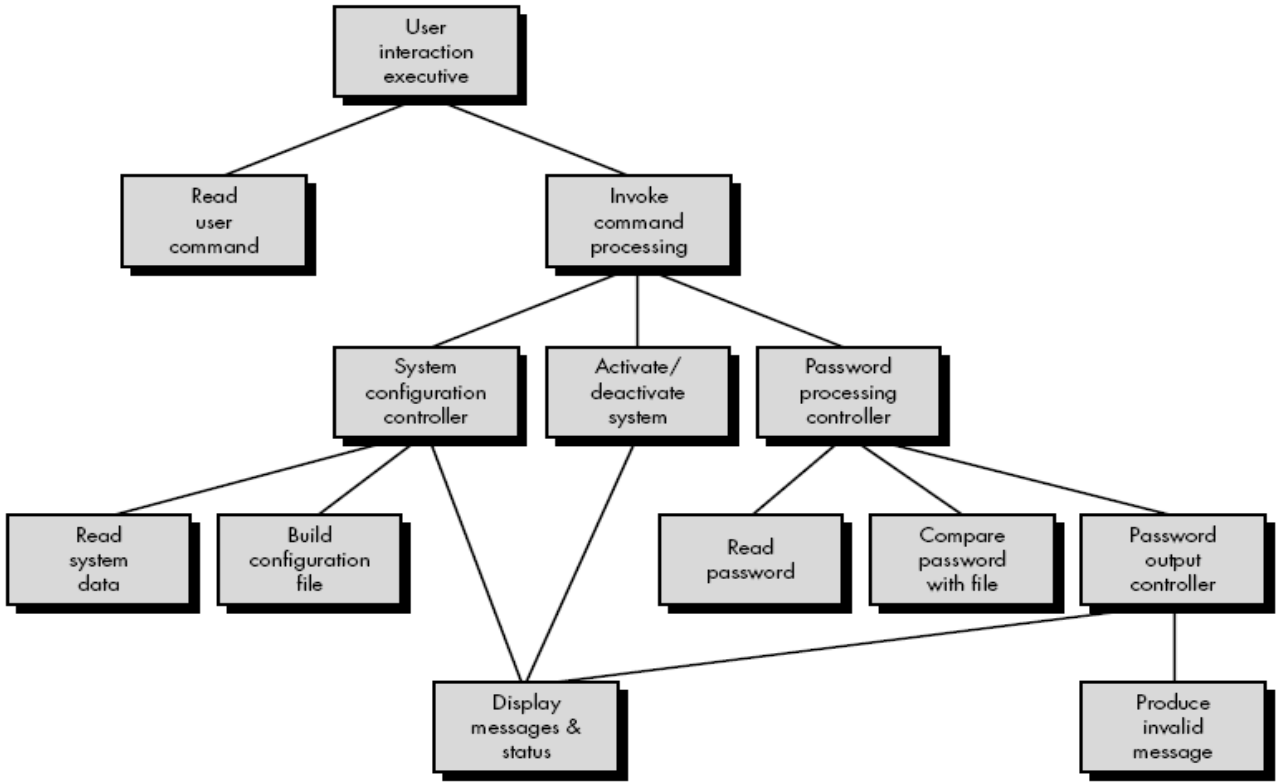
- Step 6. Factor and refine the transaction structure and the structure of each action path.
- Each action path of the data flow diagram has its own information flow characteristics.

As an example, consider the password processing information flow shown (inside shaded area) in figure

The flow exhibits classic transform characteristics. A password is input (incoming flow) and transmitted to a transform center where it is compared against stored passwords.

An alarm and warning message (outgoing flow) are produced (if a match is not obtained).

The "configure" path is drawn similarly using the transform mapping. The resultant software architecture is shown in the figure below



Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. This step for transaction mapping is identical to the corresponding step for transform mapping. In both design approaches, criteria such as module independence, practicality (efficacy of implementation and test), and maintainability must be carefully considered as structural modifications are proposed.

