1.1 Definitions

Software: is a computer program which may be applied for specified set of procedural steps has been defined (e.g., algorithm).

Software engineering: is the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.

Software products: are software systems which delivered to a customer with the documentation that describes how to install and use the system. Software products may be developed for a particular customer or may be developed for a general marker. There are two kinds of software products:

1) **Generic products**: These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include software for PCs such as databases, word processors, drawing packages, and project-management tools.
2) **Customized products**: These are systems that are wanted by a particular customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process, and air traffic control systems.

Process: is a collection of activities, actions, and tasks that are performed when some work product is to be created.

1.2 Software Application Domains

1) **System software**: a collection of programs written to service other programs such as compilers, editors, file management utilities, operating system components, and networking software.
2) **Application software**.
3) **Engineering/scientific software**.

4) **Embedded software**: resides within a product or system and is used to implement and control features and functions for the end user and for the system itself such as digital functions in an automobile (fuel control, dashboard displays, and braking systems).
5) **Product-line software**.
6) **Web/mobile applications**.
7) **Artificial intelligence software**: makes use of heuristics to solve complex problems that can't be solved using regular computation or straightforward analysis. Applications within this area include robotics, decision-making systems, pattern recognition (image and voice), machine learning, and game playing.

1.3 Software Process

A **software process** is a sequence of related activities that leads to the production of a software product. There are four fundamental activities that are common to all software processes. These activities are:

1) **Software specification** (requirements engineering), where customers and engineers define the software that is to be produced and the constraints on its operation.
2) **Software development** (software design and implementation), where the software is designed and programmed.
3) **Software validation**, where the software is checked to ensure that it is what the customer requires.
4) **Software evolution**, where the software is modified to reflect changing customer and market requirements.

The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes. In the **waterfall model**, they are organized in sequence, whereas in **incremental development** they are interleaved. How these activities are carried out depends on the type of software, people, and organizational structures involved.

2.1 Introduction

The **requirements** for a system are the descriptions of what the system should do, the services that it provides and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information.

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development.

There are four main activities in the requirements engineering process:

1) **Feasibility study**.
2) **Requirements elicitation and analysis**.
3) **Requirements specification**.
4) **Requirements validation**.

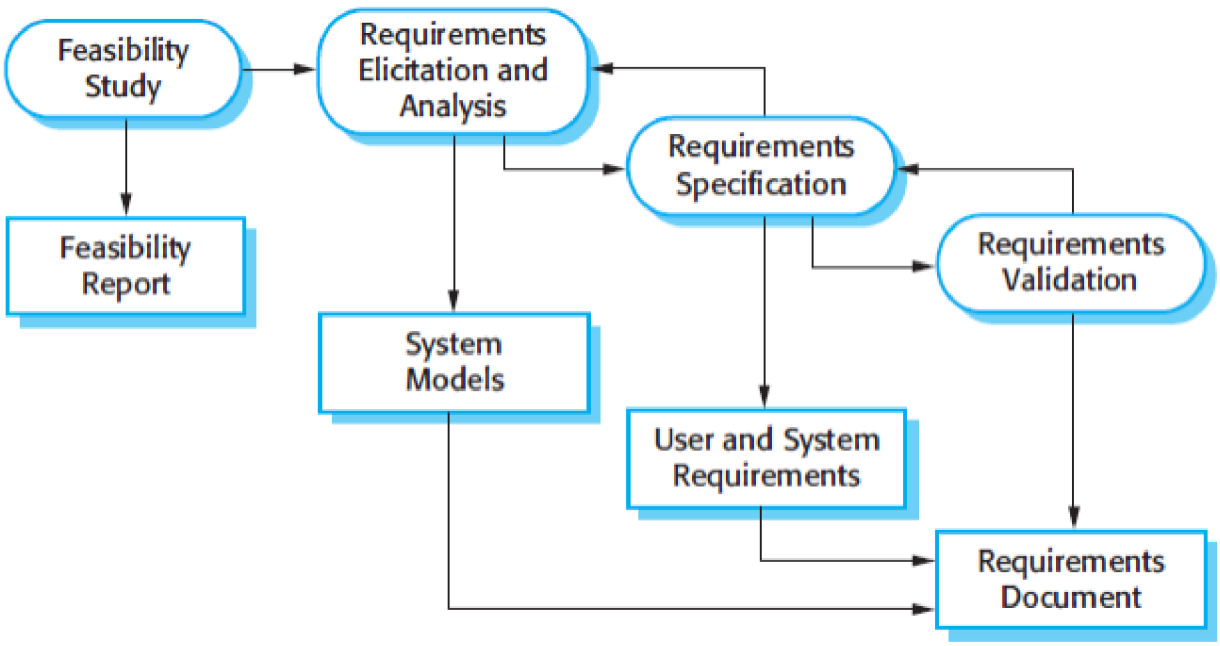


Figure 2.1: Requirements engineering process.

2.2 Requirements Types

Requirements can be classified into two types:

1) **User requirements**: are statements of what services the system is expected to provide to system users and the constraints under which it must operate. User requirements should specify only the external behavior of the system. User requirements should be written in natural language, with simple tables, forms, and simple diagrams (not used software codes, structured notation, or formal notations). These requirements are used by client managers, system end-users, client engineers, and system architects.
2) **System requirements**: are more detailed descriptions of the software system's functions, services, and operational constraints. They should not concern with how the system should be designed or implemented. These requirements are used by system end-users, client engineers, system architects, and system developers.

Another classification for requirements can be found as follow:

1) **Functional requirements**: these are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements.
2) **Non-functional requirements**: these are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services.

it is often more difficult to relate components to non-functional requirements because of two reasons:

1) Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.

2) A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. In addition, it may also generate requirements that restrict existing requirement.

2.3 Feasibility Study

an estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether or not to go ahead with a more detailed analysis.

2.4 Requirements Elicitation and Analysis

This is the process of driving user and system requirements by specifying the application domain, what services the system should provide, the required performance of the system, hardware and software constraints, and so on.

A **system stakeholder** is anyone who should have some direct or indirect influence on the system requirements. Stakeholders include end-users who will interact with the system, engineers who are developing or maintaining other related systems, business managers, domain experts, and trade union representatives.

This process consists of the following activities:

1) **Requirements discovery** (sometime called **requirements elicitation**): is the process of gathering information about the required system and existing systems, and obtaining the user and system requirements from this information. Sources of information during the requirements discovery phase include documentation, system stakeholders, specifications of other systems that interact with the specified system, and specifications of similar systems. Interaction with

stakeholders is done through interviews and observation and using scenarios and prototypes to help stakeholders understand what the system will be like.

2) **Requirements classification and organization**: this activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system.
3) **Requirements prioritization and negotiation**: when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.
4) **Requirements specification**: the requirements are documented and input into the next phase. Formal or informal requirements documents may be produced.

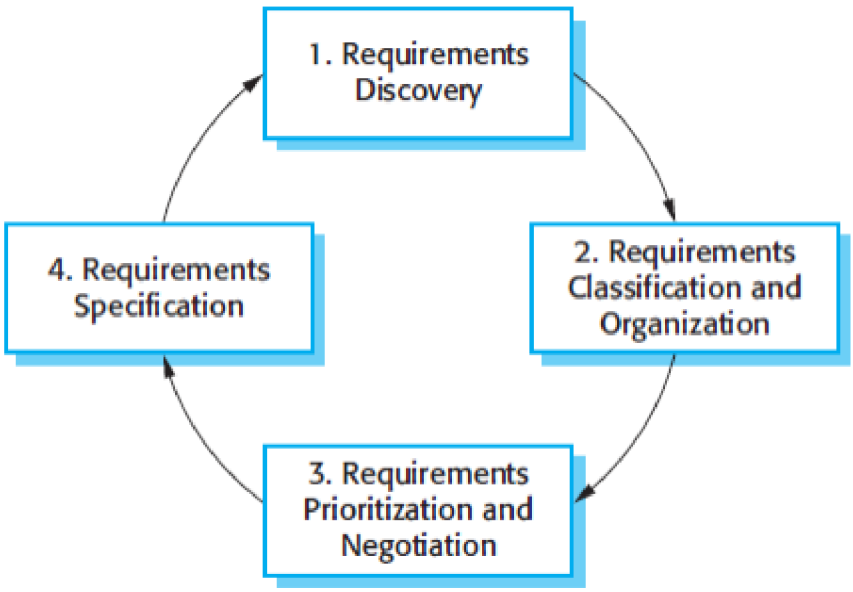


Figure 2.2: Requirements elicitation and analysis

A set of operational principles relates analysis methods. These principles are:

1) The information domain of a problem must be represented and understood.
2) The functions that the software performs must be defined.

3) The behavior of the software (as a consequence of external events) must be represented.
4) The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
5) The analysis task should move from essential information toward implementation detail.

2.5 Requirements Specification

It is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements (both user and system).

The user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent.

The **software requirements document** (sometimes called the **software requirements specification or SRS**) is an official statement of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements. Sometimes, the user and system requirements are integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification. If there are a large number of requirements, the detailed system requirements may be presented in a separate document.

Requirements document is used by:

1) **System customers**: use the document to specify the documents and read them to check that they meet their needs.
2) **Managers**: use the document to plan a bid for the system and to plan the system development process.
3) **System engineers**: use the document to understand what system is to be developed.
4) **System test engineers**: use the document to develop validation tests for the system.

5) **System maintenance engineers**: use the document to understand the system and the relationships between its parts.

2.6 Requirements Validation

Requirements validation is the process of checking requirements that are gathered, analyzed, and documented. It is important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

1) **Validity** checks: a user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required.
2) **Consistency** checks: requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.
3) **Completeness** checks: the requirements document should include requirements that define all functions and the constraints intended by the system user.
4) **Realism** checks: using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.
5) **Verifiability**: to reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that a set of tests should be written to ensure that the delivered system meets each specified requirement.

2.7 Requirements Modeling

The requirements modeling action results in one or more of the following types of models:

- **Data models**

  They depict the information domain for the problem. An example of such models is **Entity Relationship Diagram (ERD).**

- **Functional models**

  Software transforms information, and in order to accomplish this, it must perform at least three generic functions: input, processing, and output. When functional models of an application are created, the software engineer focuses on problem specific functions. Examples of such models are **Data Flow Diagram (DFD)** and **Control Flow Diagram (CFD)**.

- **Behavioral models**

  Most software responds to events from the outside world. A computer program always exists in some state (e.g., waiting, computing, printing, and polling) that is changed only when some event occurs. For example, software will remain in the wait state until (1) an internal clock indicates that some time interval has passed, (2) an external event (e.g., a mouse movement) causes an interrupt, or (3) an external system signals the software to act in some manner. A behavioral model creates a representation of the states of the software and the events that cause a software to change state. An example of such models is **State Transition Diagram (STD)**.

These models provide a software designer with information that can be translated to architectural, interface-, and component designs.

