Design Concepts

Introduction: Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. Design principles establish an overriding philosophy that guides you in the design work you must perform. Design is pivotal to successful software engineering

The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight Software design changes continually as new methods, better analysis, and broader understanding evolve

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).

Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in following figure.

The requirements model, manifested by **scenario-based, class-based, flow-oriented, and behavioral elements,** feed the design task.

The **data/class design** transforms class models into design class realizations and the requisite data structures required to implement the software.

The **architectural design** defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer- based system—is derived from the requirements model.

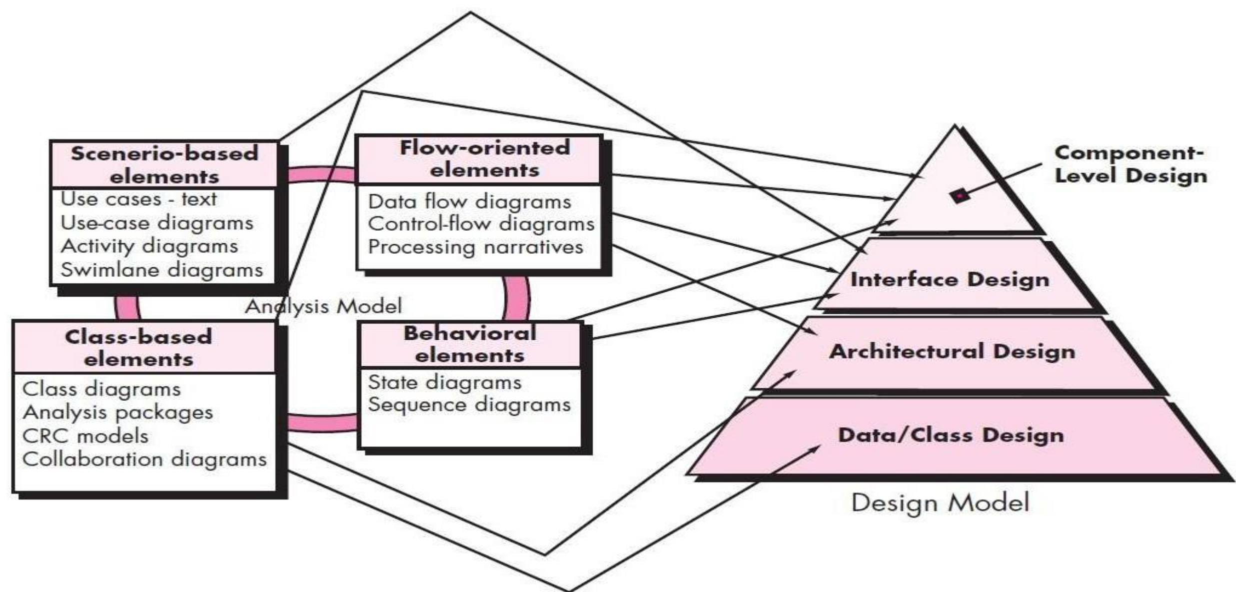


Fig : Translating the requirements model into the design model

The **interface design** describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The **component-level design** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

The importance of software design can be stated with a single word—

***quality*.** Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow.

THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "**blueprint**" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a **high level of abstraction**

Software Quality Guidelines and Attributes

McGlaughlin suggests **three** characteristics that serve as a guide for the evaluation of a good design:

- The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- 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. In order to evaluate the quality of a design representation, consider the following guidelines:

1. 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,2 thereby facilitating implementation and testing.

2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

3. A design should contain distinct representations of data, architecture, interfaces, and components.

4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

5. A design should lead to components that exhibit independent functional characteristics.

6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

8. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes. Hewlett-Packard developed a set of software quality attributes that has been given the acronym **FURPS—functionality, usability, reliability, performance, and supportability**. The **FURPS** quality attributes represent a target for all software design:

- ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system..
- ***Usability*** is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- ***Performance*** is measured by considering processing speed, response time,

resource consumption, throughput, and efficiency.

• ***Supportability*** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, ***maintainability***— and in addition, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can be localized.

The Evolution of Software Design

The evolution of software design is a continuing process that has now spanned almost six decades. Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top down manner. Procedural aspects of design definition evolved into a philosophy called ***structured programming***.

A number of design methods, growing out of the work just noted, are being applied throughout the industry. All of these methods have a number of common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you answer the following questions:

• What criteria can be used to partition software into individual components?

• How is function or data structure detail separated from a conceptual representation of the software?

• What uniform criteria define the technical quality of a software design?

The following brief overview of important software design concepts that span both traditional and object-oriented software development.

Abstraction

Abstraction is the act of representing essential features without including the background details or explanations. the *abstraction* is used to reduce complexity and allow efficient design and implementation of complex *software* systems. Many levels of abstraction can be posed. At the **highest level** of abstraction, a solution is stated in broad terms using the language of the problem environment. At **lower levels** of abstraction, a more detailed description of the solution is provided.

As different levels of abstraction are developed, you work to create both **procedural** and **data abstractions**.

A ***procedural abstraction*** refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed.

A ***data abstraction*** is a named collection of data that describes a data object.

Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

Shaw and Garlan 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, filters) and the manner in which those components are packaged and interact with one another.

- **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 one or more of a number of different models. ***Structural models:*** *R*epresent architecture as an organized collection of program components. ***Framework models:*** Increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

Dynamic models *:* Address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

Process models *:*Focus on the design of the business or technical process that the system must accommodate.

Functional models can be used to represent the functional hierarchy of a system.

A number of different ***architectural description languages*** **(ADLs)** have been developed to represent these models.

Patterns

Brad Appleton defines a ***design pattern*** in the following manner: "A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns"

A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of Concerns

Separation of concerns is a design concept that suggests that 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.

Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

8.3.5 Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called ***module.***

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

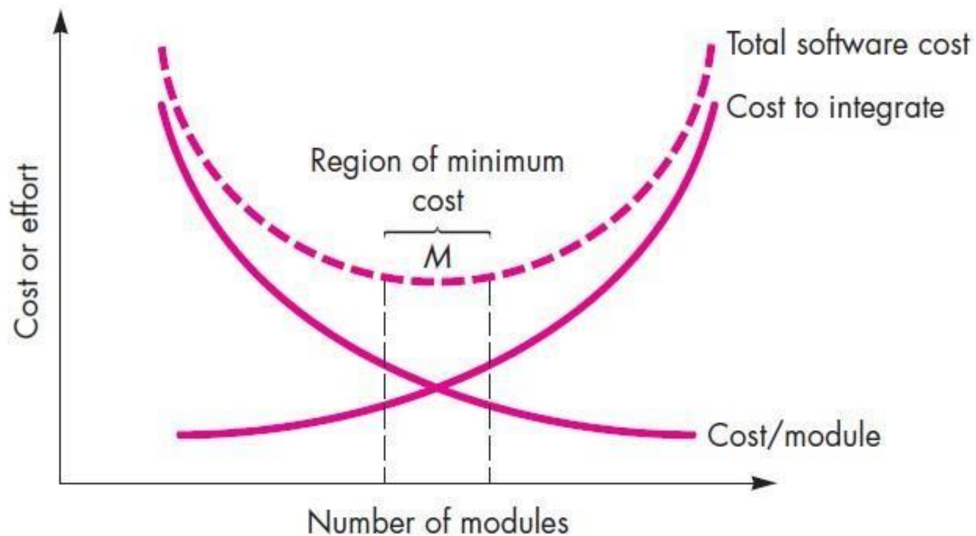


Fig : Modularity and software cost

Information Hiding

The principle of information hiding suggests that modules be "characterized by design decisions that hides from all others." In other words, modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with "**single minded**" function and an "aversion" to excessive interaction with other modules.

Independence is assessed using **two** qualitative criteria: **cohesion** and **coupling**. ***Cohesion*** is an indication of the relative functional strength

of a module. ***Coupling*** is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding concept. 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 do just one thing. Although you should always strive for **high cohesion** (i.e., single-mindedness).

Coupling is an indication of interconnection among modules in a software structure. 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. In software design, you should strive for the **lowest possible coupling**.

Refinement

Stepwise refinement is a **top-down** design strategy originally proposed by Niklaus Wirth. Refinement is actually a process of ***elaboration***. You begin with a statement of function that is defined at a high level of abstraction.

Abstraction and **refinement** are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses.

Aspects

An ***aspect*** is a representation of a crosscutting concern. A crosscutting concern is some characteristic of the system that applies across many different requirements.

Refactoring

An important design activity suggested for many agile methods, ***refactoring*** is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. **Fowler** 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.**"

Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.

Design Classes

The requirements model defines a set of analysis classes. Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. A set of ***design classes*** that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.

Five different types of design classes, each representing a different layer of the design architecture, can be developed:

- ***User interface classes*** define all abstractions that are necessary for human computer interaction (HCI). The design classes for the interface may be visual representations of the elements of the metaphor.
- ***Business domain classes*** are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- ***Process classes*** implement lower-level business abstractions required to fully manage the business domain classes.
- ***Persistent classes*** represent data stores (e.g., a database) that will persist beyond the execution of the software.
- ***System classes*** implement software management and control functions that enable the system to operate and communicate within its computing

environment and with the outside world.

Arlow and Neustadt suggest that each design class be reviewed to ensure that it is "**well- formed**." They define **four** characteristics of a well-formed design class:

- **Complete and sufficient.** A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.
- **Primitiveness.** Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.
- **High cohesion.** A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.
- **Low coupling.** Within the design model, it is necessary for design classes to collaborate with one another. If a design model is highly coupled, the system is difficult to implement, to test, and to maintain over time.