Transform Mapping

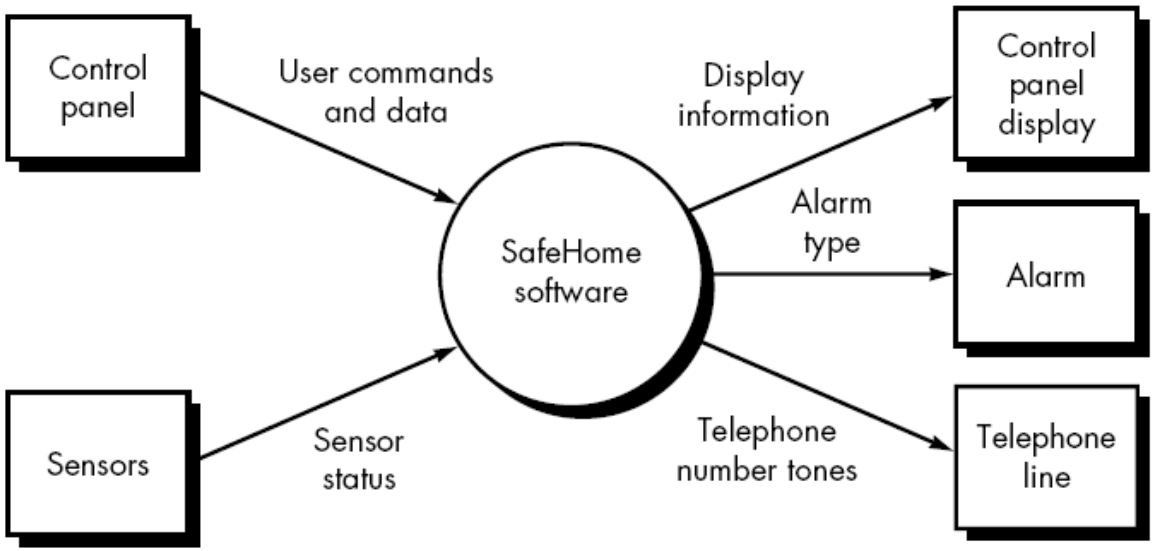
Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style.

example system—a portion of the SafeHome security software.

An Example

This product monitors the real world and reacts to changes that it encounters.

It also interacts with a user through a series of typed inputs and alphanumeric displays. The level 0 data flow diagram for SafeHome, is shown in figure



During requirements analysis, more detailed flow models would be created for SafeHome. In addition, control and process specifications, a data dictionary, and various behavioral models would also be created.