The requirements model must achieve three primary objectives: **(1)** to describe what the customer requires, **(2)** to establish a basis for the creation of a software design, and **(3)** to define a set of requirements that can be validated once the software is built.

2.8 Requirements Management

Requirements management is the process of understanding and controlling changes to system requirements. The requirements for large software systems are always changing because problems solving by these systems can't be completely defined from the first time. So that after defining requirements (initial requirements), stakeholders specify additional user and system requirements and these requirements must be added. Requirements engineering must add these new requirements and specify relations between them and existing requirements and update the existing requirements to accommodate these relations. A formal process must be established for making change proposals and linking these to system requirements.

3.1 Introduction

Software design is a creative activity in which software components and their relationships are identified based on a customer's requirements. **Implementation** is the process of realizing the design as a program.

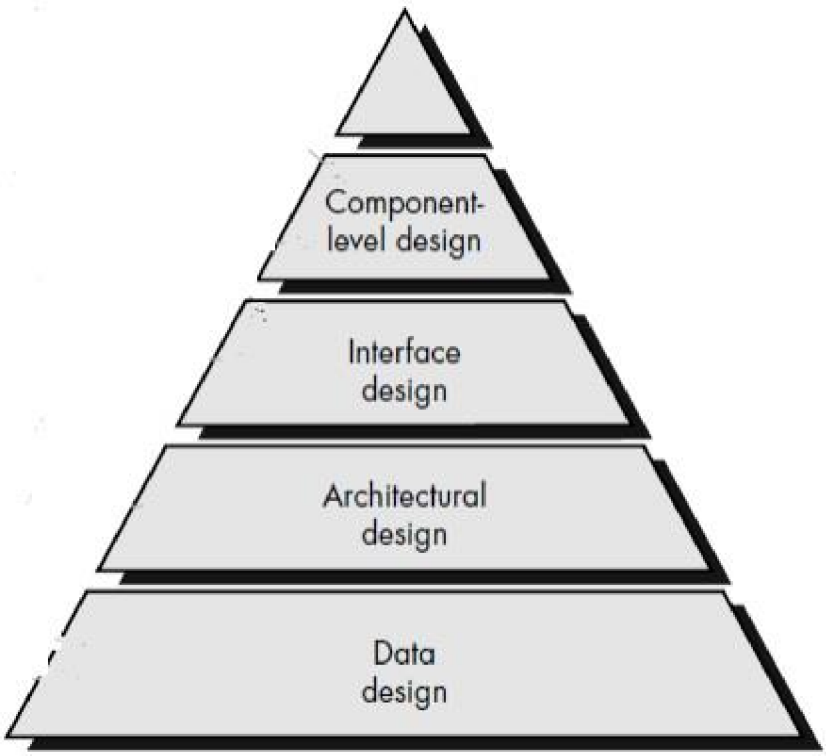


Figure 3.1: Design model.

Design model consists of the following steps:

1) **Data design.**
2) **Architectural design.**
3) **Interface design.**
4) **Component design**

The characteristics of a good design are:

1) The design should implement all explicit requirements contained in the requirements model, and it must accommodate all the implicit requirements desired by stakeholders.

2) The design should be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
3) The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

3.2 Design Concepts

Several concepts must be defined and understood before design the system model. Some of these concepts are:

1) Abstraction

There are two types of abstraction. 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.

2) Architecture

An architecture is the structure or organization of program components (modules), the ways in which these components interact, and the structure of data that are used by the components.

3) Modularity

Software is divided into separately named and addressable components, sometimes called **modules** that are integrated to satisfy problem requirements. Given the same set of requirements, the more modules used in your program means smaller individual sizes. However, as the number of modules grows, the effort (cost) associated with integrating modules with each other grows. There is a number, M, of modules that would result in minimum development cost, but we can't predict M with assurance.

The design of the system and the resulting program are modularized so that development can be more easily planned, software increments can be defined and delivered, changes can be more easily accommodated, testing and debugging can

be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

4) Information hiding

Modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

Information hiding is useful 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, errors introduced during modification are less likely to propagate to other locations within the software.

5) Functional independence

Functional independence is achieved by developing modules with "single-minded" function and little interaction with other modules. Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.

Independence is assessed using two qualitative criteria: cohesion and coupling. **Cohesion** is an indication of the relative functional strength of a module. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program.

Coupling is an indication of interconnections 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.

6) Stepwise refinement

Stepwise refinement is a top-down design strategy. Refinement is a process of elaboration. You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement

describes function or information conceptually but provides no indication of the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more detail as each successive refinement (elaboration) occurs.

7) **Refactoring**

Refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

3.3 Design Principles

1) Design should be traceable to the requirements model.
2) Always consider the architecture of the system to be built.
3) Design of data is as important as design of processing functions.
4) Interfaces (both internal and external) must be designed with care.
5) User interface design should be tuned to the needs of the end user.
6) Component-level design should be functionally independent.
7) Components should be loosely coupled to one another and to the external environment.
8) Design representations (models) should be easily understandable.
9) The design should be developed iteratively.

3.4 Data Design

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 or 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.

3.5 Architectural Design

Architectural design is concerned with understanding how a system should be organized and designing the overall structure of that system such that the design will satisfy the functional and non-functional requirements of a system. It identifies the main structural components in a system and the relationships between them. The activities within the process depend on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components. The architectural design representation is derived from the requirements model.

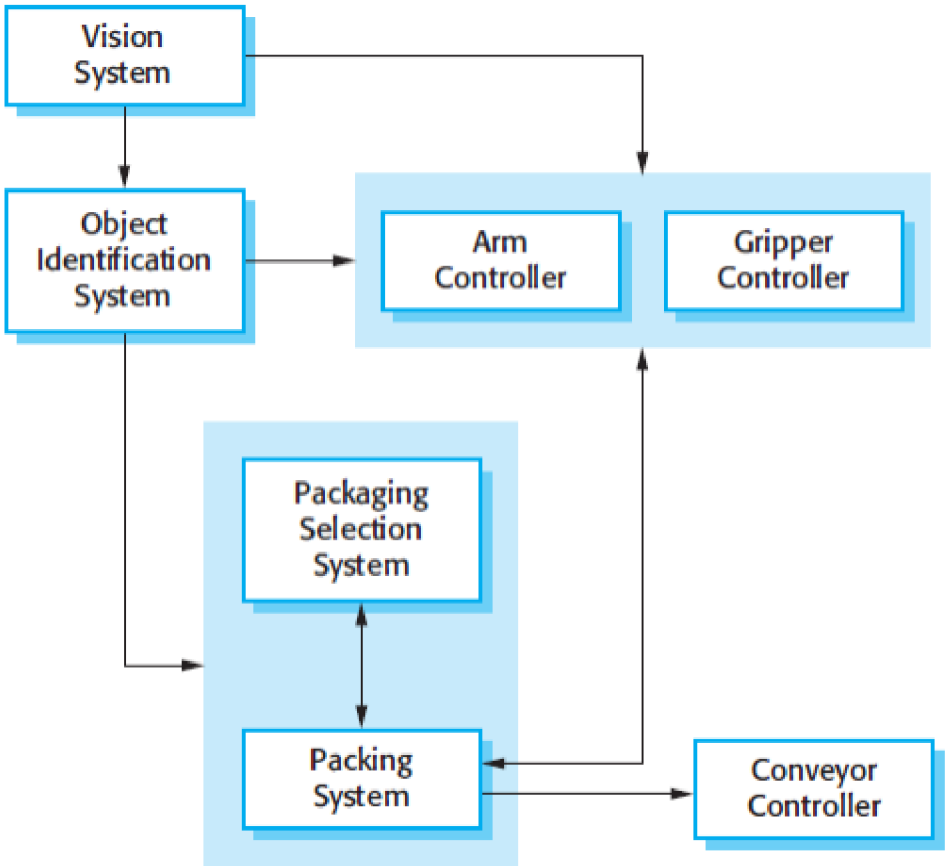


Figure 3.2: The architecture of a packing robot control system.

Figure 3.2 shows an abstract model of the architecture for a packing robot system that shows the components that have to be developed. This robotic system can pack different kinds of object. It uses a vision component to pick out objects on a conveyor, identify the type of object, and select the right kind of packaging. The system then moves objects from the delivery conveyor to be packaged. It places packaged objects on another conveyor. The architectural model shows these components and the links between them.

System architectures are often modeled using simple block diagrams, as in Figure 3.2. Each box in the diagram represents a component. Boxes within boxes indicate that the component has been decomposed to sub-components. Arrows mean that data and or control signals are passed from component to component in the direction of the arrows.

- How will the system be distributed across a number of cores or processors?
- What architectural styles might be used?
- How will the structural components in the system be decomposed into subcomponents?
- What strategy will be used to control the operation of the components in the system?
- What architectural organization is best for delivering the non-functional requirements of the system?
- How will the architectural design be evaluated?
- How should the architecture of the system be documented?

3.5.1 Architectural Styles

An **architectural style** is a transformation that is imposed on the design of an entire system. 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.
4) Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its parts.

There are many styles, some of them are:

1) Layered architecture

The system is organized into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.

This style is used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.

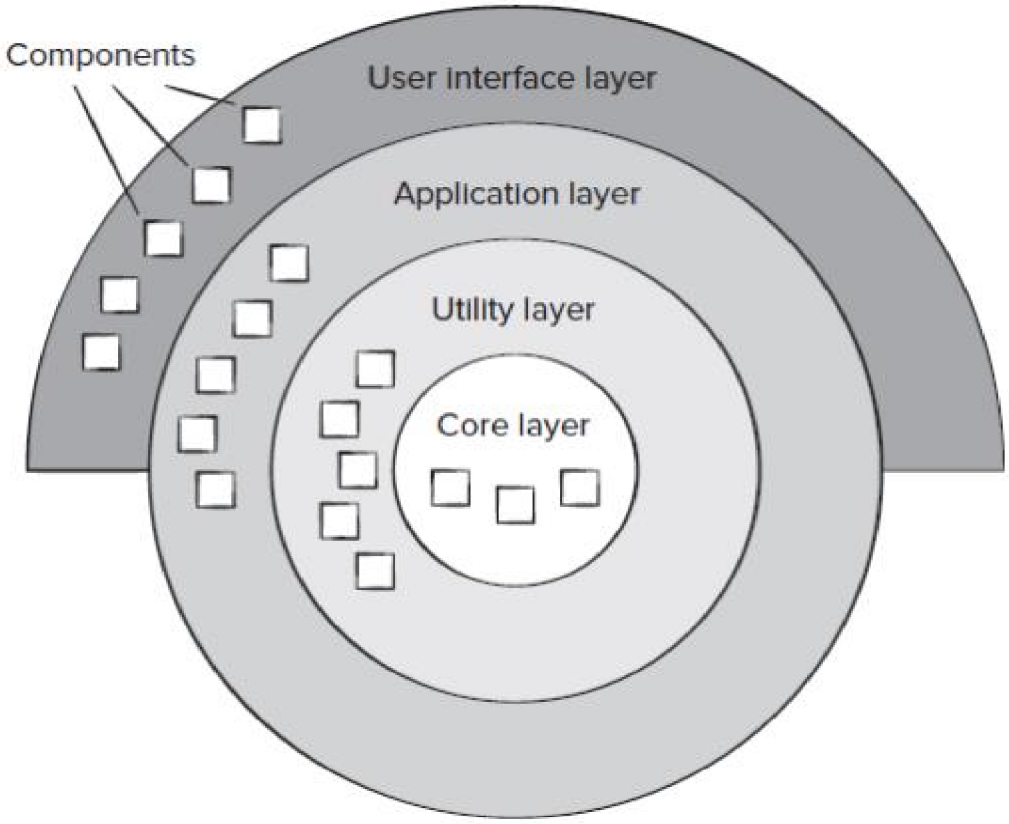


Figure 3.3: Layered Architecture.