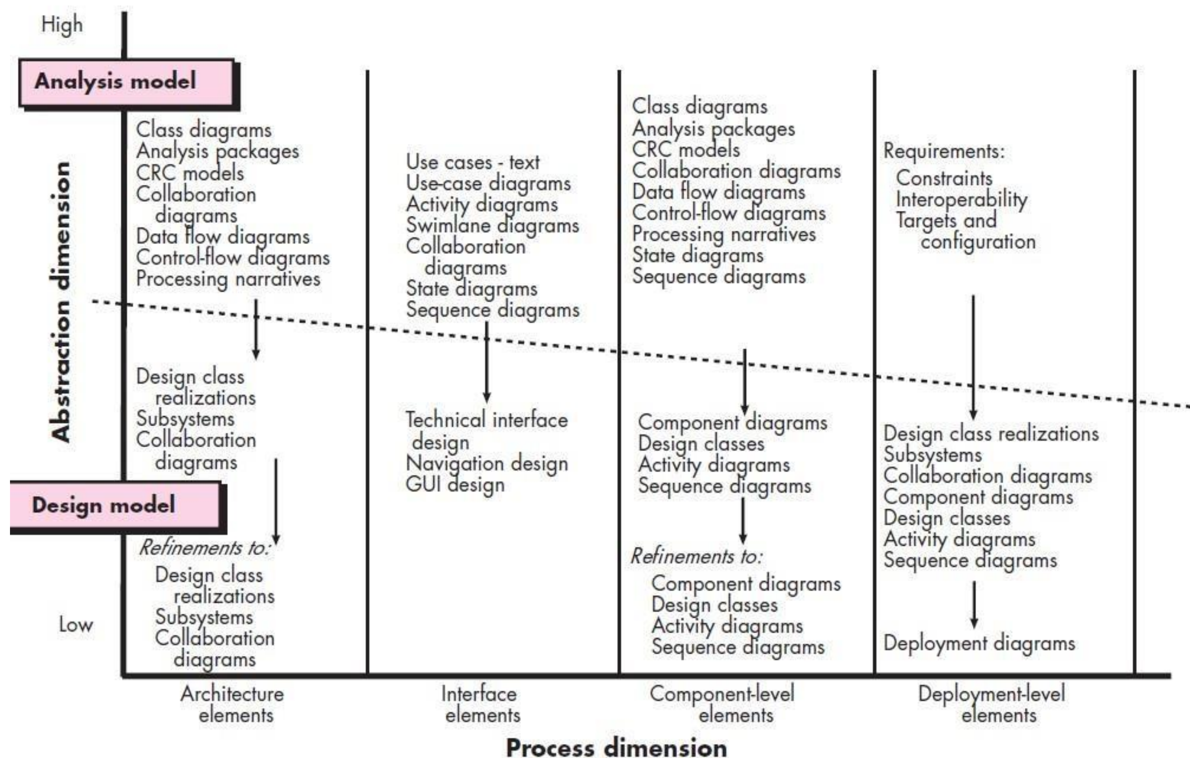
THE DESIGN MODEL

The design model can be viewed in **two** different dimensions. The ***process dimension*** indicates the evolution of the design model as design tasks are executed as part of the software process. The ***abstraction dimension*** represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. The design model has **four** major elements: data, architecture, components, and interface.

3.4.1. Data Design Elements

Data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. The structure of data has always been an important part of software design. At the program **component level**, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high- quality applications. At the **application level**, the translation of a data model into a database is pivotal to achieving the business objectives of a system. At the **business level**, the collection of

information stored in disparate databases and reorganized into a "data warehouse"



enables data mining or knowledge discovery that can have an impact on the success of the business itself.

Fig : Dimensions of the design model

3.4.2 Architectural Design Elements

The ***architectural design*** for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. Architectural design elements give us an overall view of the software.

The architectural model is derived from **three** sources: (1) information about the application domain for the software to be built; (2) specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and (3) the availability of architectural styles and patterns.

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model.Interface Design Elements

The interface design for software is analogous to a set of detailed drawings for the doors, windows, and external utilities of a house.

There are **three** important elements of interface design: (1) the user interface (UI); (2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and (3) internal interfaces between various design components.

These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

Component-Level Design Elements

The component-level design for software is the equivalent to a set of detailed drawings for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, sinks,

showers, tubs, drains, cabinets, and closets.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations.

Deployment-Level Design Elements

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

Deployment diagrams begin in descriptor form, where the deployment environment is described in general terms. Later, instance form is used and elements of the configuration are explicitly described.

Architectural Design

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.

What Is Architecture?

Bass, Clements, and Kazman define this elusive term in the following way:

"**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 you to

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) reduce the risks associated with the construction of the software.

Why Is Architecture Important?

Bass and his colleagues identify **three** key reasons that software architecture is 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" The architectural design model and the architectural patterns contained within it are transferable.

Architectural Descriptions

An architectural description of a software-based system must exhibit characteristics that are analogous to those noted for the office building.

The IEEE Computer Society has proposed, *Recommended Practice for Architectural Description of Software-Intensive Systems,* with the following objectives:

(1) to establish a conceptual framework and vocabulary for use during the design of software architecture,

(2) to provide detailed guidelines for representing an architectural description, and

(3) to encourage sound architectural design practices.

The IEEE standard defines an *architectural description* (AD) as "a collection of products to document an architecture." The description itself is represented using multiple views, where each *view* is "a representation of a whole system from the perspective of a related set of concerns."

Architectural Decisions

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Therefore, architectural decisions themselves can be considered to be one view of the architecture. The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

ARCHITECTURAL GENRES

The architectural ***genre*** will often dictate the specific architectural approach to the structure that must be built. In the context of architectural design, *genre* implies a specific category within the overall software domain. Within each category, you encounter a number of subcategories. **Grady Booch** suggests the following architectural genres for software-based systems:

- **Artificial intelligence**—Systems that simulate or augment human cognition, locomotion, or other organic processes.
- **Commercial and nonprofit**—Systems that are fundamental to the operation of a business enterprise.
- **Communications**—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure.
- **Content authoring**—Systems that are used to create or manipulate textual or multimedia artifacts. • **Devices**—Systems that interact with the physical world to provide some point service for an individual.
- **Entertainment and sports**—Systems that manage public events or that provide a large group entertainment experience.
- **Financial**—Systems that provide the infrastructure for transferring and managing money and other securities.
- **Games**—Systems that provide an entertainment experience for individuals or groups.
- **Government**—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.
- **Industrial**—Systems that simulate or control physical processes.
- **Legal**—Systems that support the legal industry.
- **Medical**—Systems that diagnose or heal or that contribute to medical research.
- **Military**—Systems for consultation, communications, command, control, and intelligence as well as offensive

and defensive weapons.

- **Operating systems**—Systems that sit just above hardware to provide basic software services.
- **Platforms**—Systems that sit just above operating systems to provide advanced services.
- **Scientific**—Systems that are used for scientific research and applications.
- **Tools**—Systems that are used to develop other systems.
- **Transportation**—Systems that control water, ground, air, or space vehicles.
- **Utilities**—Systems that interact with other software to provide some point service.

ARCHITECTURAL STYLES

An *architectural style* as a descriptive mechanism to differentiate the house from other 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, 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.

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system.

A Brief Taxonomy of Architectural Styles

Data-centered architectures. 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. The following figure illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. Data-centered architectures

promote ***integrability.***

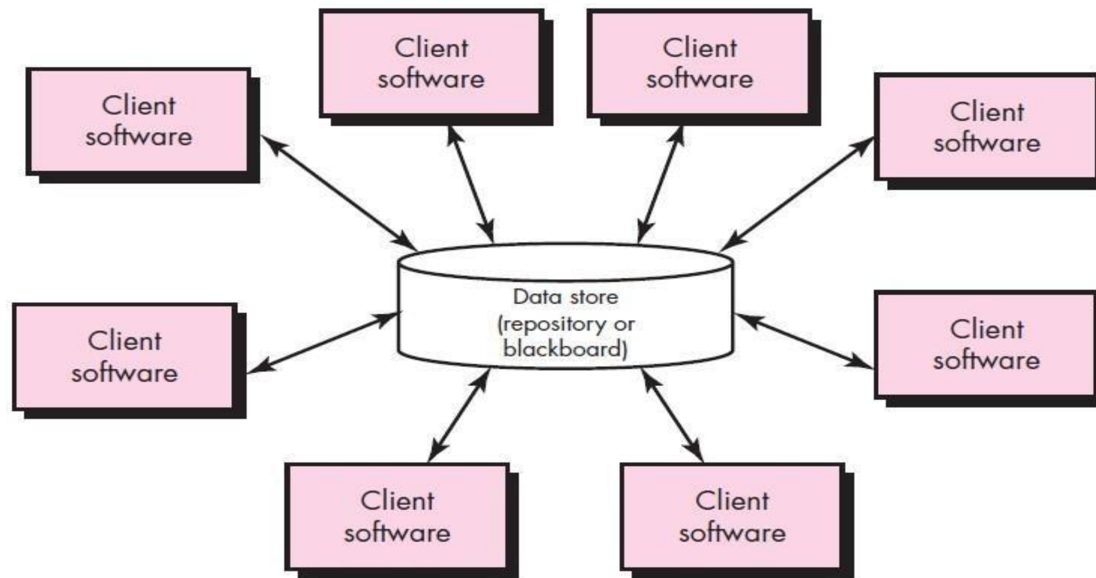


Fig : Data-centered architecture

Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern shown in following figure. It has a set of components, called ***filters***, connected by ***pipes*** that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output of a specified form. However, the filter does not require knowledge of the Workings of its neighboring filters.

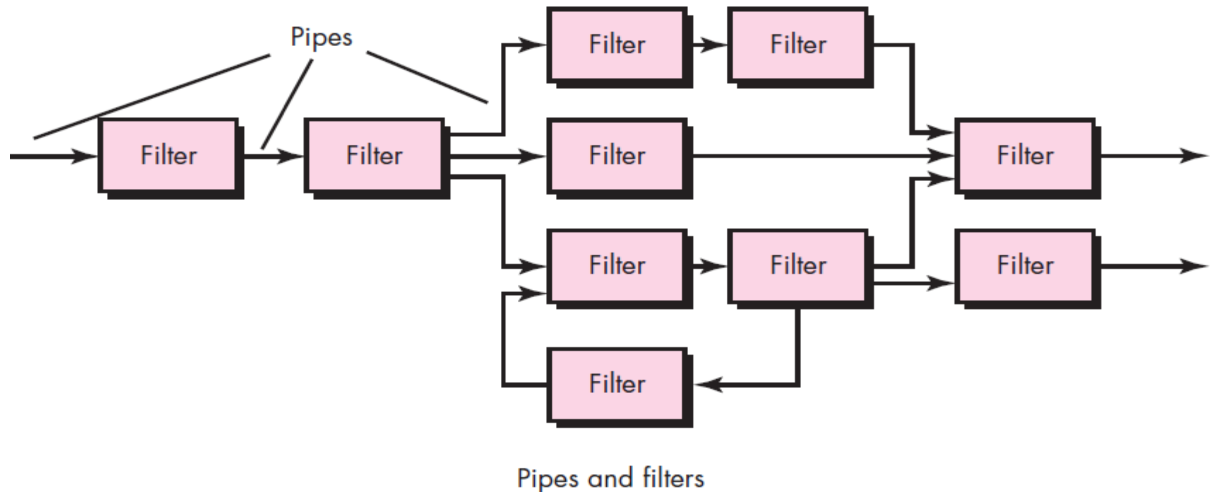
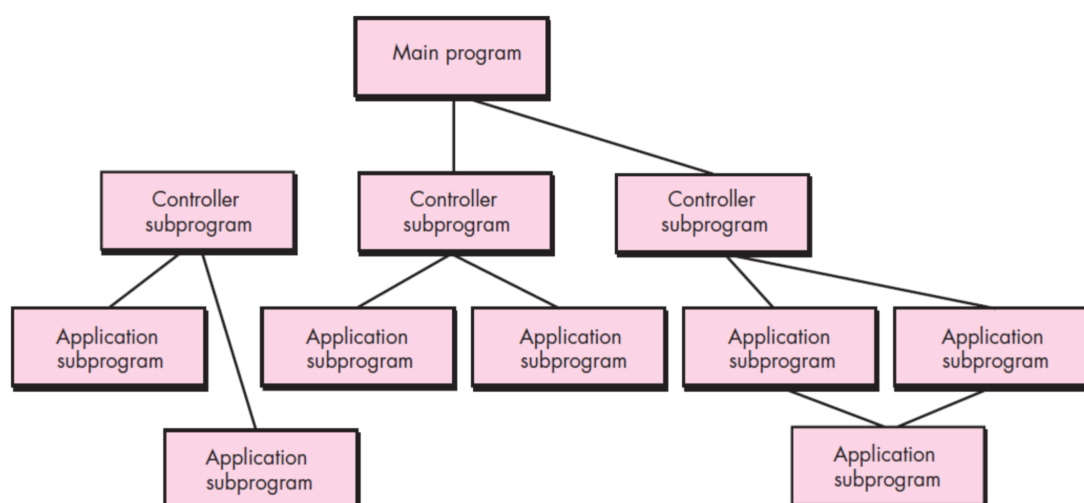