Design Steps

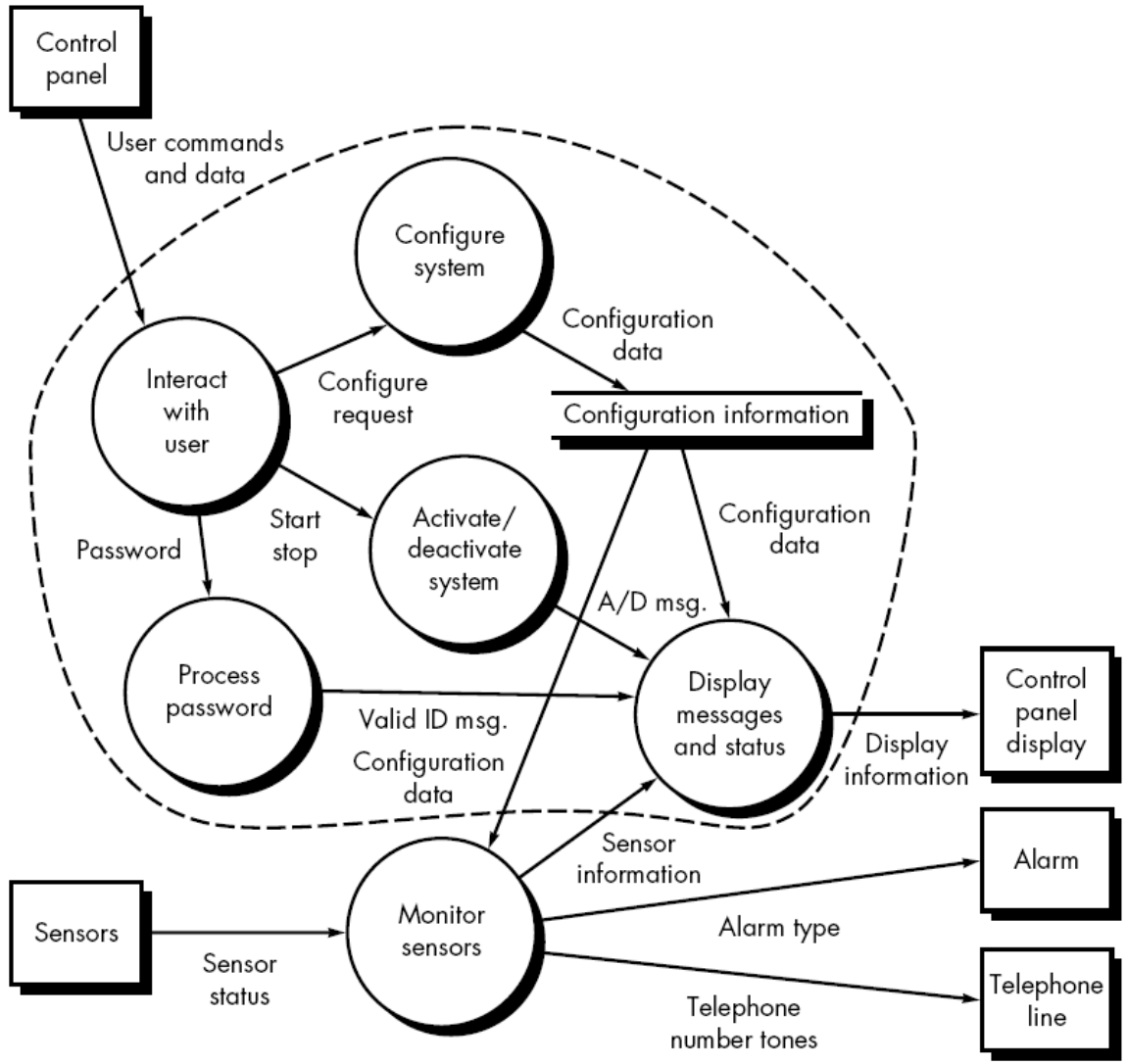
The preceding example will be used to illustrate each step in transform mapping. The steps begin with a re-evaluation of work done during requirements analysis and then move to the design of the software architecture.

Step 1. Review the fundamental system model.

The fundamental system model encompasses the level 0 DFD and supporting information.