Advantages of this style are: it allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.

Disadvantage of this style is that in practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it.

The basic structure of a layered architecture is illustrated in Figure 3.3. A number of different layers are defined. 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.

2) Data-Centered architecture

A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 3.4 illustrates a typical data-centered style. Client software accesses a central **repository**.

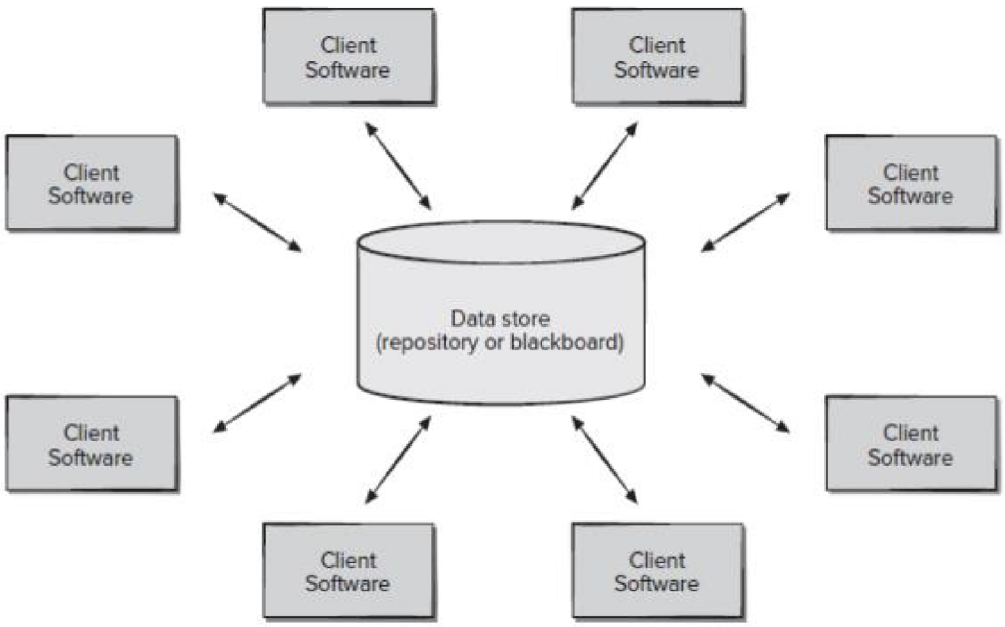


Figure 3.4: Data-Centered architecture.

In some cases, the data repository is passive and control is the responsibility of the components using the repository. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "**blackboard**" that sends notifications to client software when data of interest to the client changes.

This style is used when the system has a large volumes of information that has to be stored for a long time.

Advantages of this style are: components can be independent—they do not need to know of the existence of other components and they independently execute processes. Changes made by one component can be propagated to all components.

The **disadvantage** of this style is that the repository is a single point of failure so problems in the repository affect the whole system and distributing the repository across several computers may be difficult.

3) **Data-Flow architecture**

In this style (also called **pipe and filter**), a series of functional transformations process their inputs and produce outputs. A pipe-and-filter pattern (Figure 3.5) has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of other filters in the architecture and is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters. The transformations may execute sequentially or in parallel.

This style is commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.

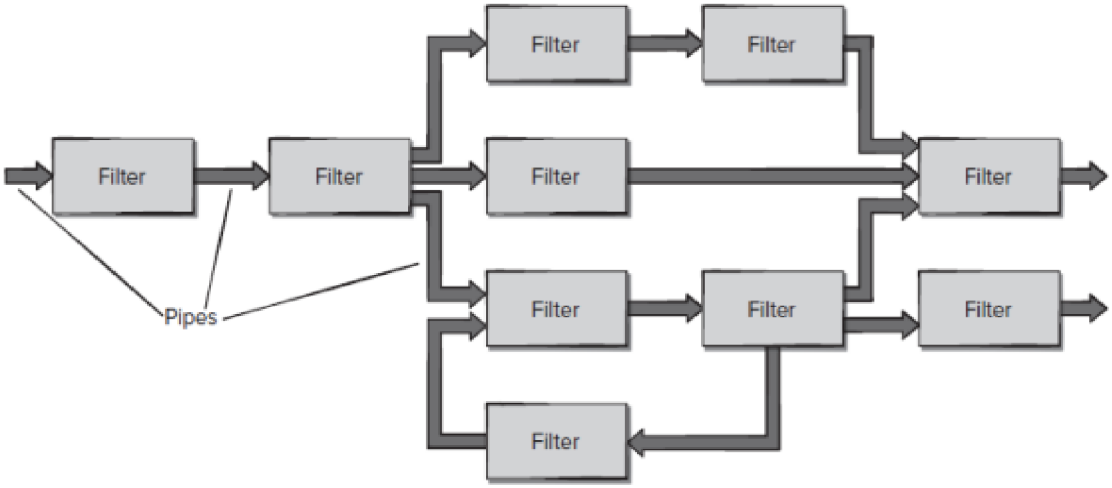


Figure 3.5: Pipe-and-filter architecture.

3.6 Interface Design

The principles for interface design are:

1) Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
2) Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, or voice recognition commands.
3) Allow user interaction to be interruptible and undoable. The user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action or any linear sequence of actions.
4) Hide technical internals from the casual user.
5) Design for direct interaction with objects that appear on the screen. For example, an application interface that allows a user to drag a document into the "trash" is an implementation of direct manipulation.

The analysis and design process for user interfaces is iterative. The user interface analysis and design process consists of four activities: (1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation.

Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. The information gathered as part of the analysis action is used to create an analysis model for the interface.

The goal of **interface design** is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

Interface construction normally begins with the creation of a prototype that represent user requirements. A user interface tool kit may be used to complete the construction of the interface.

Interface validation focuses on **(1)** the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements; **(2)** the degree to which the interface is easy to use and easy to learn, and **(3)** the user's acceptance of the interface as a useful tool in her work.

3.7 Component Design

Component design will reduce the number of errors introduced during coding. A **component** (also called module) 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.

During component design, each module is elaborated. The module interface is defined explicitly. That is, each data or control object that flows across the interface is represented. The data structures that are used internal to the module are defined. The algorithm that allows the module to accomplish its intended function is designed using the stepwise refinement.

Information from requirements and architectural models must be transformed into a design representation that provides sufficient detail to guide the construction (coding and testing) activity.

3.7.1 Cohesion

There are several types of cohesion. These types are (ordered from low to high): **Coincidental** cohesion: a module that performs a set of tasks that relate to each other loosely. **Logical** cohesion: a module that performs tasks that are related logically (e.g., a module that produces all output regardless of type). **Temporal** cohesion: a module contains tasks that are related by the fact that all must be executed with the same span of time. **Procedural** cohesion: processing elements of a module are related and must be executed in a specific order. **Communicational** cohesion: all processing elements concentrate on one area of a data structure. **High** cohesion: is characterized by a module that performs one distinct procedural task. We always strive for high cohesion.

3.7.2 Coupling

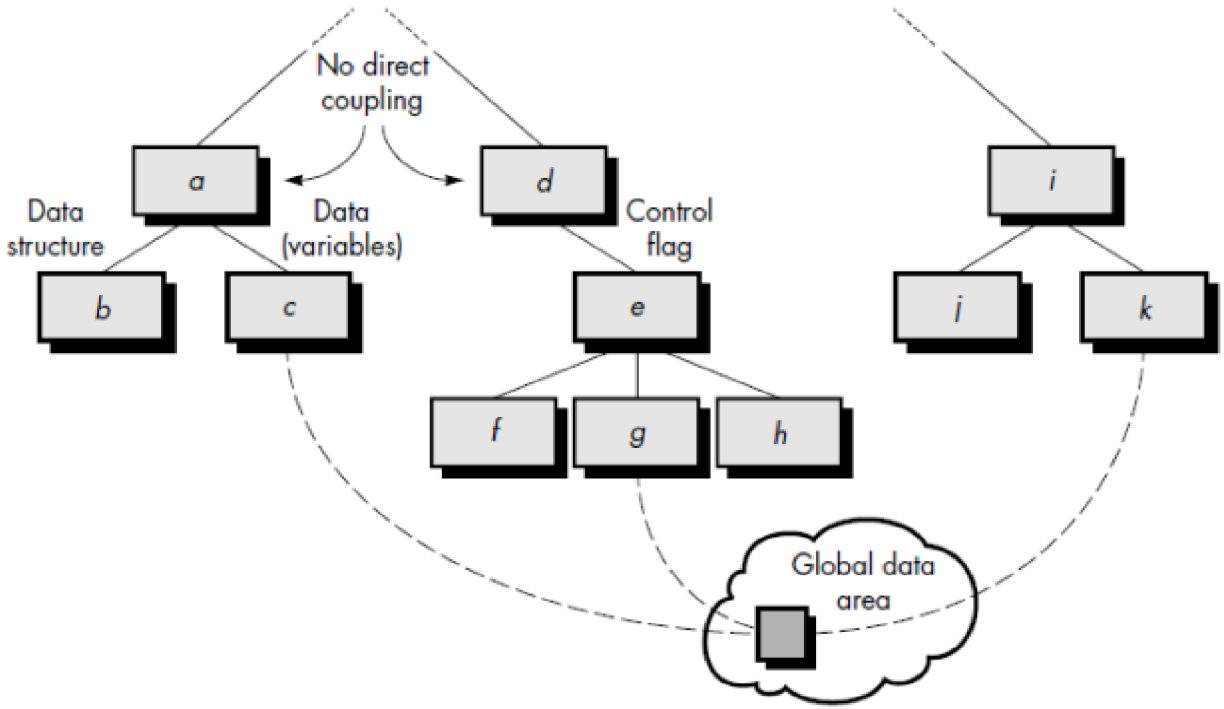


Figure 3.6: Types of coupling.