Fig : Data-flow architecture

Call and return architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of sub styles exist within this category:

- ***Main program/subprogram architectures***. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components. The following figure illustrates an architecture of this type.
- ***Remote procedure call architectures***. The components of a



main program/subprogram architecture are distributed across multiple computers on a network.

Fig : Main program/subprogram architecture

Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via **message passing**.

Layered architectures. The basic structure of a layered architecture is illustrated in following figure. 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.

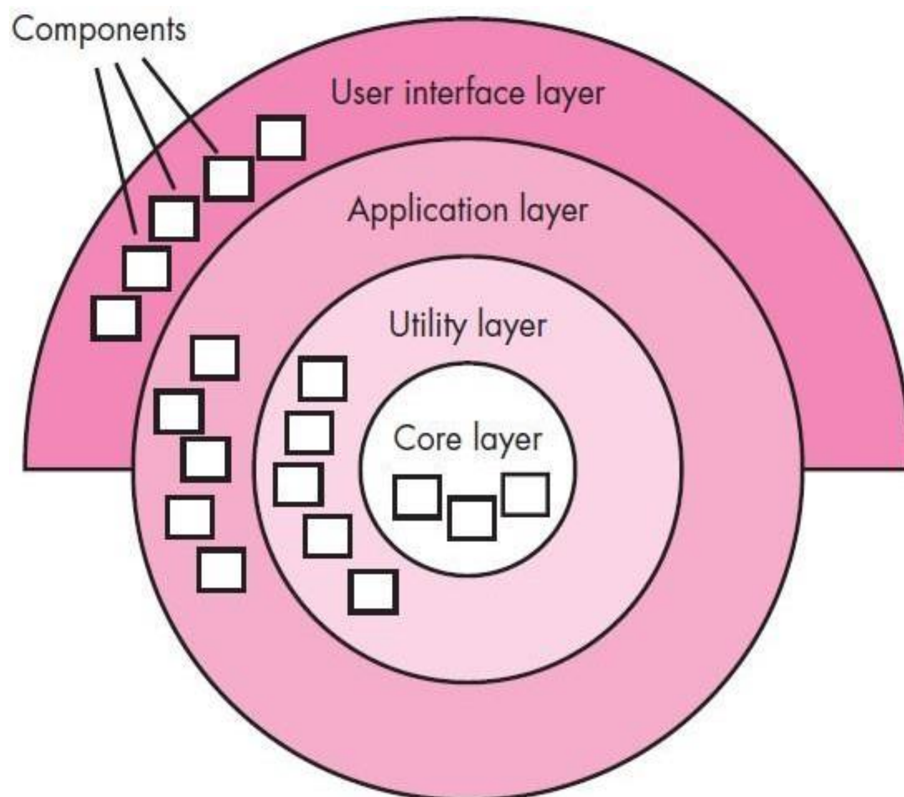


Fig : Layered architecture

Architectural Patterns

Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

Organization and Refinement

The following questions provide insight into an architectural style:

Control. How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology? Is control synchronized or do components operate asynchronously?

Data. How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer? Do data components exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active? How do data and control interact within the system?

These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

ARCHITECTURAL DESIGN

As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. Once context is modeled and all external software interfaces have been described, you can identify

a set of architectural **archetypes**.

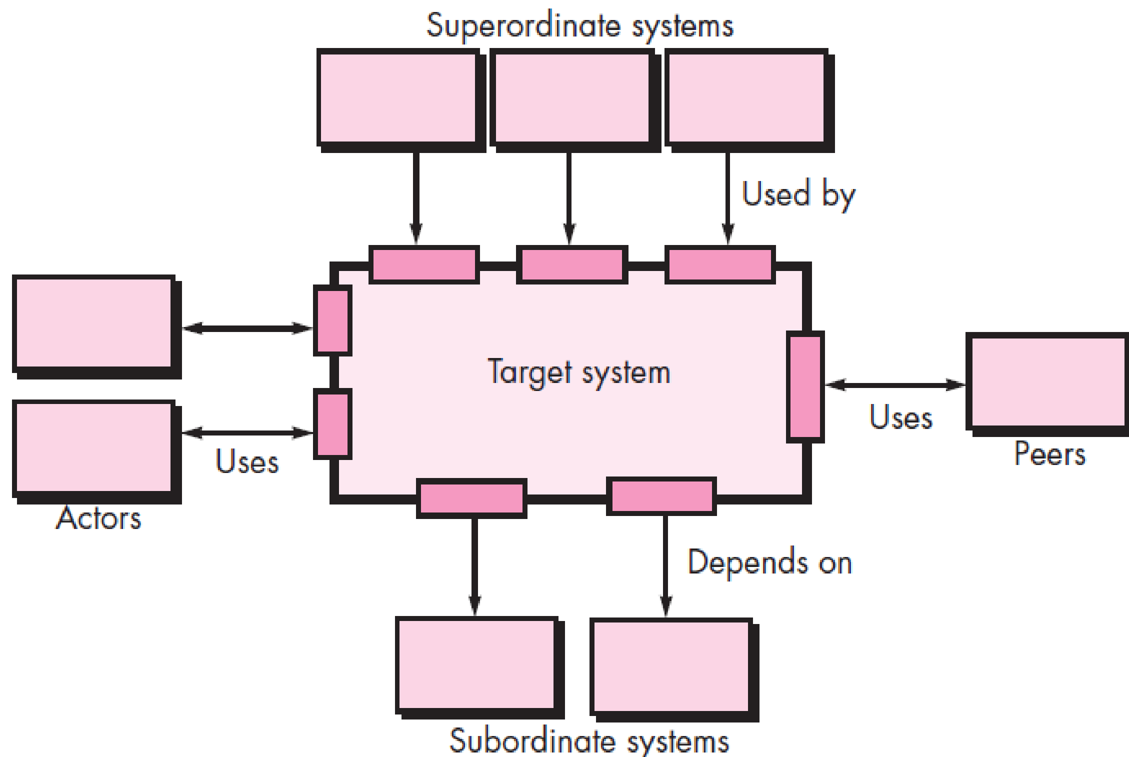
An ***archetype*** is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.

Representing the System in Context

At the architectural design level, a software architect uses an ***architectural context diagram*****(ACD)** to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in following figure. Referring to the figure, systems that interoperate with the *target system* (the system for which an architectural design is to be developed) are represented as

- ***Superordinate systems***—those systems that use the target system as part of some higher-level processing scheme.
- ***Subordinate systems***—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- ***Peer-level systems***—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.

• ***Actors***—entities (people, devices) that interact with the target



system by producing or consuming information that is necessary for requisite processing.

Fig : Architectural context diagram

Defining Archetypes

An ***archetype*** is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

The following archetypes can be used :

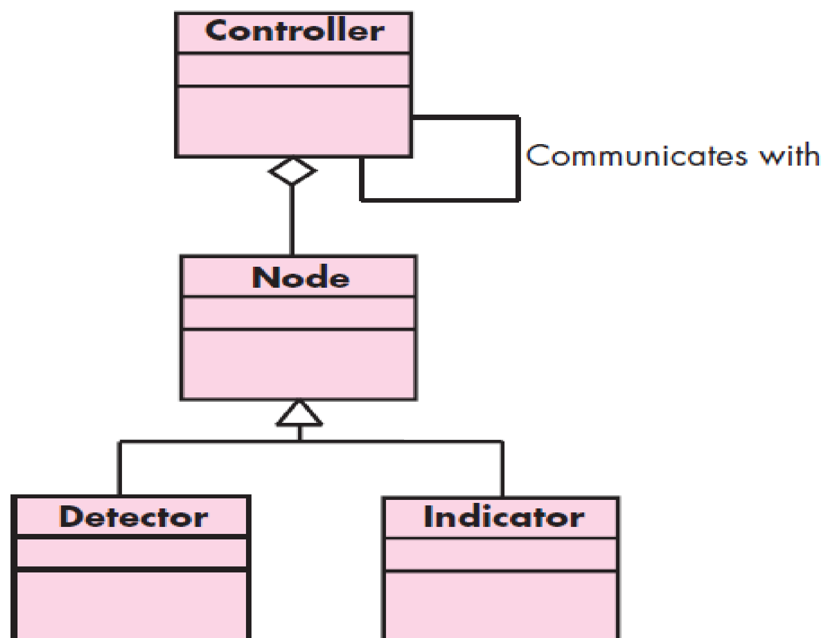
• **Node.** Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of

alarm (output) indicators.

• **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.

• **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

• **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If



controllers reside on a network, they have the ability to communicate with one another.

Refining the Architecture into Components

As the software architecture is refined into components, the structure of the system begins to emerge. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. Set of top-level components that address the following functionality:

• ***External communication management***—coordinates

communication of the security function with external entities such as other Internet-based systems and external alarm notification.