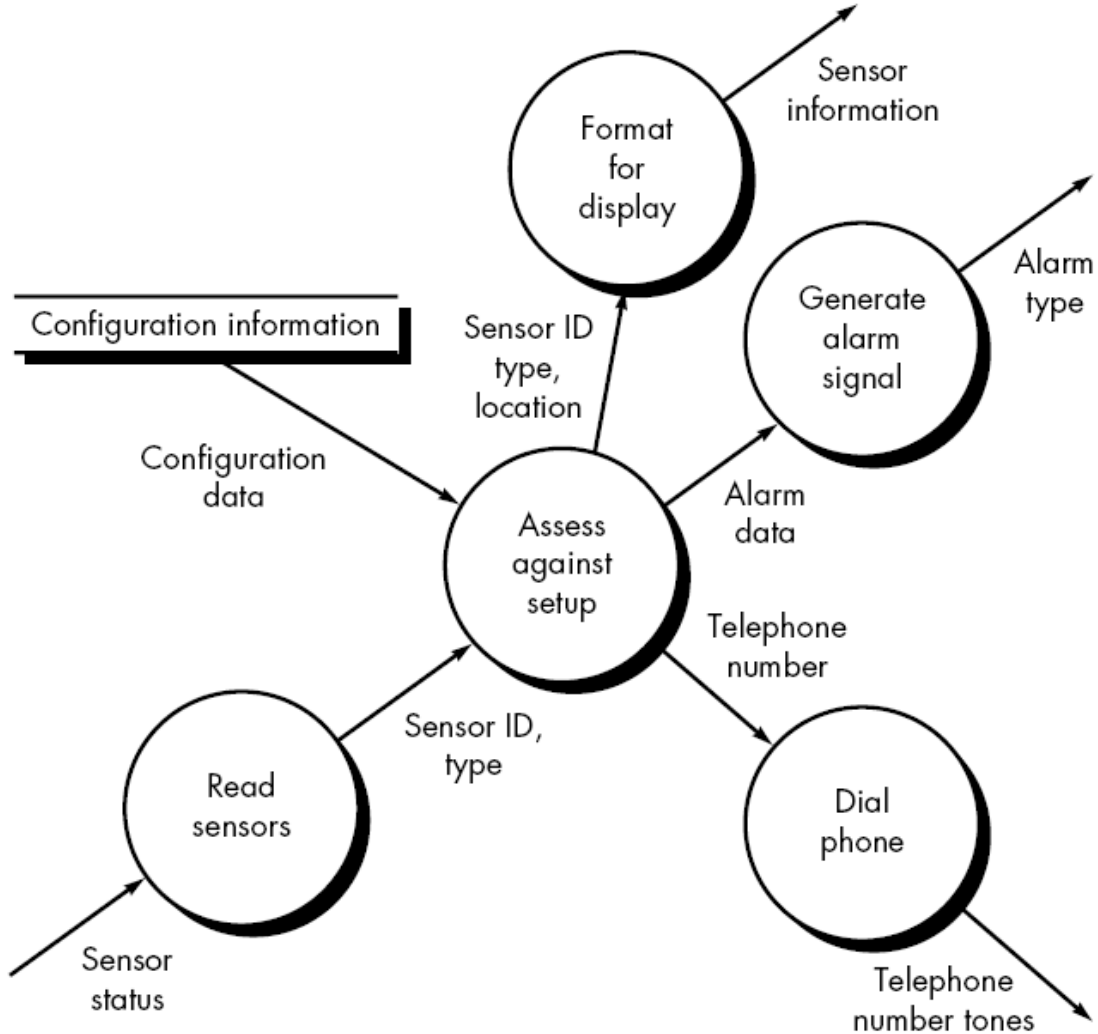
The design step begins with an evaluation of both the System Specification and the Software Requirements Specification.

Both documents describe information flow and structure at the software interface. Figure 1 and 2 depict level 0 and level 1 data flow for the SafeHome software.


Control panel
User commands and data
Configure system
Configuration data
Interact with user
Configure request
Configuration information
Start stop
Activate/ deactivate system
Configuration data
Password
A/D msg.
Process password
Display messages and status
Control panel display
Valid ID msg.
Configuration data
Display information
Sensor information
Alarm
Sensors
Monitor sensors
Alarm type
Sensor status
Telephone line
Telephone number tones