Figure 3.6 provides examples of different types of module coupling. These types are (ordered from low to high): **No direct** coupling (Modules a and d are subordinate to different modules and each is unrelated). **Data** coupling: Module c is subordinate to module a and is accessed via a simple argument list, through which data are passed (i.e., simple data are passed; a one-to-one correspondence of items exists). **Stamp** coupling: a portion of a data structure (rather than simple arguments) is passed via a module interface (between modules b and a). **Control** coupling: passage of control between modules (the pass of the control flag between modules d and e). **External** coupling: modules are tied to an environment external to software. For example, I/O couples a module to specific devices, formats, and communication protocols. **Common** coupling: a number of modules reference a global data area (Modules c, g, and k each access a data item in a global data area (e.g., a disk file or a globally accessible memory area)). **Content** coupling: occurs when one module makes use of data or control information maintained within the boundary of another module or when branches are made into the middle of a module.

The coupling modes just discussed occur because of design decisions made when structure was developed. Variants of **external** coupling, however, may be introduced during coding. For example, **compiler** coupling ties source code to specific (and often nonstandard) attributes of a compiler; **operating system** (OS) coupling ties design and resultant code to operating system that can create problems when OS changes occur. We always strive for low coupling.

3.7.3 Structured Programing

Any program, regardless of application area or technical complexity, can be designed and implemented using only three structured constructs. The constructs are sequence, condition, and repetition. **Sequence** implements processing steps that are essential in the specification of any algorithm. **Condition** provides the facility for selected processing based on some logical occurrence, and **repetition** allows for looping. These three constructs are fundamental to structured programming—an

important component-level design technique. The use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability.

These constructs can be represented in two different ways: flowchart and box diagram. A **flowchart** is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of control. The structured constructs may be nested within one another as shown in Figure 3.8.

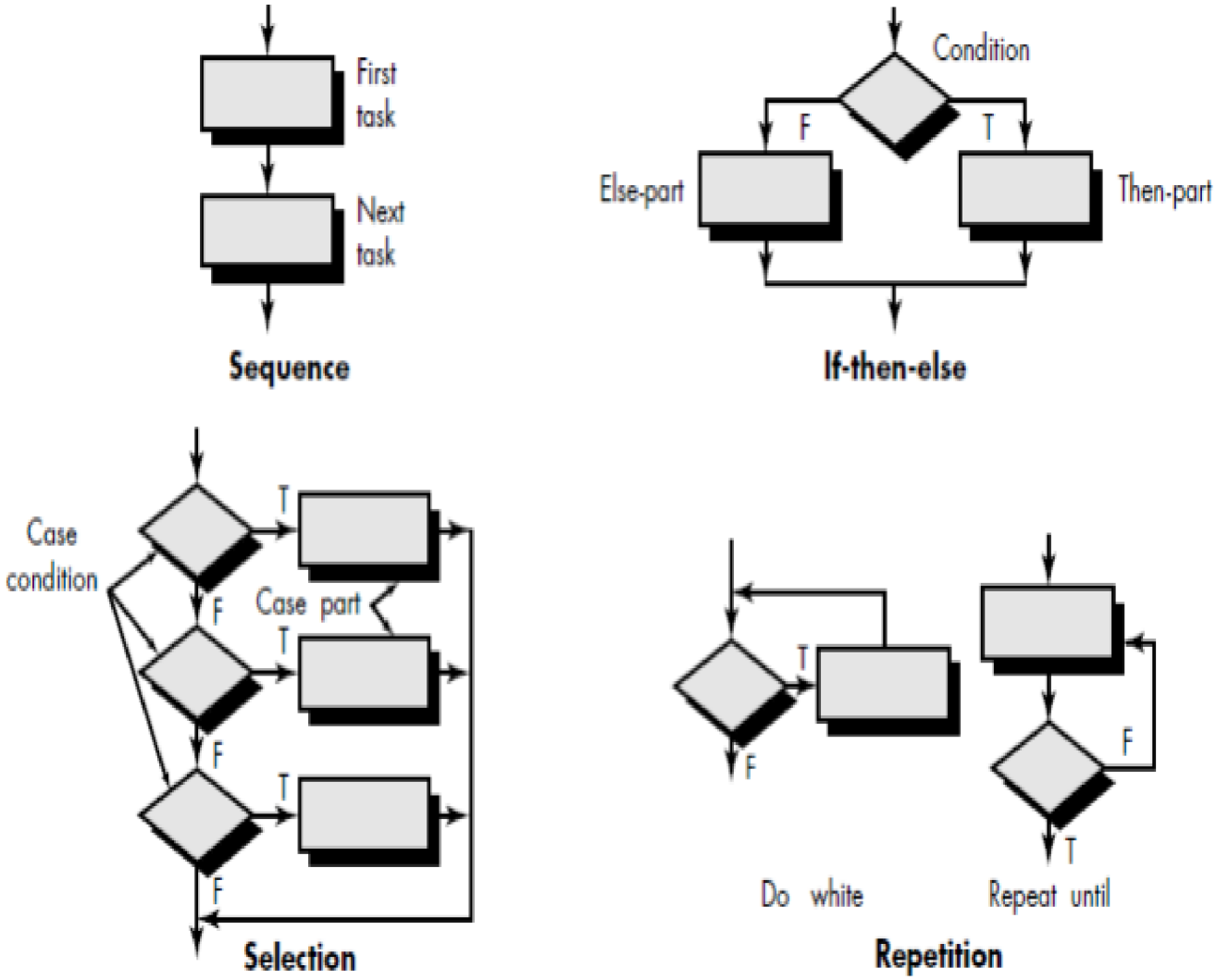


Figure 3.7: Flowchart constructs.

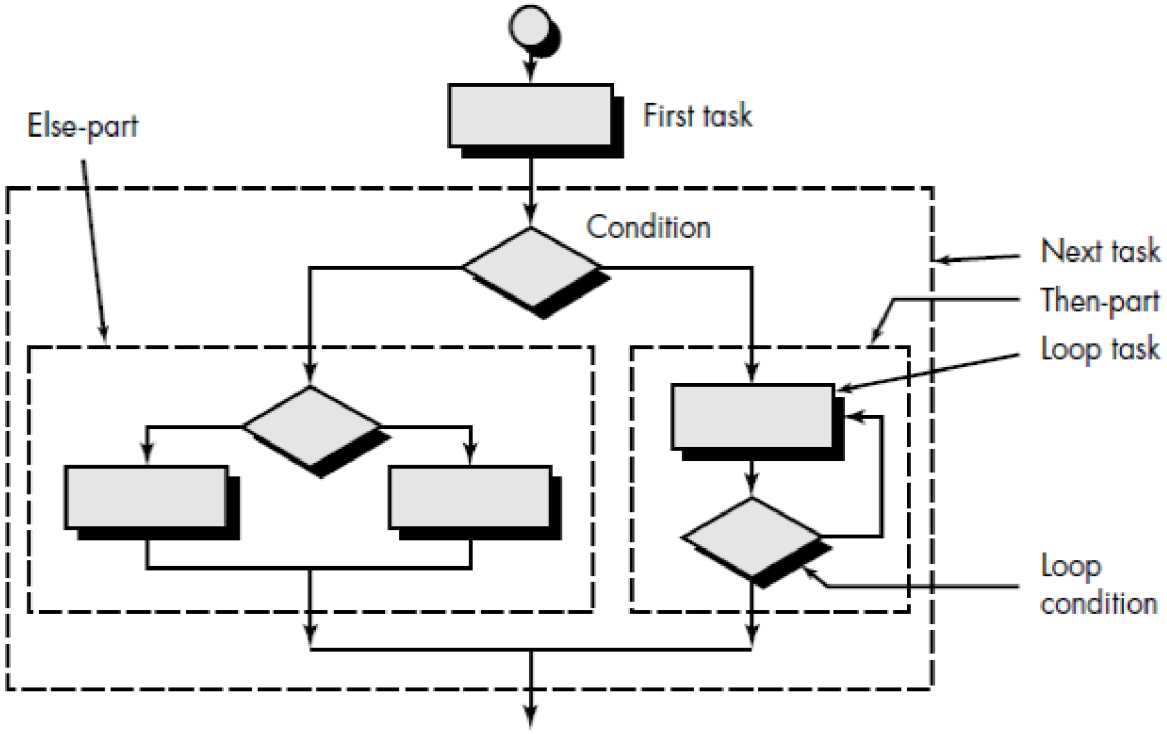


Figure 3.8: Nested constructs.

Box diagram (also called Nassi-Shneiderman charts, N-S charts, or Chapin charts) have the following characteristics:

1) Functional domain (that is, the scope of repetition or if-then-else) is well defined and clearly visible as a pictorial representation.
2) Arbitrary transfer of control is impossible.
3) The scope of local and/or global data can be easily determined.
4) Recursion is easy to represent.

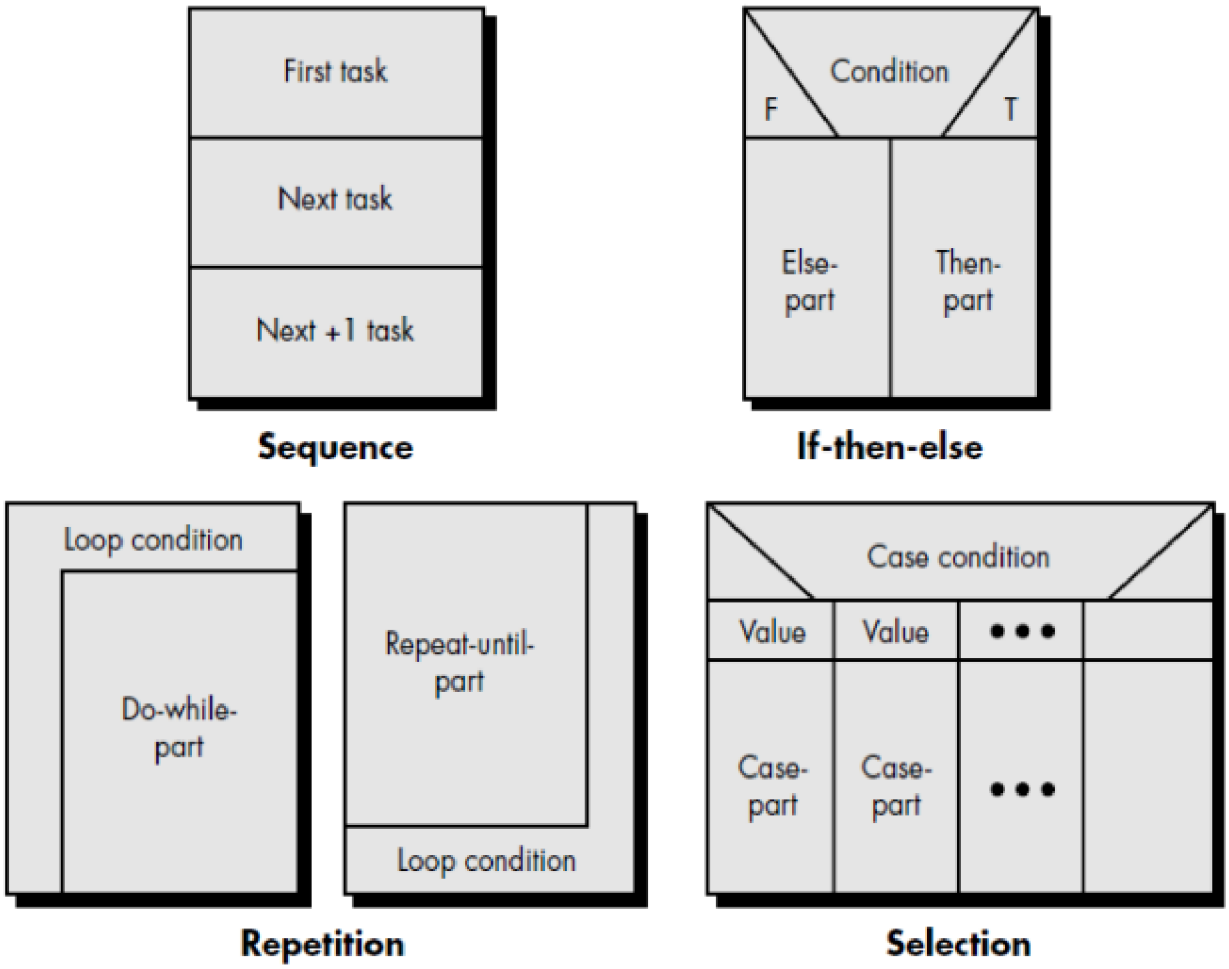


Figure 3.9: Box diagram constructs.