- ***Control panel processing***—manages all control panel functionality.
- ***Detector management***—coordinates access to all detectors attached to the system.
- ***Alarm processing***—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall architecture.

Component-level design

Component-level design occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established. The intent is to translate the design model into operational software.

WHAT IS A COMPONENT?

A ***component*** is a modular building block for computer software. More formally, the *OMG Unified Modeling Language Specification* defines a component as “a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”

The true meaning of the term ***component*** will differ depending on the point of view of the software engineer who uses it.

An Object-Oriented View

In the context of object-oriented software engineering, a component contains a set of collaborating classes. Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation. As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be

defined. To accomplish this, you begin with the requirements model and elaborate analysis classes and infrastructure classes.

The Traditional View

In the context of traditional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it. A traditional component, also called a **module**, resides within the software architecture and serves one of **three** important roles:

(1) A control component that coordinates the invocation of all other problem domain components,

(2) a problem domain component that implements a complete or partial function that is required by the customer, or

(3) an infrastructure component that is responsible for functions that support the processing required in the problem domain.

DESIGNING CLASS-BASED COMPONENTS

Basic Design Principles

Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied.

The Open-Closed Principle (OCP). ***"A module [component] should be open for extension but closed for modification"*** This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, you should specify the component in a way that allows it to be extended without the need to make internal modifications to the component itself.

The Liskov Substitution Principle (LSP). ***"Subclasses should be substitutable for their base classes"***. This design principle, originally proposed by Barbara Liskov, suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead. LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it. In the context of this discussion, a "contract" is a ***precondition*** that must be true before the component uses a base class and a ***post condition*** that should be true after the component uses a base class.

Dependency Inversion Principle (DIP). ***"Depend on abstractions. Do not depend on concretions"***. The more a component depends on other concrete components, the more difficult it will be to extend.

The Interface Segregation Principle (ISP). ***"Many client-specific interfaces are better than one general purpose interface"***. ISP suggests that you should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

The Release Reuse Equivalency Principle (REP). ***"The granule of reuse is the granule of release"***. When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity and the people who will use it. The developer commits to establish a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version. Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as

newer versions evolve.

The Common Closure Principle (CCP). ***"Classes that change together belong together."*** Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area. When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

The Common Reuse Principle (CRP). ***"Classes that aren't reused together should not be grouped together"***. When one or more classes within a package changes, the release number of the package changes. All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operates without incident. If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed.

Component-Level Design Guidelines

Ambler suggests the following guidelines:

Components. Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model.

Interfaces. Interfaces provide important information about communication and collaboration. Ambler recommends that (1) lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams

grow complex; (2) for consistency, interfaces should flow from the left-hand side of the component box; (3) only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available.

Cohesion

cohesion is the “single-mindedness” of a component. Lethbridge and Laganiére define a number of different types of cohesion

Functional. Exhibited primarily by operations, this level of cohesion occurs when a component performs a targeted computation and then returns a result.

Layer. Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

Communicational. All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

Coupling

Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep **coupling as low as is possible**.

Class coupling can manifest itself in a variety of ways. Lethbridge and Laganiére define the following coupling categories:

Content coupling. Occurs when one component “surreptitiously modifies data that is internal to another component”.

Common coupling. Occurs when a number of components all make use of a global variable. Although this is sometimes necessary, common coupling can lead to uncontrolled error propagation and

unforeseen side effects when changes are made.

Control coupling. Occurs when operation *A()* invokes operation *B()* and passes a control flag to

B. The control flag then "directs" logical flow within *B*. The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result.

Stamp coupling. Occurs when **ClassB** is declared as a type for an argument of an operation of **ClassA**. Because **ClassB** is now a part of the definition of **ClassA**, modifying the system becomes more complex.

Data coupling. Occurs when operations pass long strings of data arguments. The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.

Routine call coupling. Occurs when one operation invokes another. This level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system.

Type use coupling. Occurs when component **A** uses a data type defined in component **B.** If the type definition changes, every component that uses the definition must also change.

Inclusion or import coupling. Occurs when component **A** imports or includes a package or the content of component **B**.

External coupling. Occurs when a component communicates or collaborates with infrastructure components. Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

Software must communicate internally and externally. Therefore, coupling is a fact of life. However, the designer should

work to **reduce coupling whenever possible**.

CONDUCTING COMPONENT-LEVEL DESIGN

The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain. Using the requirements and architectural model, each analysis class and architectural component is elaborated.

Step 2. Identify all design classes that correspond to the infrastructure domain. These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point.

Step 3. Elaborate all design classes that are not acquired as reusable components. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

Step 3a. Specify message details when classes or components collaborate. The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system. Although this design activity is optional, it can be used as a precursor to the specification of interfaces that show how components within the system communicate and collaborate.

Step 3c. Elaborate attributes and define data types and data structures required to implement them. In general, data structures and types used to define attributes are defined within the context of

the programming language that is to be

Step 3d. Describe processing flow within each operation in detail. This may be accomplished using a programming language-based pseudocode or with a UML activity diagram. Each

software component is elaborated through a number of iterations that apply the stepwise refinement concept.

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

Step 5. Develop and elaborate behavioral representations for a class or component. UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design class.

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams are used as part of architectural design and are represented in descriptor form. In this form, major system functions (often represented as subsystems) are represented within the context of the computing environment that will house them. During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components.

Step 7. Refactor every component-level design representation and always consider alternatives. The first component-level model

you create will not be as complete, consistent, or accurate as the *n*th iteration you apply to the model.

Deployment level design elements

- The deployment level design element shows the software functionality and subsystem that allocated in the physical computing environment which support the software.
- Following figure shows three computing environment as shown. These are the personal computer, the CPI server and the Control panel.

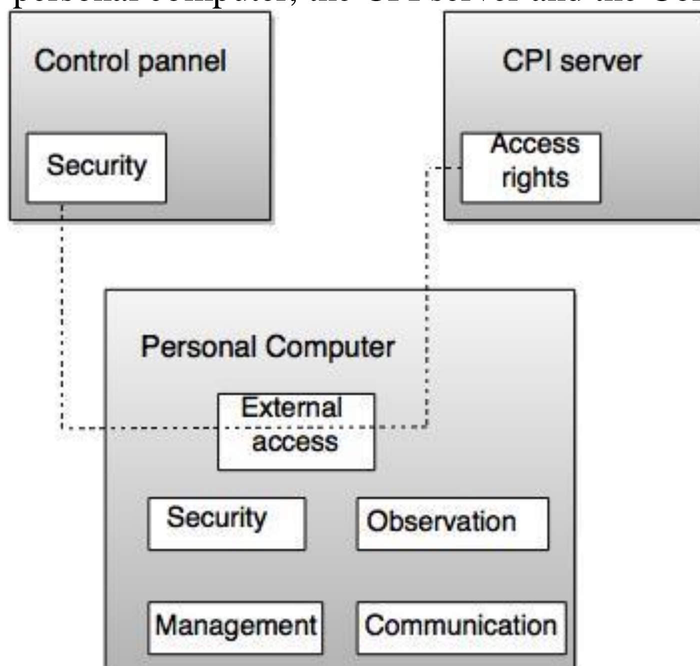


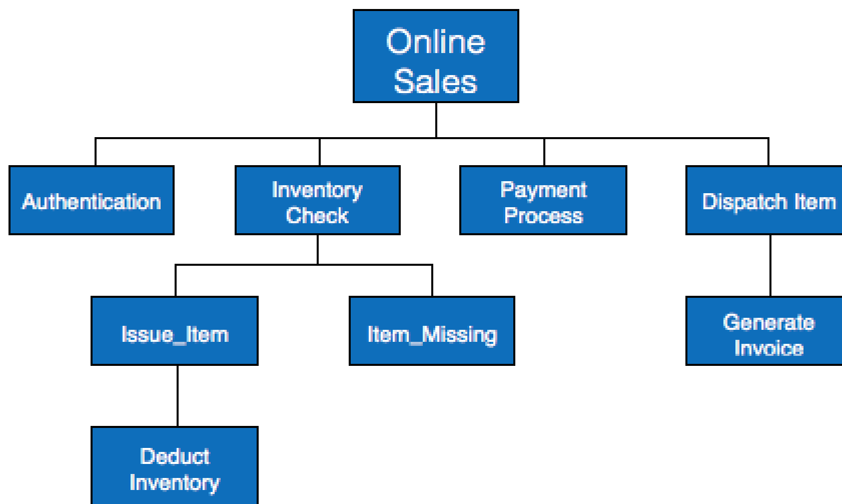
Fig. - Deployment level diagram

HIPO Diagram

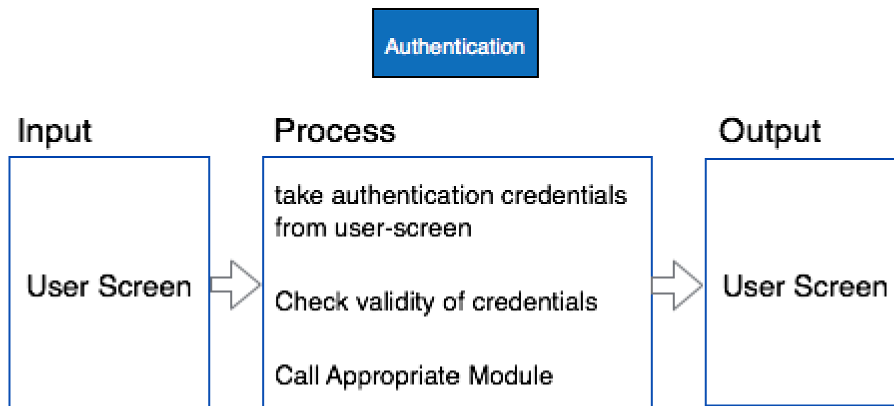
HIPO (Hierarchical Input Process Output) diagram is a combination of two organized method to analyze the system and provide the means of documentation. HIPO model was developed by IBM in year 1970.

HIPO diagram represents the hierarchy of modules in the software system. Analyst uses HIPO diagram in order to obtain high-level view of system functions. It decomposes functions into sub-functions in a hierarchical manner. It depicts the functions performed by system.

HIPO diagrams are good for documentation purpose. Their graphical representation makes it easier for designers and managers to get the pictorial idea of the system structure.



In contrast to IPO (Input Process Output) diagram, which depicts the flow of control and data in a module, HIPO does not provide any information about data flow or control flow.



Example

Both parts of HIPO diagram, Hierarchical presentation and IPO Chart are used for structure design of software program as well as documentation of the same.