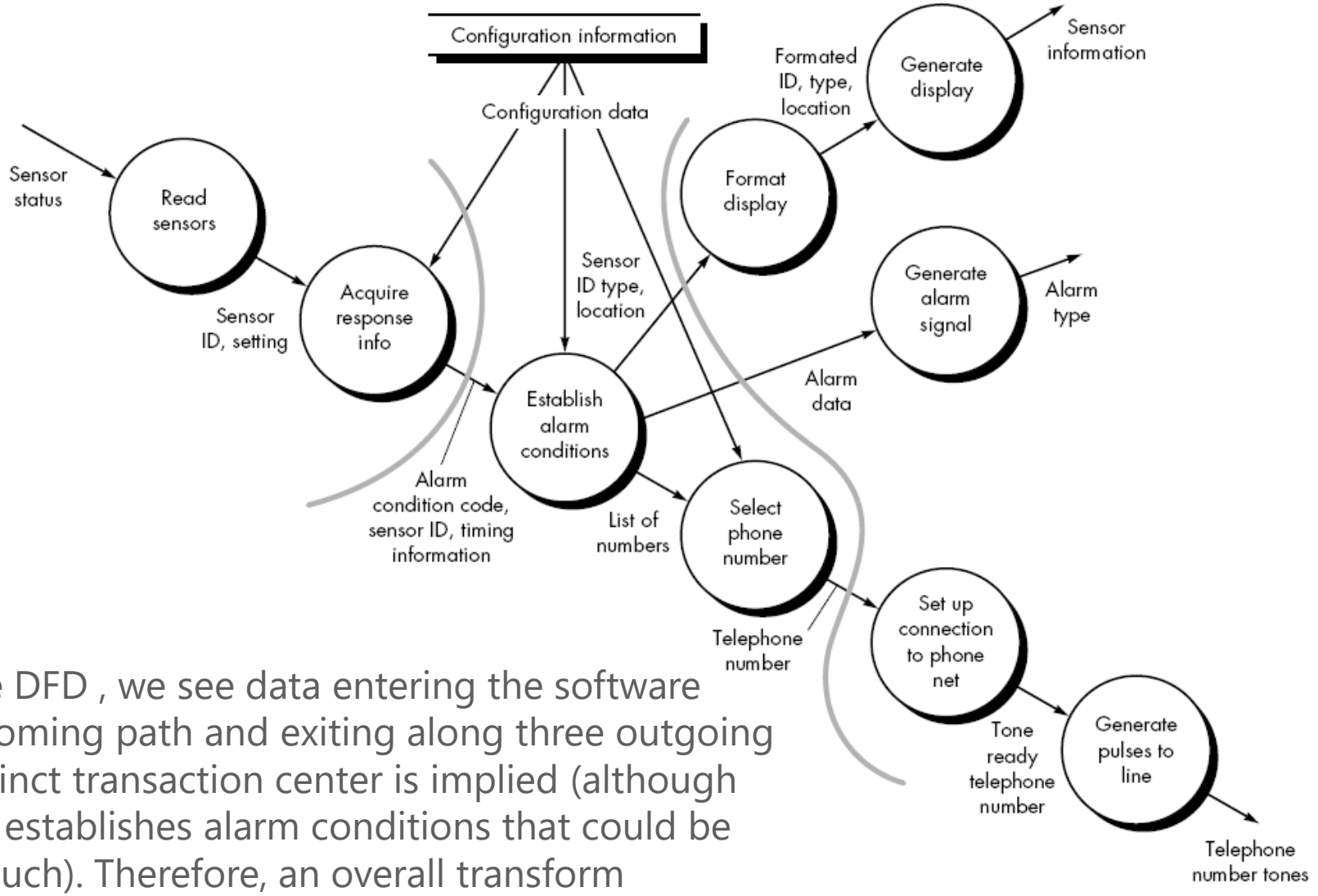
Step 2. Review and refine data flow diagrams for the software.

- Information obtained from analysis models contained in the Software Requirements Specification is refined to produce greater detail.
- For example, the level 2 DFD for monitor sensors is examined, and a level 3 data flow diagram is derived .
- At level 3, each transform in the data flow diagram exhibits relatively high cohesion.
- That is, the process implied by a transform performs a single, distinct function that can be implemented in the SafeHome software.


Format for display
Sensor information
Configuration information
Sensor ID type, location
Generate alarm signal
Alarm type
Configuration data
Assess against setup
Alarm data
Telephone number
Read sensors
Sensor ID, type
Dial phone
Sensor status
Telephone number tones

Step 3. Determine whether the DFD has transform or transaction flow characteristics.

- In general, information flow within a system can always be represented as transform.
- In this step, the designer selects global flow characteristics based on the prevailing nature of the DFD.
- In addition, local regions of transform or transaction flow are isolated.
- These subflows can be used to refine program architecture derived from a global characteristic described previously.
- For now, we focus our attention only on the monitor sensors subsystem data flow depicted in figure.



Evaluating the DFD , we see data entering the software along one incoming path and exiting along three outgoing paths. No distinct transaction center is implied (although the transform establishes alarm conditions that could be perceived as such). Therefore, an overall transform characteristic will be assumed for information flow.

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries.

- In the preceding section incoming flow was described as a path in which information is converted from external to internal form;
- outgoing flow converts from internal to external form. Incoming and outgoing flow boundaries are open to interpretation.
- That is, different designers may select slightly different points in the flow as boundary locations.

- **Step 5. Perform "first-level factoring." Program structure represents a top-down distribution of control.**
- Factoring results in a program structure in which top-level modules perform decision making and low-level modules perform most input, computation, and output work. Middle-level modules perform some control and do moderate amounts of work.