3.7.4 Decision Table

In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions. Decision tables provide a notation that translates actions and conditions into a tabular form. Decision table organization is illustrated in Figure 3.10.

	Rules								
Conditions	**1**	**2**	**3**	**4**					n
Condition #1	✓			✓	✓				
Condition #2		✓		✓					
Condition #3			✓		✓				
Actions									
Action #1	✓			✓	✓				
Action #2		✓		✓					
Action #3			✓						
Action #4			✓	✓	✓				
Action #5	✓	✓			✓				

Figure 3.10: Decision table structure.

The following steps are applied to develop a decision table:

1) List all actions that can be associated with a specific procedure (or module).
2) List all conditions (or decisions made) during execution of the procedure.
3) Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
4) Define rules by indicating what action(s) occurs for a set of conditions.

To illustrate the use of a decision table, consider the following excerpt from a processing narrative for a public utility billing system: If the customer account is billed using a fixed rate method, a minimum monthly charge is assessed for consumption of less than 100 KWH (kilowatt-hours). Otherwise, computer billing applies a Schedule A rate structure. However, if the account is billed using a variable rate method, a Schedule A rate structure will apply to consumption below 100 KWH, with additional consumption billed according to Schedule B.

Figure 3.11 illustrates a decision table representation of the preceding narrative.

	Rules				
Conditions	**1**	**2**	**3**	**4**	**5**
Fixed rate acct.	T	T	F	F	F
Variable rate acct.	F	F	T	T	F
Consumption <100 kwh	T	F	T	F	
Consumption ≥100 kwh	F	T	F	T	
Actions					
Min. monthly charge	✓				
Schedule A billing		✓	✓		
Schedule B billing				✓	
Other treatment					✓

Figure 3.11: Resultant decision table.

4.1 Introduction

Testing is a process of executing a program with the intent of finding an error. Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). **Verification** refers to the set of tasks that ensure that software correctly implements a specific function. **Validation** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Verification: "Are we building the product right?"

Validation: "Are we building the right product?"

The attributes of a "good" test are:

1) A good test has a high probability of finding an error.
2) A good test is not redundant. Testing time and resources are limited.
3) A good test should be "best of breed". In a group of tests that have a similar intent, time and resource limitations may dictate the execution of only those tests that have the highest likelihood of uncovering a whole class of errors.
4) A good test should be neither too simple nor too complex. Each test should be executed separately.

Software quality is an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.

Figure 4.1 shows software quality factors. These factors are:

1) **Correctness**: the extent to which a program satisfies its specification and fulfills the customer's mission objectives.
2) **Reliability**: the extent to which a program can be expected to perform its intended function with required precision.
3) **Efficiency**: the amount of computing resources and code required by a program to perform its function.

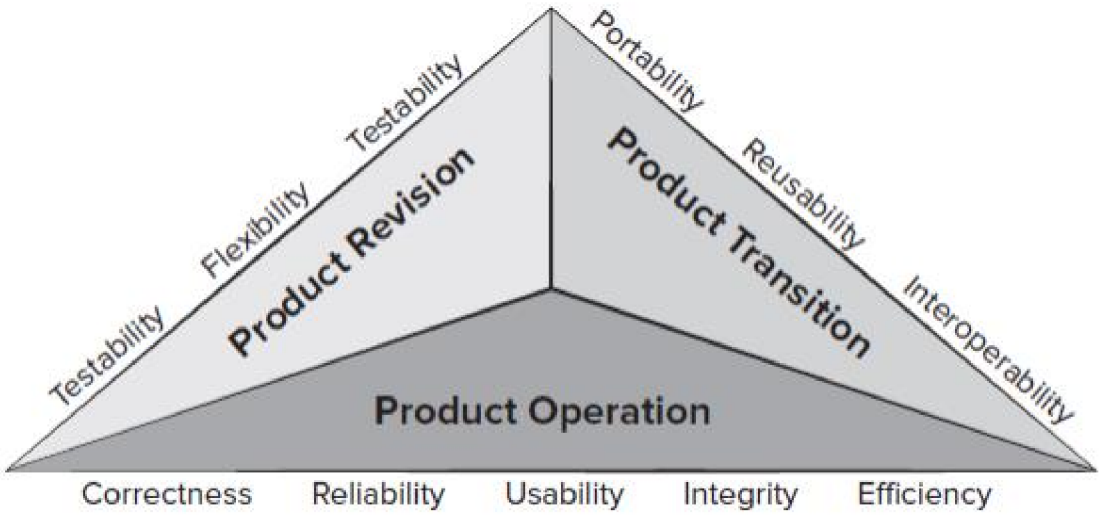


Figure 4.1: Quality factors.

4) **Integrity**: extent to which access to software or data by unauthorized persons can be controlled.
5) **Usability**: effort required to learn, operate, prepare input, and interpret output of a program.
6) **Maintainability**: effort required to locate and fix an error in a program.
7) **Flexibility**: effort required to modify an operational program.
8) **Testability**: effort required to test a program to ensure that it performs its intended function.
9) **Portability**: effort required to transfer the program from one hardware and/or software system environment to another.
10) **Reusability**: extent to which a program [or parts of a program] can be reused in other applications.
11) **Interoperability**: effort required to couple one system to another.

A **formal technical review (FTR)** is a software quality control activity performed by software engineers (and others). The objectives of an FTR are:

1) To uncover errors in function, logic, or implementation for any representation of the software.
2) To verify that the software under review meets its requirements.

3) To ensure that the software has been represented according to predefined standards.
4) To achieve software that is developed in a uniform manner.
5) To make projects more manageable.

4.2 Software Testing Strategy

There are four activities in this strategy as shown in figure 4.2.

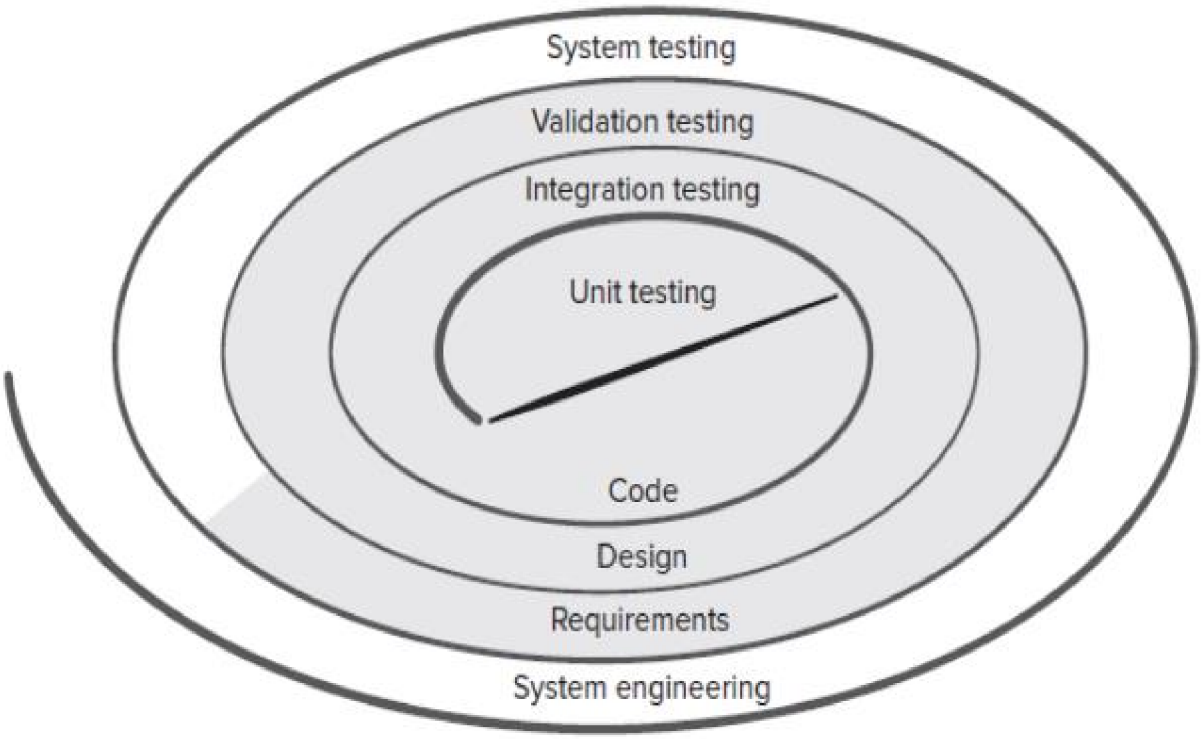


Figure 4.2: Software testing strategy.

Unit testing concentrates on each unit (component) of the software as implemented in source code. In **integration** testing, the focus is on design and the construction of the software architecture. In **validation** testing, requirements established as part of requirements modeling are validated against the software that has been constructed and it ensures that software meets all functional, behavioral, and performance requirements. In **system** testing, the software and other system elements (e.g., hardware, people, and databases) are tested as a whole.

4.3 Unit Testing

In unit testing, the module interface is tested to ensure that information properly flows into and out of the program unit under test. Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error-handling paths are tested.