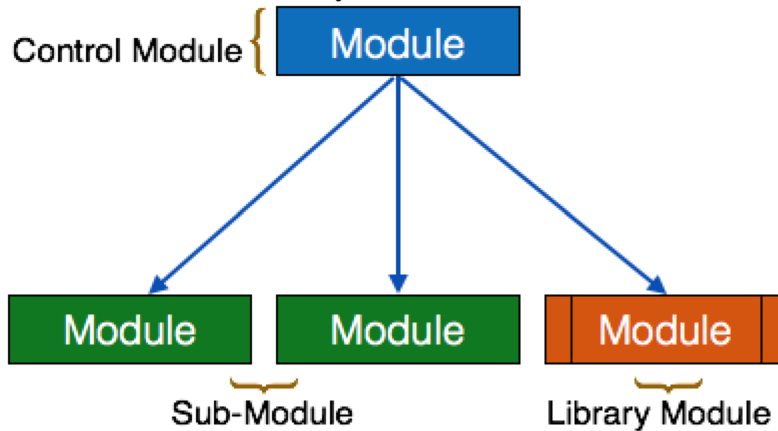
Structure Charts

Structure chart is a chart derived from Data Flow Diagram. It represents the system in more detail than DFD. It breaks down the entire system into lowest functional modules, describes functions and sub-functions of each module of the system to a greater detail than DFD.

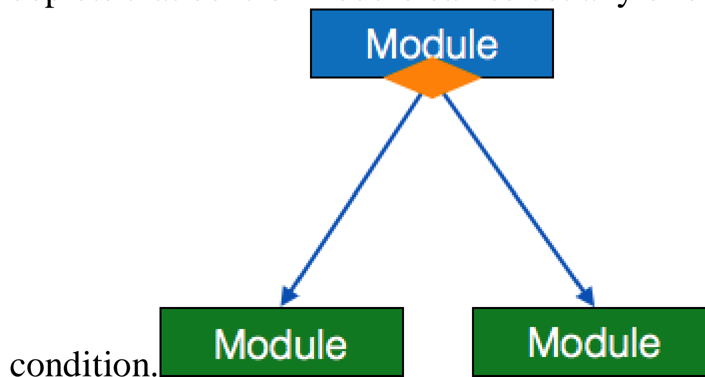
Structure chart represents hierarchical structure of modules. At each layer a specific task is performed.

Here are the symbols used in construction of structure charts -

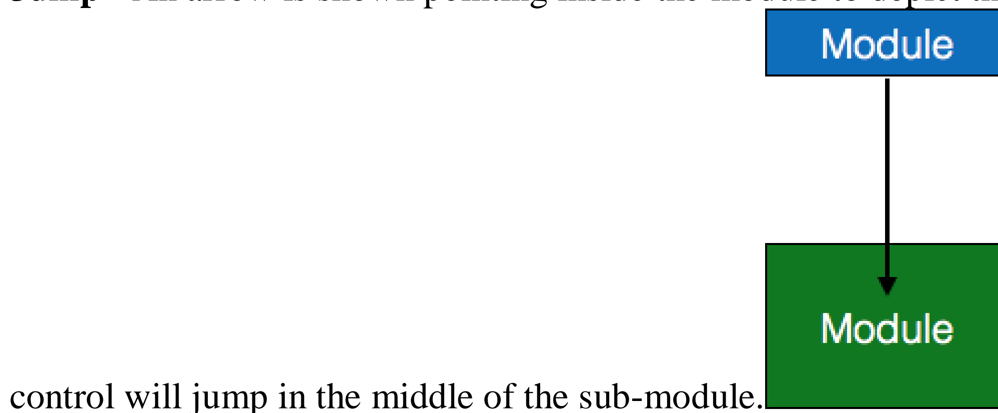
- **Module** - It represents process or subroutine or task. A control module branches to more than one sub-module. Library Modules are re-usable and invokable from any module.



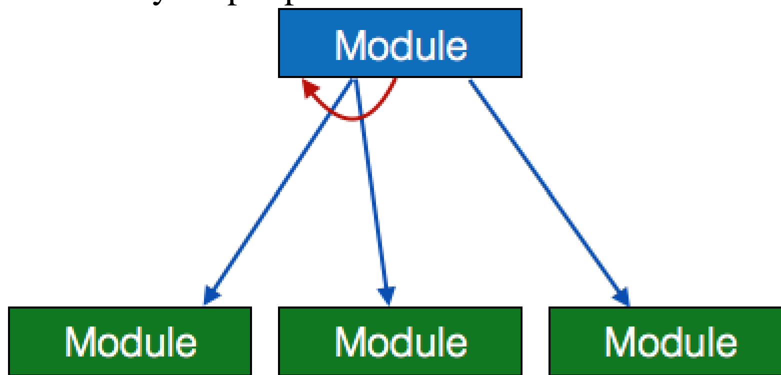
- **Condition** - It is represented by small diamond at the base of module. It depicts that control module can select any of sub-routine based on some condition.



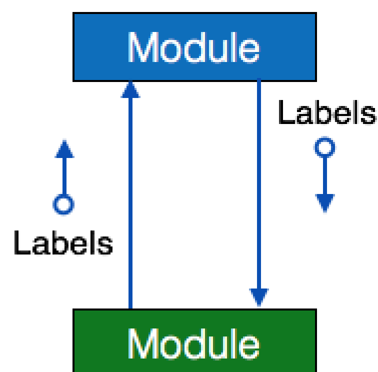
- **Jump** - An arrow is shown pointing inside the module to depict that the control will jump in the middle of the sub-module.



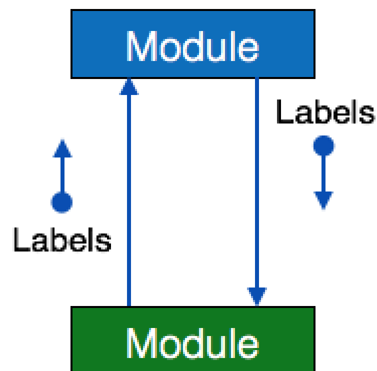
- **Loop** - A curved arrow represents loop in the module. All sub-modules covered by loop repeat execution of module.



- **Data flow** - A directed arrow with empty circle at the end represents data flow.



- **Control flow** - A directed arrow with filled circle at the end represents control flow.

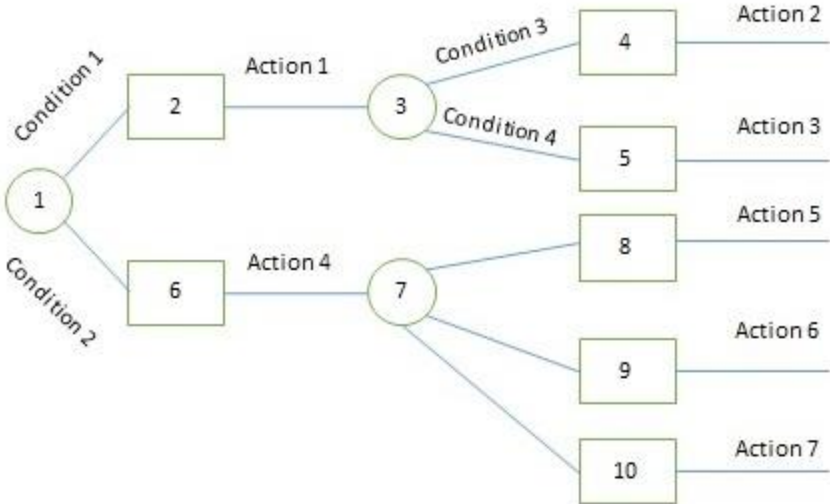


Decision Trees

Decision trees are a method for defining complex relationships by describing decisions and avoiding the problems in communication. A decision tree is a diagram that shows alternative actions and conditions within horizontal tree framework. Thus, it depicts which conditions to consider first, second, and so on.

Decision trees depict the relationship of each condition and their permissible actions. A square node indicates an action and a circle indicates a condition. It

forces analysts to consider the sequence of decisions and identifies the actual decision that must be made.



The major limitation of a decision tree is that it lacks information in its format to describe what other combinations of conditions you can take for testing. It is a single representation of the relationships between conditions and actions.

For example, refer the following decision tree −

Decision Tables

Decision tables are a method of describing the complex logical relationship in a precise manner which is easily understandable.

- It is useful in situations where the resulting actions depend on the occurrence of one or several combinations of independent conditions.
- It is a matrix containing row or columns for defining a problem and the actions.

Components of a Decision Table

- **Condition Stub** − It is in the upper left quadrant which lists all the condition to be checked.
- **Action Stub** − It is in the lower left quadrant which outlines all the action to be carried out to meet such condition.
- **Condition Entry** − It is in upper right quadrant which provides answers to questions asked in condition stub quadrant.
- **Action Entry** − It is in lower right quadrant which indicates the appropriate action resulting from the answers to the conditions in the condition entry quadrant.

The entries in decision table are given by Decision Rules which define the relationships between combinations of conditions and courses of action. In rules section,

- Y shows the existence of a condition.
- N represents the condition, which is not satisfied.
- A blank - against action states it is to be ignored.
- X (or a check mark will do) against action states it is to be carried out.

For example, refer the following table −

CONDITIONS	Rule 1	Rule 2	Rule 3
Advance payment made	Y	N	N
Purchase amount = Rs 10,000/-	-	Y	Y
Regular Customer	-	Y	N
ACTIONS			
Give 5% discount	X	X	-
Give no discount	-	-	X

Structured Flowchart

Structures
You can make your flowcharts easier to understand and less subject to errors by using only a fixed set of structures. These structures include:

- Sequence
- Decision
- Loop
- Case

Whether you are flowcharting software programs or business processes, using only these structures will make it easier to find and correct errors in your charts. Each structure has a simple flow of control with one input and one output. These structures can then be nested within each other. Any chart can be drawn using only these structures. You do not have to use GOTO or draw spaghetti diagrams just because you are drawing a flowchart. You can draw structured flowcharts.
The Colored Edge Shapes Stencil
The samples shown below were all drawn using RFFlow. It will allow you to draw charts just like these.

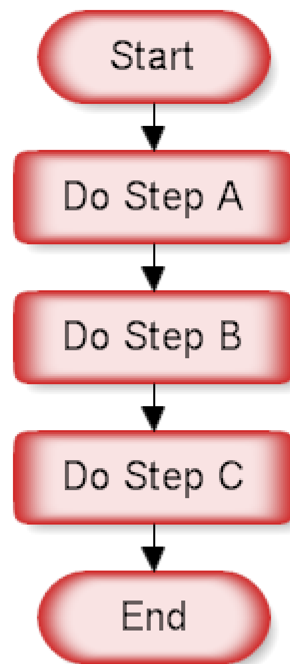
Once RFFlow is installed, run RFFlow and click on the **More Shapes**

button.