- **Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture.**
- Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure.

- **Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.**
- A first-iteration architecture can always be refined by applying concepts of module independence . Modules are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

Refactoring of designs

Refactoring is the process of restructuring code, while not changing its original functionality. The goal of refactoring is to improve internal code by making many small changes without altering the code's external behavior.

software developers refactor code to improve the design, structure and implementation of software. Refactoring improves code readability and reduces complexities.

These changes preserve the software's original behavior and do not modify its behavior.

Purpose of refactoring

Refactoring improves code by making it:

- More efficient by addressing dependencies and complexities.
- More maintainable or reusable by increasing efficiency and readability.
- Cleaner so it is easier to read and understand.
- Easier for software developers to find and fix bugs or vulnerabilities in the code.

- Code modification is done without changing any functions of the program itself. Many basic editing environments support simple refactorings like renaming a function or variable across an entire code base.
- Refactoring can be performed after a product has been deployed, before adding updates and new features to existing code, or as a part of day-to-day programming.
- A better time to perform refactoring, is before adding updates or new features to existing code.

Benefits of refactoring

- Makes the code easier to understand and read because the goal is to simplify code and reduce complexities.
- Improves maintainability and makes it easier to spot bugs or make further changes.
- Encourages a more in-depth understanding of code. Developers have to think further about how their code will mix with code already in the code base.
- Focus remains only on functionality. Not changing the code's original functionality ensures the original project does not lose scope.

challenges of refactoring

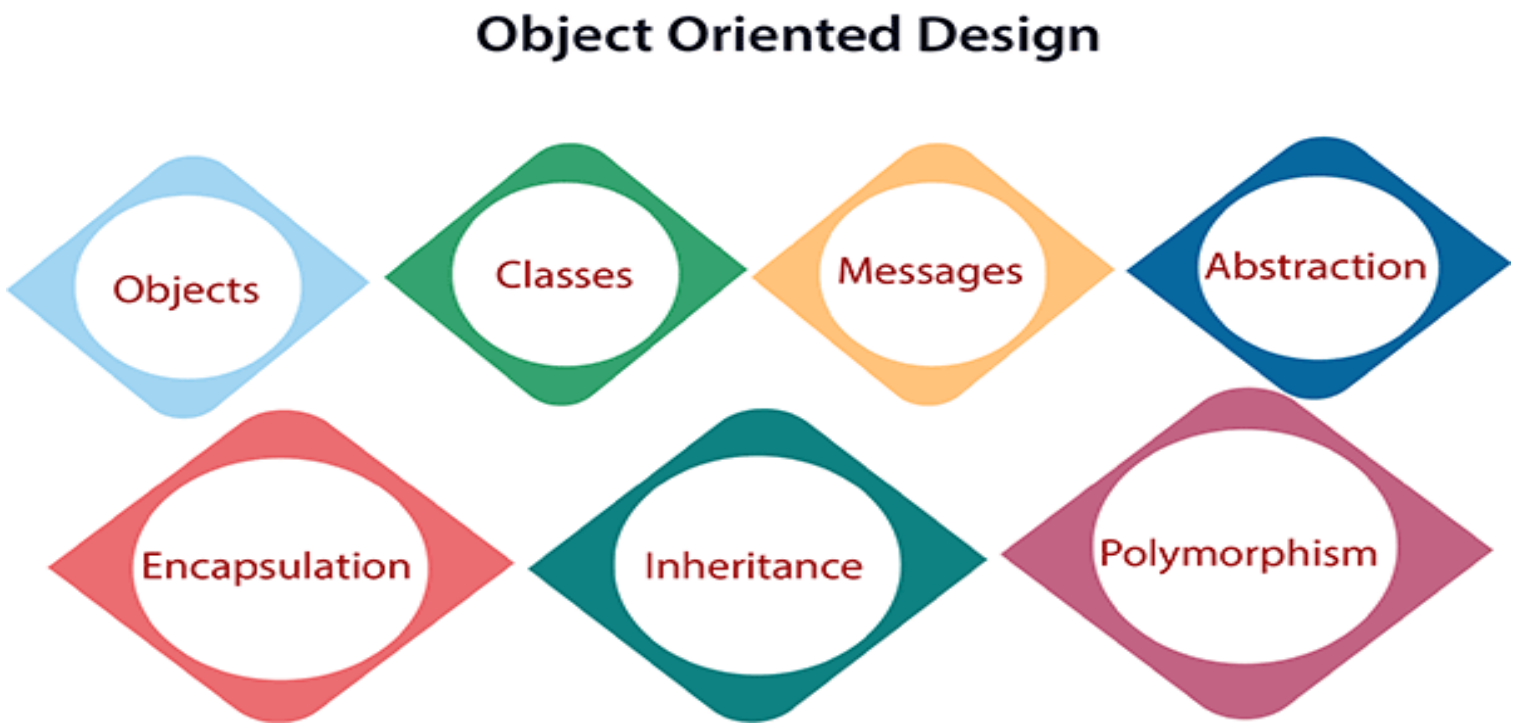
- The process will take extra time if a development team is in a rush and refactoring is not planned for.
- Without clear objectives, refactoring can lead to delays and extra work.
- Refactoring cannot address software flaws by itself, as it is made to clean code and make it less complex.