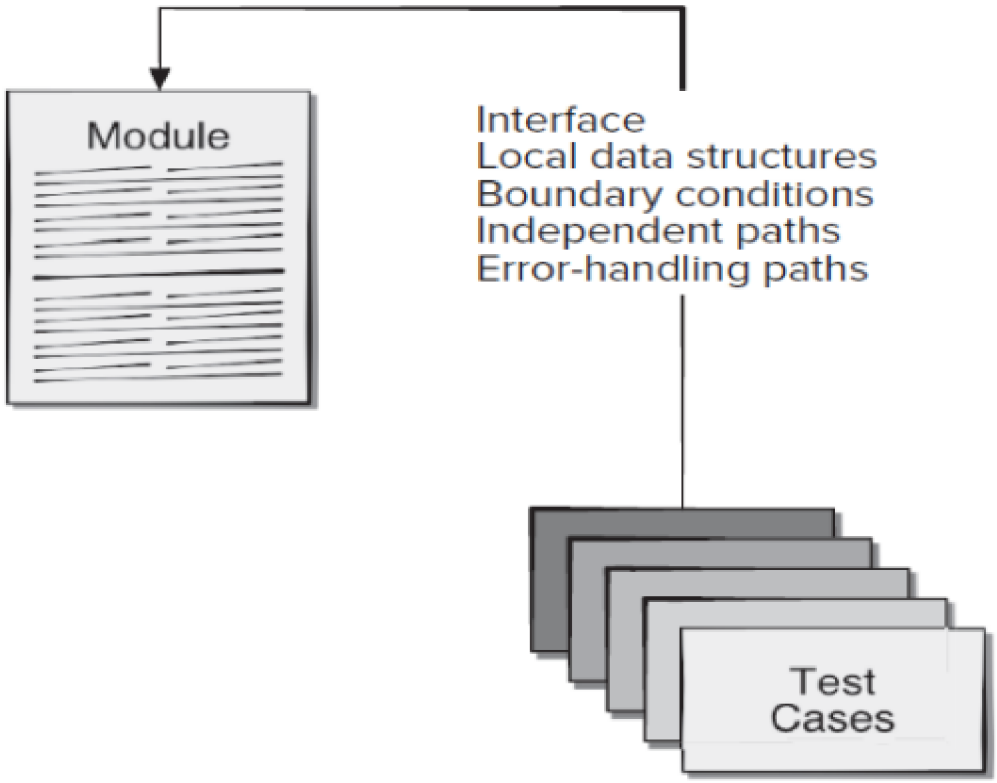


Figure 4.3: Unit testing.

Among the potential errors that should be tested when error handling is evaluated are:

1) Error description is unintelligible.
2) Error noted does not correspond to error encountered.
3) Error condition causes system intervention prior to error handling.

4) Exception-condition processing is incorrect.
5) Error description does not provide enough information to assist in the location of the cause of the error.

Any engineered product can be tested in one of two ways: (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function (this is called **black-box testing**). (2) Knowing the internal workings of a product, tests can be conducted to ensure that internal operations are performed according to specifications and all internal components have been adequately exercised (this is called **white-box testing**).

4.4 White Box Testing

White-box testing, sometimes called **glass-box** testing or **structural** testing is a test case design that uses the control structure described as part of component design to derive test cases. Using white-box testing methods, test cases are derived such that:

1) Guarantee that all independent paths within a module have been exercised at least once.
2) Exercise all logical decisions on their true and false sides.
3) Execute all loops at their boundaries and within their operational bounds.
4) Exercise internal data structures to ensure their validity.

4.4.1 Basis Path Testing

Basis path testing is a white-box testing technique. The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

Flow graph (or program graph) is a graph that consists of nodes and edges. Each node represents one or more procedural statements. Each edge represents flow of

control and must terminate at a node. Area bounded by edges and nodes is called region.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When considering flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. The value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program. It is computed in one of three ways:

1) The number of regions of the flow graph corresponds to the cyclomatic complexity.
2) Cyclomatic complexity V(G) for a flow graph G is defined as $V(G) = E - N + 2$
   Where E is the number of flow graph edges and N is the number of flow graph nodes.
3) Cyclomatic complexity V(G) for a flow graph G is also defined as $V(G) = P + 1$
   Where P is the number of predicate nodes contained in the flow graph G.

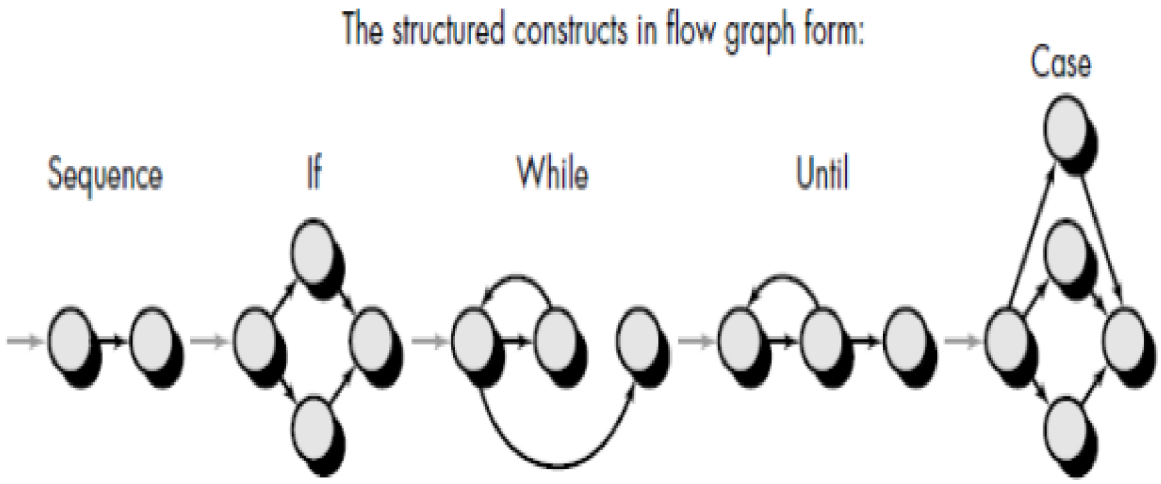


Figure 4.4: Flow graph notations.

A **graph matrix** is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column

correspond to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix is shown in Figure 4.5. A new matrix called **connection matrix** can derived from the graph matrix and used to compute cyclomatic complexity. This matrix is shown in figure 4.6 below:

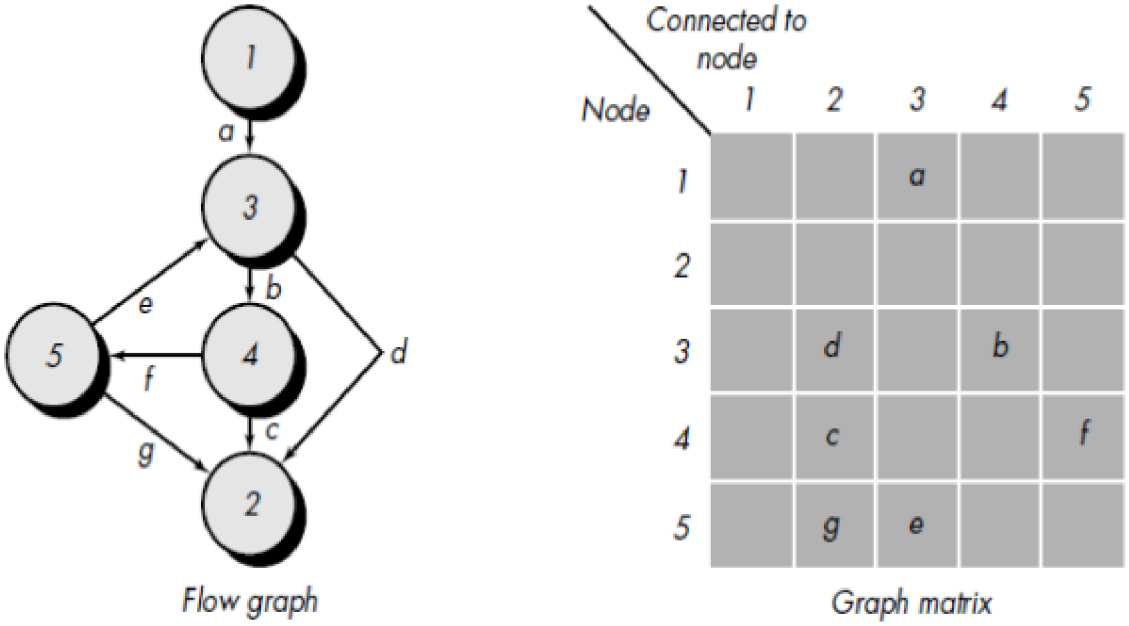


Figure 4.5: Graph matrix.

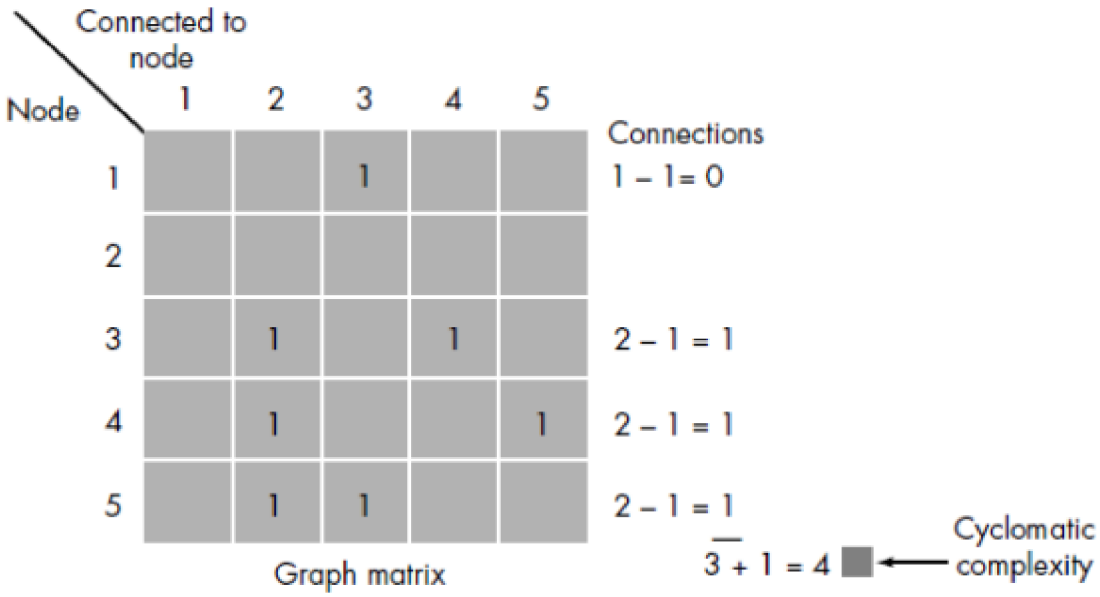


Figure 4.6: Connection matrix.