Scroll to the **Flowcharting** folder and click the plus sign to open it.

Click the **Colored Edge Shapes** stencil and then click the **Add Stencil** button.

Sequence

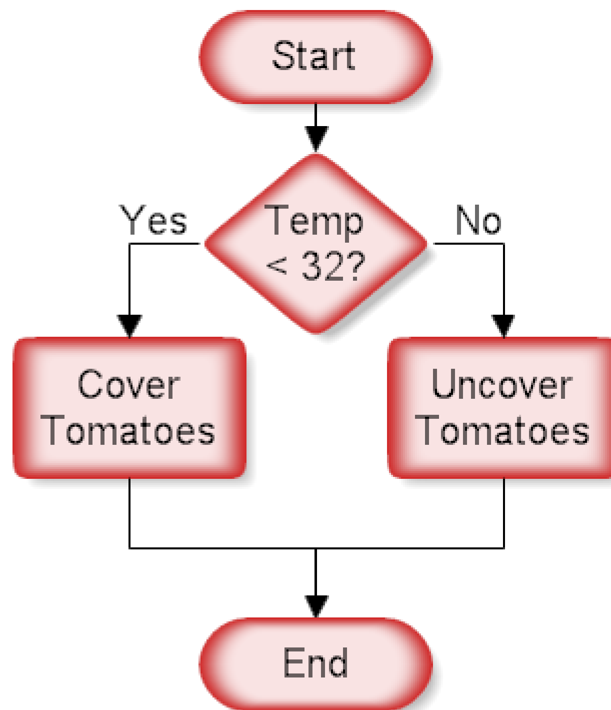


The flowchart above demonstrates a sequence of steps. The reader would start at the Start shape and follow the arrows from one rectangle to the other, finishing at the End shape. A sequence is the simplest flowcharting construction. You do each step in order.

If your charts are all sequences, then you probably don't need to draw a flowchart. You can type a simple list using your word processor. The power of a flowchart becomes evident when you include decisions and loops.

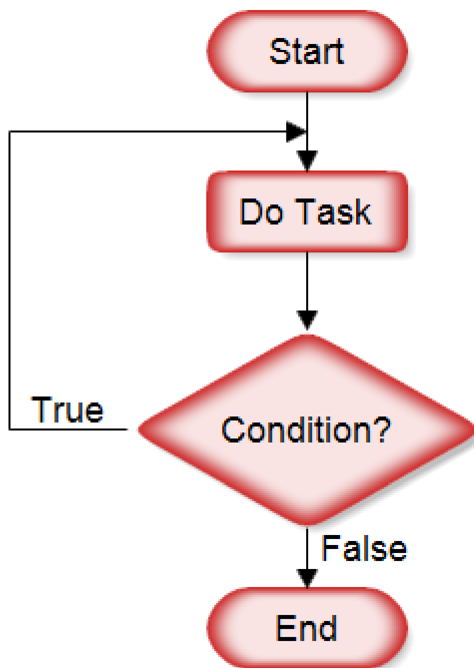
RFFlow allows you to number your shapes if you wish. Run RFFlow and click on **Tools**, **Number Shapes**, and put a check mark in **Enable numbers for the entire chart**. You can also choose to have a number or not in each individual shape and you can quickly renumber your chart at any time.

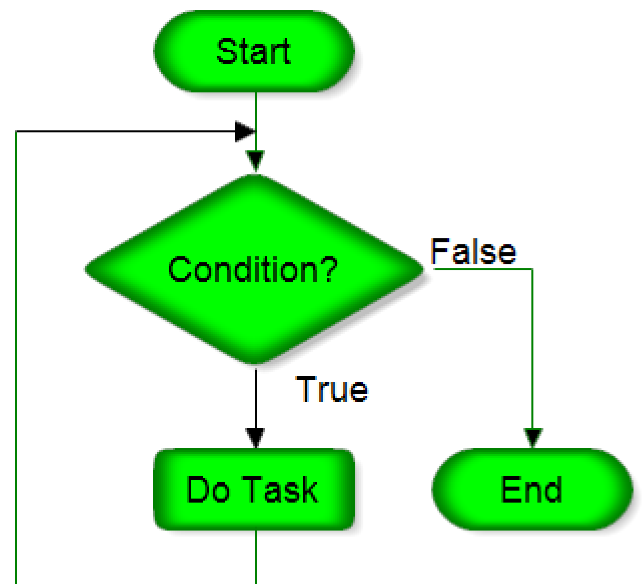
Decision



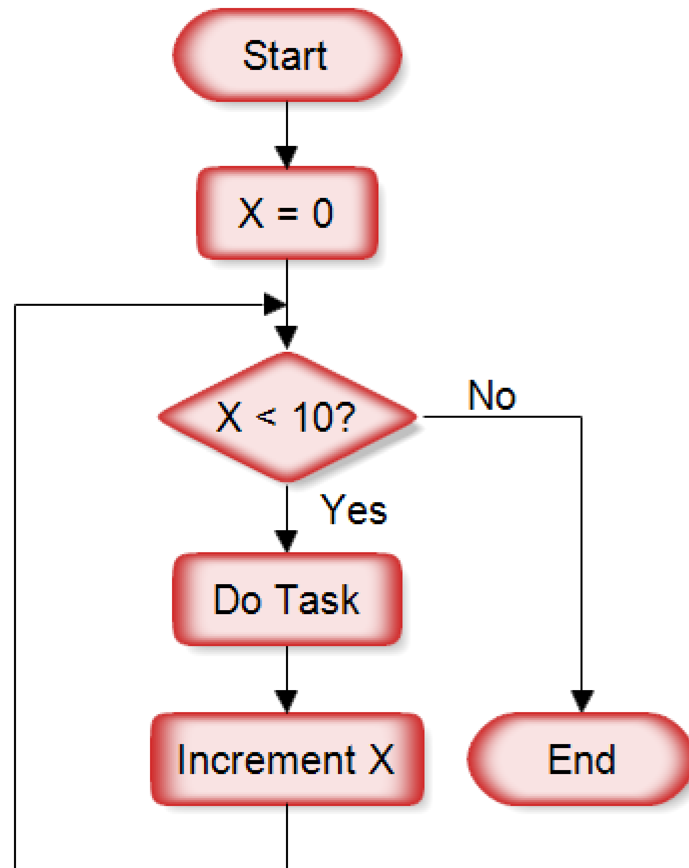
This structure is called a decision, "If Then.. Else" or a conditional. A question is asked in the decision shape. Depending on the answer the control follows either of two paths. In the chart above, if the temperature is going to be less than freezing (32 degrees Fahrenheit) the tomatoes should be covered. Most RFFlow stencils include the words "Yes" and "No" so you can just drag them onto your chart. "True" and "False" are also included in most of the flowcharting stencils.

Loop





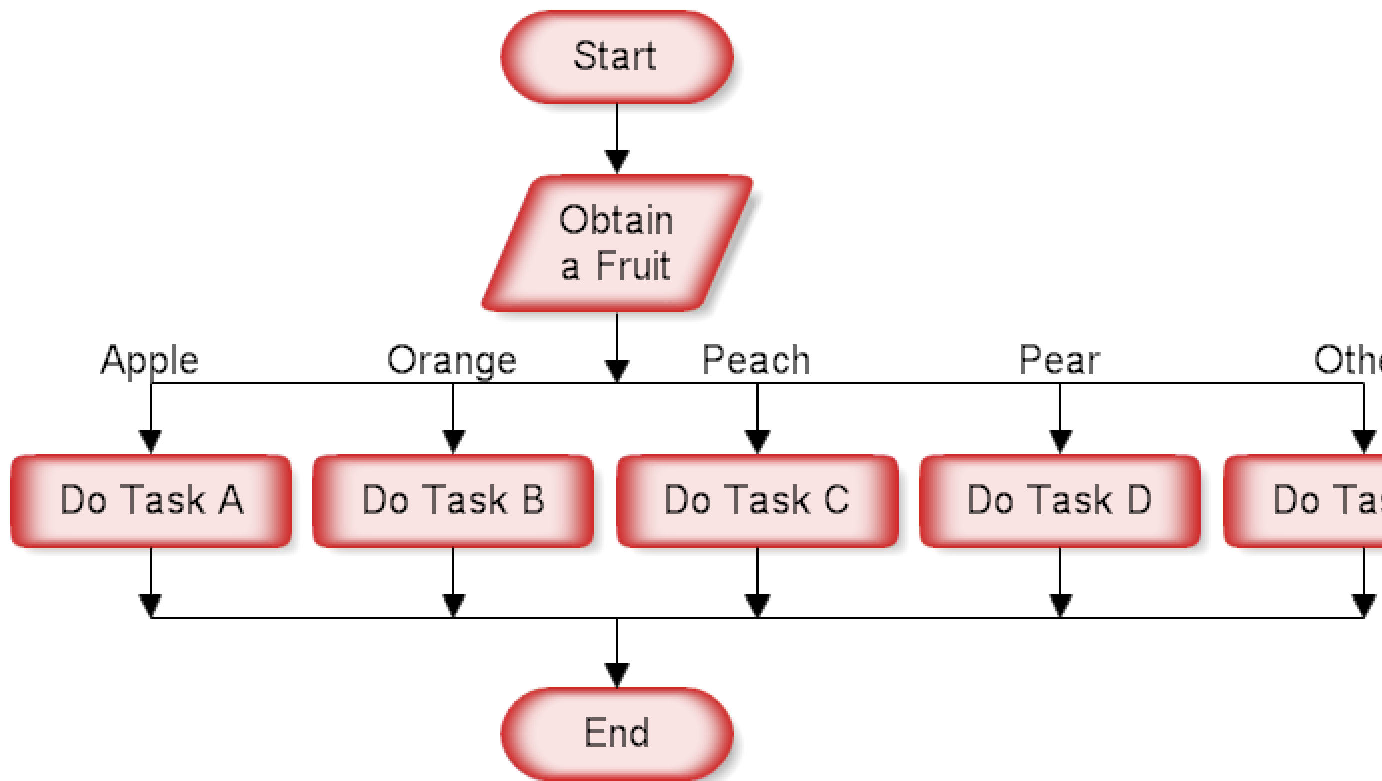
This structure allows you to repeat a task over and over. The red chart above on the left does the task and repeats doing the task until the condition is false. It always does the task at least once. The green chart on the right checks the condition first and continues doing the task while the condition is true. In the green chart the task may not be done at all. You can also have the conditions reversed and your loop is still a structured design loop.