Code refactoring best practices

- **Plan for refactoring.** It may be difficult to make time for the time-consuming practice otherwise.
- **Refactor first.** Developers should do this before adding updates or new features to existing code to reduce technical debt.
- **Refactor in small steps.** This gives developers feedback early in the process so they can find possible bugs, as well as include business requests.
- **Set clear objectives.** Developers should determine the project scope and goals early in the code refactoring process. This helps to avoid delays and extra work, as refactoring is meant to be a form of housekeeping, not an opportunity to changes functions or features.
- **Test often.** This helps to ensure refactored changes do not introduce new bugs.
- **Automate wherever possible.** Automation tools make refactoring easier and faster, thus, improving efficiency.
- **Fix software defects separately.** Refactoring is not meant to address software flaws. Troubleshooting and debugging should be done separately.
- **Understand the code.** Review the code to understand its processes, methods, objects, variables and other elements.
- **Refactor, patch and update regularly.** Refactoring generates the most return on investment when it can address a significant issue without taking too much time and effort.
- **Focus on code deduplication.** Duplication adds complexities to code, expanding the software's footprint and wasting system resources.

Object-Oriented Design

- In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities). The state is distributed among the objects, and each object handles its state data.



1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.
2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.
3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.

3.Abstraction In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.

5.Encapsulation: Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.

6.Inheritance: OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.

7. Polymorphism: OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

User Interface Design

- The visual part of a computer application or operating system through which a client interacts with a computer or software. It determines how commands are given to the computer or the program and how data is displayed on the screen.

Types of User Interface

- There are two main types of User Interface:
- Text-Based User Interface or Command Line Interface
- Graphical User Interface (GUI)

Text-Based User Interface: This method relies primarily on the keyboard. A typical example of this is UNIX.

Advantages

- Many and easier to customizations options.
- Typically capable of more important tasks.
- Disadvantages
- Relies heavily on recall rather than recognition.
- Navigation is often more difficult.

Graphical User Interface (GUI): GUI relies much more heavily on the mouse. A typical example of this type of interface is any versions of the Windows operating systems.