Example

Consider the following program:

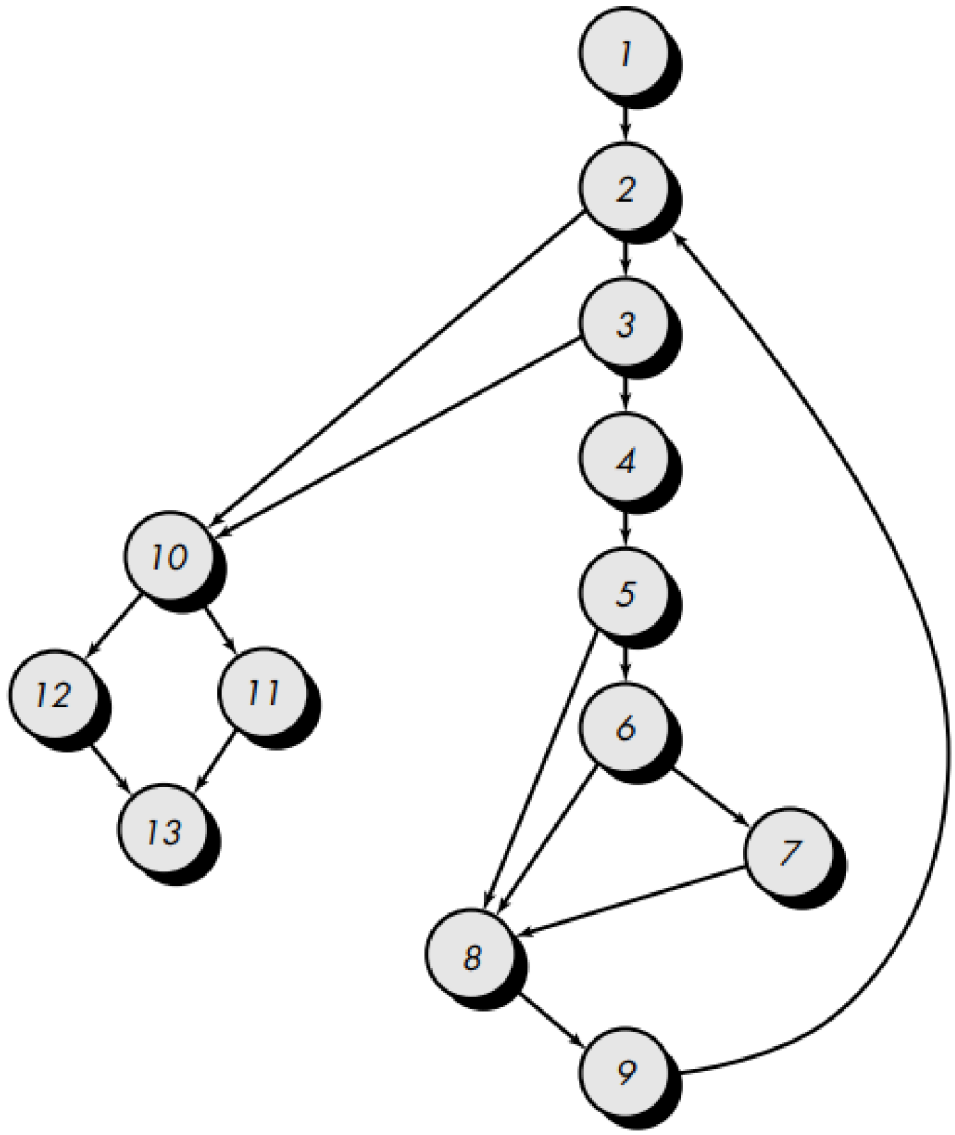
```
PROCEDURE average;

* This procedure computes the average of 100 or fewer
  numbers that lie between bounding values; it also computes the
  sum and the total number valid.

INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid;
   minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
   i = 1;
1  total.input = total.valid = 0;   2
   sum = 0;
   DO WHILE value[i] <> -999 AND total.input < 100   3
    4 increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum   6
   5      THEN increment total.valid by 1;
       7          sum = s sum + value[i]
          ELSE skip
      ENDIF
   8  increment i by 1;
   9 ENDDO
      IF total.valid > 0   10
     11 THEN average = sum / total.valid;
   12    ELSE average = -999;
    13 ENDIF
END average
```

1) Using the design or code as a foundation, draw a corresponding flow graph.



2) Determine the cyclomatic complexity of the resultant flow graph.

V(G) = 6 regions

V(G) = 17 edges – 13 nodes + 2 = 6

V(G) = 5 predicate nodes + 1 = 6

3) Determine a basis set of linearly independent paths.

path 1: 1-2-10-11-13

path 2: 1-2-10-12-13

path 3: 1-2-3-10-11-13

path 4: 1-2-3-4-5-8-9-2-. . .

path 5: 1-2-3-4-5-6-8-9-2-. . .

path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable.

4) Prepare test cases that will force execution of each path in the basis set.

Path 1 test case:

value(k) = valid input, where k < i for $2 \leq i \leq 100$

value(i) = - 999 where $2 \leq i \leq 100$

Expected results: Correct average based on k values and proper totals.

Note: Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

Path 2 test case:

value(1) = - 999

Expected results: Average = - 999; other totals at initial values.

Path 3 test case:

Attempt to process 101 or more values.

First 100 values should be valid.

Expected results: Same as test case 1.

Path 4 test case:

value(i) = valid input where i < 100

value(k) < minimum where k < i

Expected results: Correct average based on k values and proper totals.

Path 5 test case:

value(i) = valid input where i < 100

value(k) > maximum where k <= i

Expected results: Correct average based on n values and proper totals.

Path 6 test case:

value(i) = valid input where i < 100

Expected results: Correct average based on n values and proper totals.

4.4.2 Loop Testing

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Two different classes of loops can be defined: simple loops and concatenated loops (Figure 4.7).

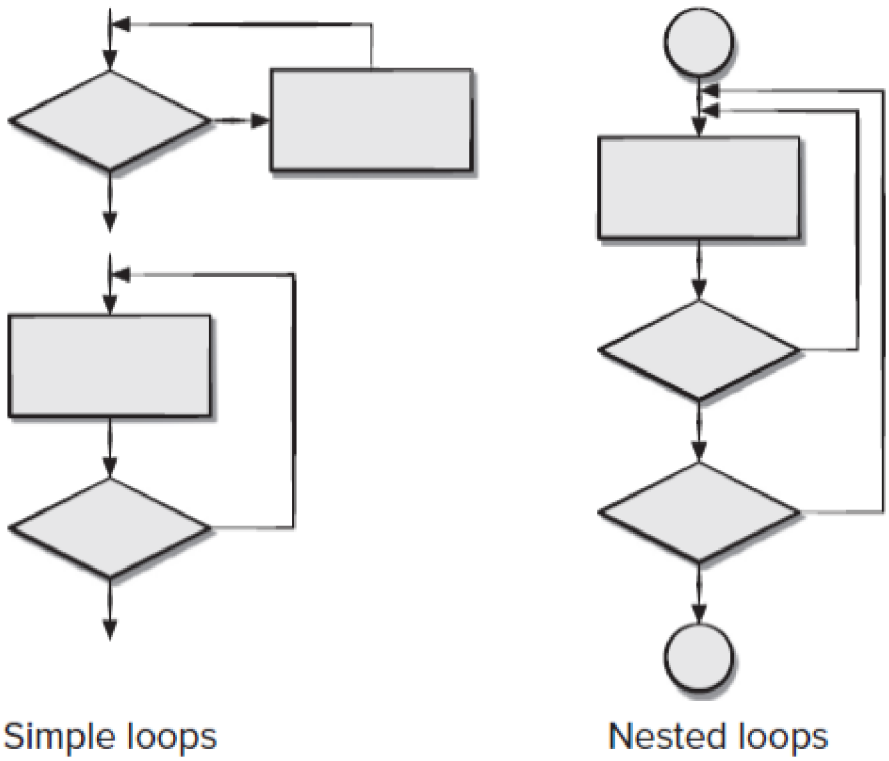


Figure 4.7: Classes of loops.

Simple Loops: the following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

5) Skip the loop entirely.
6) Only one pass through the loop.
7) Two passes through the loop.
8) m passes through the loop where $m < n$.
9) $n - 1$, n, $n + 1$ passes through the loop.

Nested Loops: the following set of tests can be applied to nested loops.

1) Start at the innermost loop. Set all other loops to minimum values.

2) Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3) Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4) Continue until all loops have been tested.

4.5 Black Box Testing

Black-box testing, also called **behavioral** testing or **functional** testing, focuses on the functional requirements of the software. That is, black-box testing techniques enable the tester to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white box methods.

Black-box testing attempts to find errors in the following categories:

1) Incorrect or missing functions.
2) Interface errors.
3) Errors in data structures or external database access.
4) Behavior or performance errors.
5) Initialization and termination errors.

4.5.1 Equivalence Partitioning

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. Test-case design for equivalence partitioning is based on an evaluation of **equivalence classes** for an input condition. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an **input condition** is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

1) If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2) If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3) If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4) If an input condition is Boolean, one valid and one invalid class are defined.

4.5.2 Boundary Value Analysis

A greater number of errors occurs at the boundaries of the input domain rather than in the "center." Therefore, boundary value analysis (BVA) has been developed as a testing technique. Boundary value analysis is a test-case design technique that complements equivalence partitioning. BVA selects test cases at the "edges" of the class. BVA derives test cases from both input and the output domains. Guidelines for BVA are:

1) If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2) If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3) Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4) If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

4.6 Integration Testing

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.

4.6.1 Top-Down Integration

Top-down integration testing is an incremental approach to construction of the software architecture. Modules (components) are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

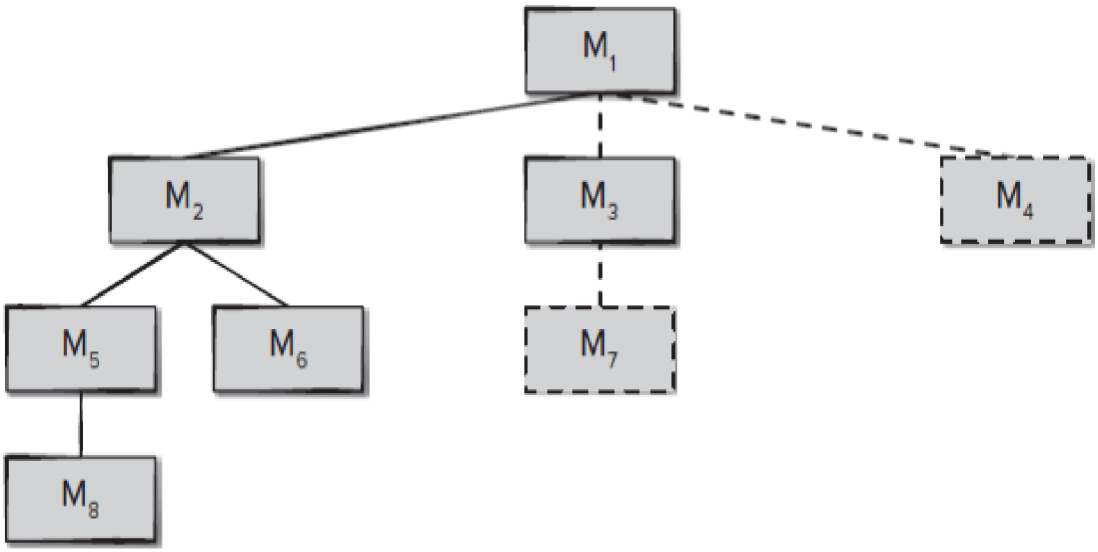


Figure 4.8: Top-Down integration.

Referring to Figure 4.8, **depth-first** integration integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components M1, M2, M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central

and right-hand control paths are built. **Breadth-first** integration incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.

4.6.2 Bottom-Up Integration

Bottom-up integration testing begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).

A bottom-up integration strategy may be implemented with the following steps:

1) Low-level components are combined into clusters (sometimes called builds) that perform a specific software sub-function.
2) A driver (a control program for testing) is written to coordinate test-case input and output.
3) The cluster is tested.
4) Drivers are removed and clusters are combined, moving upward in the program structure.

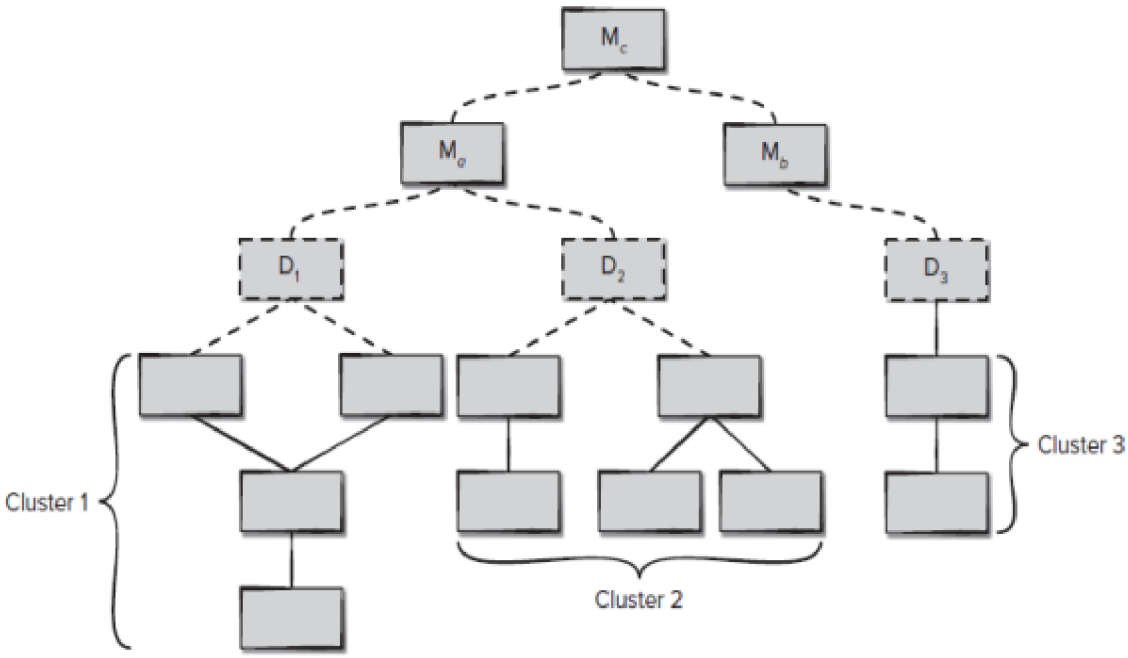


Figure 4.9: Bottom-Up integration.

Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to Ma. Drivers D1 and D2 are removed, and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

4.6.3 Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new input/output (I/O) may occur, and new control logic is invoked. Side effects associated with these changes may cause problems.

Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. Regression tests should be executed every time a major change is made to the software (including the integration of new components). Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

4.6.4 Interface Testin

Interface testing is used to check that the program component accepts information passed to it in the proper order and data types and returns information in proper order and data format.

There are different types of interfaces between program components and, consequently, different types of interface error that can occur:

1) **Parameter** interfaces: these are interfaces in which data or sometimes function references are passed from one component to another.

2) **Shared memory** interfaces: these are interfaces in which a block of memory is shared between components. Data is placed in the memory by one subsystem and retrieved from there by other sub-systems.
3) **Procedural interfaces**: these are interfaces in which one component encapsulates a set of procedures that can be called by other components.
4) **Message passing** interfaces: these are interfaces in which one component requests a service from another component by passing a message to it. A return message includes the results of executing the service.

4.7 Validation Testing

Validation tries to uncover errors, but the focus is at the requirements level—on things that will be immediately apparent to the end user. Validation succeeds when software functions in a manner that can be reasonably expected by the customer.

Reasonable expectations are defined in the Software Requirements Specification. The specification contains a section called **Validation Criteria**. Information contained in that section forms the basis for a validation testing approach.

A process called alpha and beta testing is used to uncover errors that only the end-user seems able to find. The **alpha test** is conducted at the developer's site by a customer. The developer records errors. Therefore, Alpha tests are conducted in a controlled environment. The **beta test** is conducted at one or more customer sites by the end-user of the software. Therefore, Beta tests are conducted in a non-controlled environment. The customer records all problems and reports these to the developer at regular intervals.

4.8 System Testing

Software is only one element of a larger computer-based system. Software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration are conducted to test interactions between software and these elements. At this stage, the software engineer should:

1) Design error-handling paths that test all information coming from other elements of the system.
2) Conduct a series of tests that simulate bad data or other potential errors at the software interface.
3) Record the results of tests.
4) Participate in planning and design of system tests to ensure that software is tested.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness.

4.9 Debugging

Debugging is the process of fixing errors and problems that have been discovered by testing. Using information from the program tests, debuggers use their knowledge of the programming language and the intended outcome of the test to locate and repair the program error. This process is often supported by interactive debugging tools that provide extra information about program execution.

5.1 Introduction

Software development does not stop when a system is delivered but continues throughout the lifetime of the system. After a system has been deployed, it has to change if it is to remain useful.

System change proposals are the driver for system evolution in all organizations. Change proposals may come from existing requirements that have not been implemented in the released system, requests for new requirements, bug reports from system stakeholders, and new ideas for software improvement from the system development team. Change proposals should be linked to the components of the system that have to be modified to implement these proposals.

5.2 Evolution Process

The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers. If the proposed changes are accepted, a new release of the system is planned. During release planning, all proposed changes (fault repair, adaptation, and new functionality) are considered. A decision is then made on which changes to implement in the next version of the system. The changes are implemented and validated, and a new version of the system is released. The process then iterates with a new set of changes proposed for the next release.

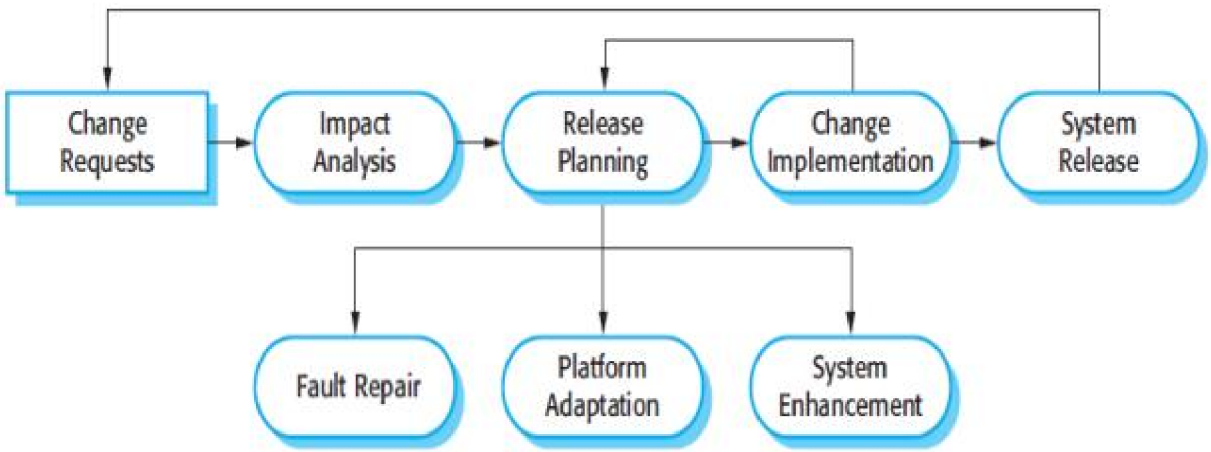


Figure 5.1: Software evolution process.

The **change implementation** stage of this process should modify the system specification, design, and implementation to reflect the changes to the system. New requirements that reflect the system changes are proposed, analyzed, and validated. System components are redesigned and implemented and the system is retested.

5.3 Software Maintenance

Software maintenance is the general process of changing a system after it has been delivered. Changes are implemented by modifying existing system components and, where necessary, by adding new components to the system. There are three different types of software maintenance:

1) **Fault repairs** (also called **corrective maintenance**): coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve rewriting several program components. Requirements errors are the most expensive to repair because of the extensive system redesign which may be necessary.
2) **Environmental adaptation** (also called **adaptive maintenance**): this type of maintenance is required when some aspect of the system's environment such as the hardware, the platform operating system, or other support software changes. The application system must be modified to adapt it to cope with these environmental changes.
3) **Functionality addition** (also called **perfective maintenance**): this type of maintenance is necessary when the system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.

5.4 Software Reengineering

To make legacy software systems easier to maintain, these systems can be reengineered to improve their structure and understandability. Reengineering may involve re-documenting the system, refactoring the system architecture, translating

programs to a modern programming language, and modifying and updating the structure and values of the system's data. The functionality of the software is not changed and, normally, you should try to avoid making major changes to the system architecture.

Software reengineering consists of six activities:

1) Inventory analysis: it is a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application.
2) Document restructuring.
3) Reverse engineering.
4) Code restructuring.
5) Data restructuring.
6) Forward engineering.

Reverse engineering for software is the process of analyzing a program to create a representation of the program at a higher level of abstraction than source code. Reverse engineering is a process of **design recovery**. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

5.5 Refactoring

Refactoring is the process of making improvements to a program to slow down degradation through change. It means modifying a program to improve its structure, to reduce its complexity, or to make it easier to understand. When you refactor a program, you should not add functionality but should concentrate on program improvement. You can therefore think of refactoring as '**preventative maintenance**' that reduces the problems of future change.

- **Data refactoring**

  Before data refactoring, a reverse engineering activity called **source code analysis** should be performed. All programming language statements that contain

data definitions, file descriptions, I/O, and interface descriptions are evaluated. The intent is to (1) extract data items and objects, (2) get information on data flow, and (3) understand the existing data structures that have been implemented. This activity is sometimes called data analysis.

In data refactoring, data definitions are clarified and physical modifications to existing data structures are made to make the data design more effective. This may mean a translation from one file format to another, or in some cases, translation from one type of database to another.

- **Code refactoring**

  Code refactoring is performed to yield a design that produces the same function but with higher quality than the original program. The objective is derive a design that conforms to the quality factors. It is also used to reduce coupling and improve cohesion.

- **Architecture refactoring**

  When module is too large, it must be divided into several modules which will cause changes in the architectural model.