The above chart is a "For Loop." In this example the task is performed 10 times as X counts from 0 to 10. Depending on the condition, the task may not be performed at all.

There is also a "For Each" structure that is like the for loop, but has no counter. It will go through each item of a collection and do the task. You don't have to know the length of the collection or use a counter. It is essentially saying "do this for every item in the collection".

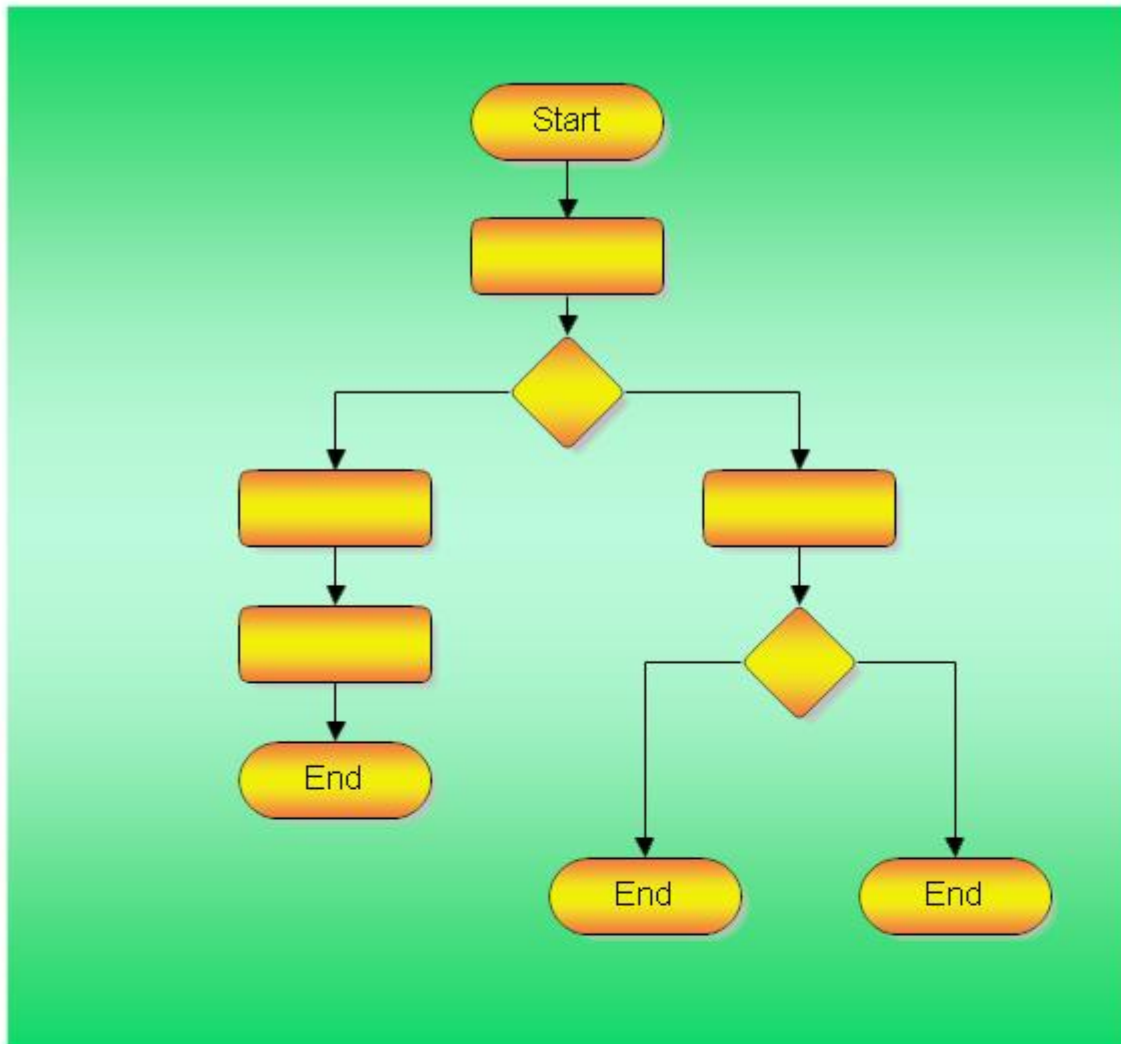
Case



The structure above is called the case structure or selection structure. The decision works fine if you have only two outputs, but if there are several, then using multiple decisions makes the chart too busy. Since the case structure can be constructed using the decision structure, it is superfluous, but useful. The case structure helps make a flowchart more readable and saves space on the paper.

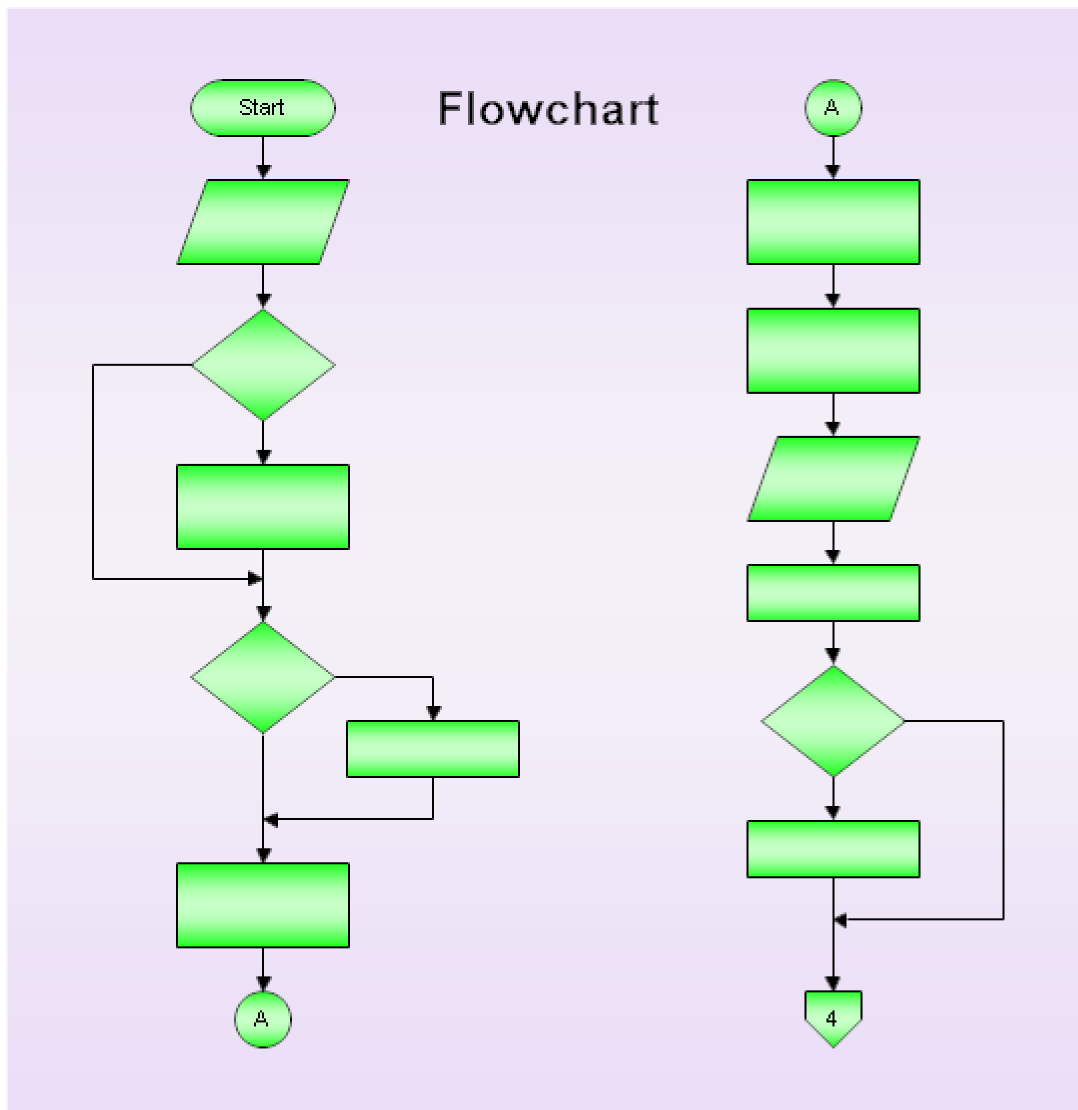
Other Good Design Practices

Start and End



Each flowchart must have one starting point. It can have multiple ending points, but only one starting point. The same terminal shape is used for the start and end. The terminal shape is a rectangle that is semicircular on the left and right as shown above. You can use other words instead of start and end, like begin and finish, or any words with a similar meaning. Some companies use an oval instead of a terminal shape. The bottom line is that it should be clear to the person looking at the chart where the chart starts and where it ends.

Connector Block and Off Page Connector



Pseudo-Code

Pseudo code is written more close to programming language. It may be considered as augmented programming language, full of comments and descriptions.

Pseudo code avoids variable declaration but they are written using some actual programming language's constructs, like C, Fortran, Pascal etc.

Pseudo code contains more programming details than Structured English. It provides a method to perform the task, as if a computer is executing the code.

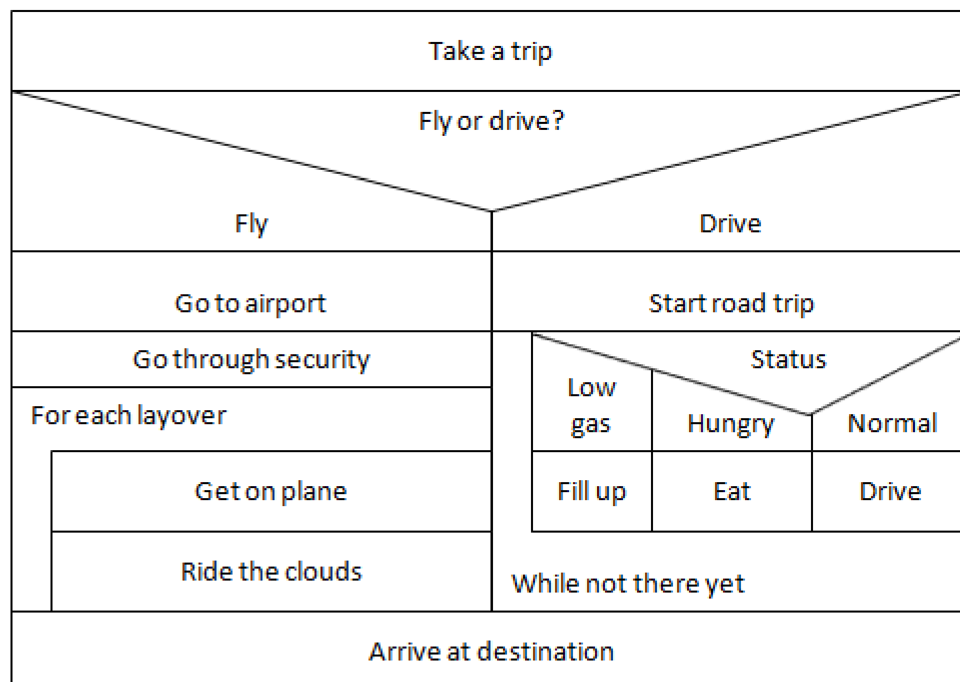
Example

Program to print Fibonacci up to n numbers.

```
void function Fibonacci
Get value of n;
Set value of a to 1;
Set value of b to 1;
Initialize I to 0
for (i=0; i< n; i++)
{
  if a greater than b
  {
    Increase b by a;
    Print b;
  }
  else if b greater than a
  {
    increase a by b;
    print a;
  }
}
```

Nassi-Shneiderman Diagram

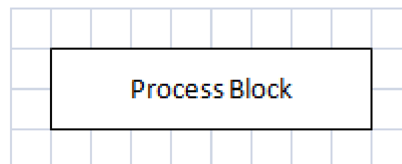
Nassi-Shneiderman diagrams (aka, NS diagrams or structograms), are used to outline structured programs. They are not very common in industry today but are sometimes used as a computer science teaching tool, often as an alternate to flowcharts. A simple example is shown below.



Sample Nassi-Shneiderman diagram

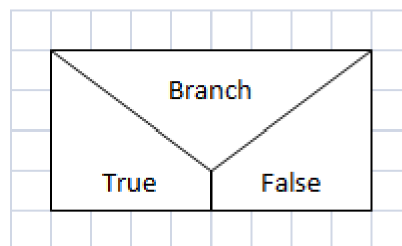
Nassi-Shneiderman Shapes

Process

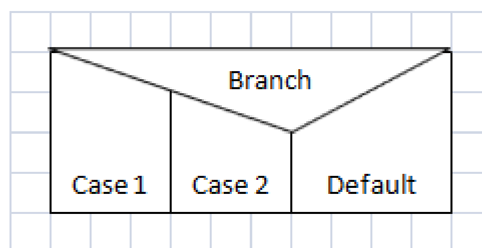


Any statement that is not
a branch or loop.

Branches

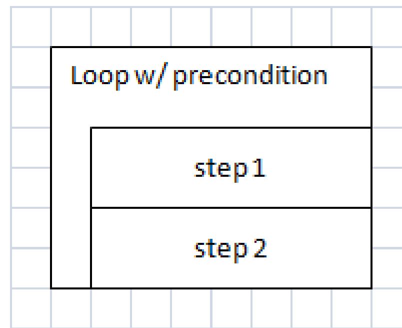


Binary branch statement,
such as an if statement
with a true/false choice.

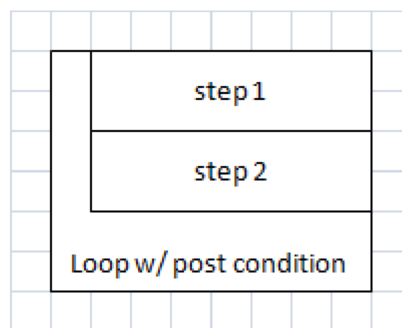


Multiple branches,
such as a switch-case statement.
Default case is the short leg of the triangle.

Loops



Loop with precondition,
such as a for loop or a while loop



Loop with postcondition,
such as a do-while loop.

Nassi-Shneiderman

Diagram in Excel

Adding a bunch of shapes to the diagram is fairly straightforward. (Well, except for the Branch triangle, which is a pain. We'll show why below.) Since the default shape style in Excel is dark blue with centered text, we will create some baseline shapes with plain styling that we place off to the side and then copy and paste to build the diagram.

Create a Grid

Before adding shapes, the first step is to create a grid and then turn on Snap to Grid. These steps are covered in the How to Flowchart in Excel article, so we won't repeat them here.

Create a Baseline Process Block

Click the Insert tab, then click the Shapes dropdown and select either a Rectangle from the Basic Shapes group or a Process shape from the Flowchart group. Use you left mouse button to draw it to size on the sheet. With the shape still selected, right-click on the shape and select Format Shape from the context menu. Follow these steps to set the styling:

- On the Format Shape dialog, select Fill from the left menu. Choose the Solid fill option and use the Color dropdown to set the color to white.
- Next, select Line Color from the left menu. Choose the Solid line option and use the Color dropdown to set the color to black.
- Next, select Line Style from the left menu. Change the Width setting to 0.75 pt.
- Click the Close button
- Type "text" (or some othert placeholder text) into the shape.
- With the shape still selected, click the Home tab, and use the font, font size, and font color toolbar controls to set the font as desired.
- Right-click on the shape again and select Set as Default Shape from the menu. Despite the misleading menu label, this sets the default style, not the default shape type. Now all shapes added will use this

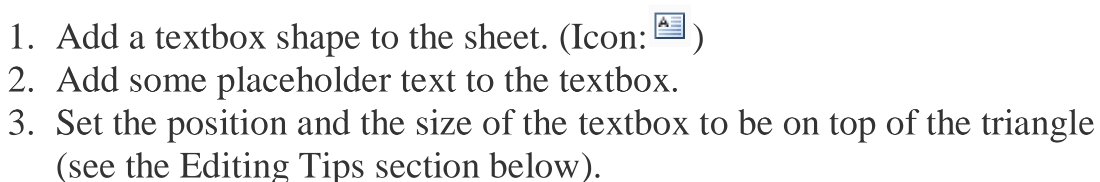
Create a Baseline Branch Shape (Triangle)

There are several triangle shapes available in Excel, but the only one that is suitable is the Isoceles Triangle under the Basic Shapes group. It has an adjustment handle (covered below in Adjusting Branch Shapes) that lets you move the center vertex for making switch branches. There is one problem, though. It points upward, and if we rotate the shape, then any text will be upside down. So what we will do is to create a branch shape and use a borderless, transparent textbox on top of that.

Triangle:

1. Add the Isoceles triangle as you did the process rectangle before.
2. Next, right-click on the triangle and select Size and Properties from the menu.
3. On the size dialog, change the Rotation to 180°.

Textbox:

1. Add a textbox shape to the sheet. (Icon: )
2. Add some placeholder text to the textbox.
3. Set the position and the size of the textbox to be on top of the triangle (see the Editing Tips section below).

Grouping the Triangle and Textbox:

Grouping shapes together lets you treat multiple shapes as a single object, which is how we will want to use our contrived branch block.

1. Click on the textbox to select it.
2. Hold the shift key and click on the triangle so both shapes are selected.
3. Right-click on a line of either shape and select Grouping > Group from the menu.

Final Edit:

Once shapes are grouped you can select individual shapes within the group by first clicking on the group to select it, and then clicking again on the individual shape. Do this now to select the textbox. Right-click on the textbox and use the Format Shape dialog to change the Fill to None and the Line Color to No line. You now have a reusable branch element.

Add Shapes to the Diagram

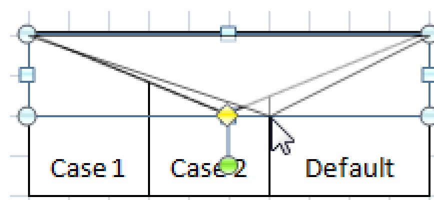
Now that we have created the base shapes, creating the diagram is just a matter of copying and pasting the baseline shapes. Of course, you will need to position and size the shapes as you build the diagram. You will also need to set the z-order* of the triangles and adjust their vertices so that the branch statements are properly positioned on top of the process blocks, which are both covered below in the Editing tips section.

** Z-order is the front to back positioning. When shapes overlap, it determines which shape is on top.*

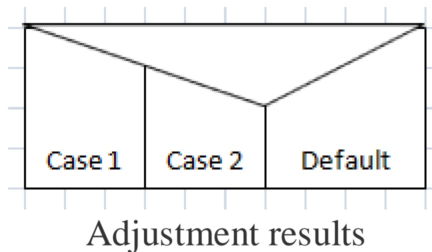
Editing Tips

Adjusting Branch Shapes

In Excel, shapes that can be altered display yellow "adjustment handles" at the adjustment points. You can click and drag these points to change the shape of the shape, so to speak.



Triangle adjustment



Adjustment results

Changing The Z-Order (Stacking Order)

When dealing with branches and loop tests, it is often necessary to change the z-order stacking of a shape by bringing it forward or sending it backward. The easiest way to do this is to right click on a shape and use Bring to Front or Send to Back from the context menu. But the following keyboard shortcuts are useful when a shape is hard to select with a mouse.

- *Alt + P A E K* - Send to Back: Places the shape underneath all other shapes.
- *Alt + P A E B* - Send Backward: Sends the shape one layer down.
- *Alt + P A F R* - Bring to Front: Places the shape on top of all other shapes.
- *Alt + P A F F* - Bring Forward: Brings the shape one layer up.

The multi-key shortcuts look odd compared to most keyboard shortcuts, but when you hold the Alt key, Excel hightlights them on the screen making the path becomes obvious. The letters map to characters in each word, even though they are not underlined like menu systems. For example, Send to Back is Page Layout > Arrange > Send Backward > Send to Back.

Selecting Shapes

You can select shapes with your mouse and use the Tab key to toggle between selected shapes. To select multiple shapes, click the first shape and then hold the Shift key down as you click on the others. You can also use a special Select Objects cursor available on the Home tab under the Find & Select menu. You need to toggle this cursor off to resume normal mouse usage - via the menu or by double-clicking anywhere on the worksheet.

Moving Shapes

Clicking and dragging with the mouse is the most obvious way, but you can also use the keyboard arrow keys to move a shape around. With Snap to Grid on, shapes will snap to the next cell as you do this.

Changing Text Alignment

On the Home tab, use the standard horizontal and vertical text alignment buttons.

Hiding the Excel Gridlines

On the View tab, uncheck the Gridlines checkbox.

Saving To Non-Excel Formats

Starting with Excel 2007, copy operations place an image of the copied range onto the Windows clipboard. To save the diagram as an image, select the cells fully encompassing the diagram, copy, and then paste into an image editor (even MS Paint will suffice). Alternately, you can paste into Word. I recommend doing a Paste Special and choosing the Enhanced Metafile format. Metafiles are like vector graphics in that when you resize the image, the lines and text will still render nicely.