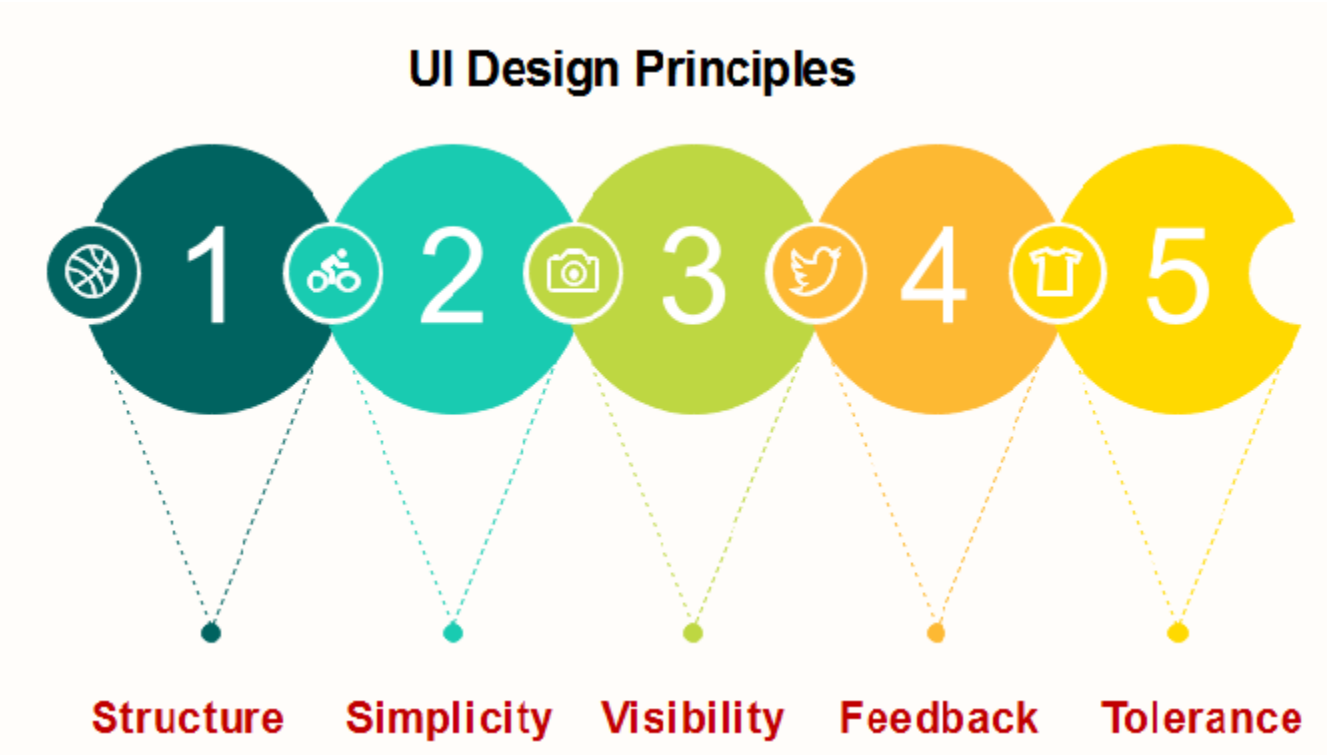
- GUI Characteristics

Characteristics	Descriptions
Windows	Multiple windows allow different information to be displayed simultaneously on the user's screen.
Icons	Icons different types of information. On some systems, icons represent files. On other icons describes processes.
Menus	Commands are selected from a menu rather than typed in a command language.
Pointing	A pointing device such as a mouse is used for selecting choices from a menu or indicating items of interests in a window.
Graphics	Graphics elements can be mixed with text or the same display.

Advantages

- Less expert knowledge is required to use it.
- Easier to Navigate and can look through folders quickly in a guess and check manner.
- The user may switch quickly from one task to another and can interact with several different applications.
- Disadvantages
- Typically decreased options.
- Usually less customizable. Not easy to use one button for tons of different variations.

UI Design Principles



- **Structure:** Design should organize the user interface purposefully, in the meaningful and usual based on precise, consistent models that are apparent and recognizable to users, putting related things together and separating unrelated things, differentiating dissimilar things and making similar things resemble one another. The structure principle is concerned with overall user interface architecture.
- **Simplicity:** The design should make the simple, common task easy, communicating clearly and directly in the user's language, and providing good shortcuts that are meaningfully related to longer procedures.
- **Visibility:** The design should make all required options and materials for a given function visible without distracting the user with extraneous or redundant data.
- **Feedback:** The design should keep users informed of actions or interpretation, changes of state or condition, and bugs or exceptions that are relevant and of interest to the user through clear, concise, and unambiguous language familiar to users.
- **Tolerance:** The design should be flexible and tolerant, decreasing the cost of errors and misuse by allowing undoing and redoing while also preventing bugs wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions.