4. Analysis. It is highly desirable if some important properties about the be- havior of the system can be determined before the system is actually built. This will allow the designers to consider alternatives and select the one that will best suit the needs. Many engineering disciplines use models to analyze design of a product for its cost, reliability, performance, etc. Archi- tecture opens such possibilities for software also.

Not all of these uses may be significant in a project and which of these uses is pertinent to a project depends on the nature of the project. In some projects communication may be very important, but a detailed performance analysis may be unnecessary (because the system is too small or is meant for only a few users). In some other systems, performance analysis may be the primary use of architecture.

Architecture Views

There is a general view emerging that there is no unique architecture of a system. The definition that we have adopted (given above) also expresses this sentiment. Consequently, there is no one architecture drawing of the system. The situation is similar to that of civil construction, a discipline that is the original user of the concept of architecture and from where the concept of software architecture has been borrowed. For a building, if you want to see the floor plan, you are shown one set of drawings. If you are an electrical engineer and want to see how the electricity distribution has been planned, you will be shown another set of drawings. And if you are interested in safety and firefighting, another set of drawings is used. These drawings are not independent of each other—they are all about the same building.

A view describes a structure of the system. We will use these two concepts— views and structures—interchangeably. We will also use the term architectural view to refer to a view. Many types of views have been proposed. Most of the proposed views generally belong to one of these three types

_ Module
– Component and connector
– Allocation

In **a module view**, the system is viewed as a collection of code units, each implementing some part of the system functionality. That is, the main elements in this view are modules. These views are code-based and do not explicitly rep- resent any runtime structure of the system. Examples of modules are packages, a class, a procedure, a method, a collection of functions, and a collection of classes. The relationships between these modules are also code-based and de- pend on how code of a module interacts with another module

component and connector (C&C) view, the system is viewed as a col- lection of runtime entities called components. That is, a component is a unit which has an identity in the executing system. Objects (not classes), a collec- tion of objects, and a process are examples of components. While executing, components need to interact with others to support the system services. Con- nectors provide means for this interaction. Examples of connectors are pipes and sockets. Shared data can also act as a connector. If the components use some middleware to communicate and coordinate, then the middleware is a connector.

allocation view focuses on how the different software units are allocated to resources like the hardware, file systems, and people. That is, an allocation view specifies the relationship between software elements and elements of the environments in which the software system is executed. They expose structural properties like which processes run on which processor, and how the system files are organized on a file system.

An architecture description consists of views of different types, with each view exposing some structure of the system. Module views show how the soft- ware is structured as a set of implementation units, C&C views show how the software is structured as interacting runtime elements, and allocation views show how software relates to nonsoftware structures. These three types of view of the same system form the architecture of the system

Component and Connector View

The C&C architecture view of a system has two main elements—components and connectors. Components are usually computational elements or data stores that have some presence during the system execution. Connectors define the means of interaction between these components. A C&C view of the system de- fines the components, and which component is connected to which and through what connector. A C&C view describes a runtime structure of the system— what components exist when the system is executing and how they interact during the execution. The C&C structure is essentially a graph, with compo- nents as nodes and connectors as edges. The C&C view is perhaps the most common view of architecture and most box-and-line drawings representing architecture attempt to capture this view. Most often when people talk about the architecture, they refer to the C&C view. Most architecture description languages also focus on the C&C view.

5.3.1 Components

Components are generally units of computation or data stores in the system. A component has a name, which is generally chosen to represent the role of the component or the function it performs. The name also provides a unique identity to the component, which is necessary for referencing details about the component in the supporting documents, as a C&C drawing will only show the component names.

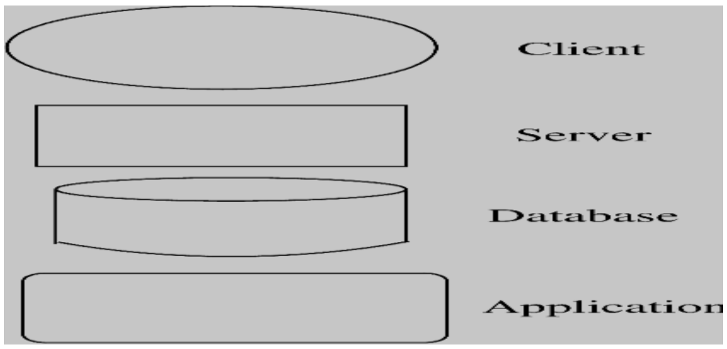

Figure 5.1: Component examples.

A component is of a component type, where the type represents a generic component, defining the general computation and the interfaces a component of that type must have. Note that though a component has a type, in the C&C architecture view, we have components (i.e., actual instances) and not types. Examples of these types are clients, servers, filters, etc. Different domains may have other generic types like controllers, actuators, and sensors (for a control system domain).

In a diagram representing a C&C architecture view of a system, it is highly desirable to have a different representation for different component types, so the different types can be identified visually. In a box-and-line diagram, often all components are represented as rectangular boxes. Such an approach will require that types of the components are described separately and the reader has to read the description to figure out the types of the components. It is much better to use a different symbol/notation for each different component type. Some of the common symbols used for representing commonly found component types are shown in Figure 5.1.

To make sure that the meanings of the different symbols are clear to the reader, it is desirable to have a key of the different symbols to describe what type of component asymbol represents.

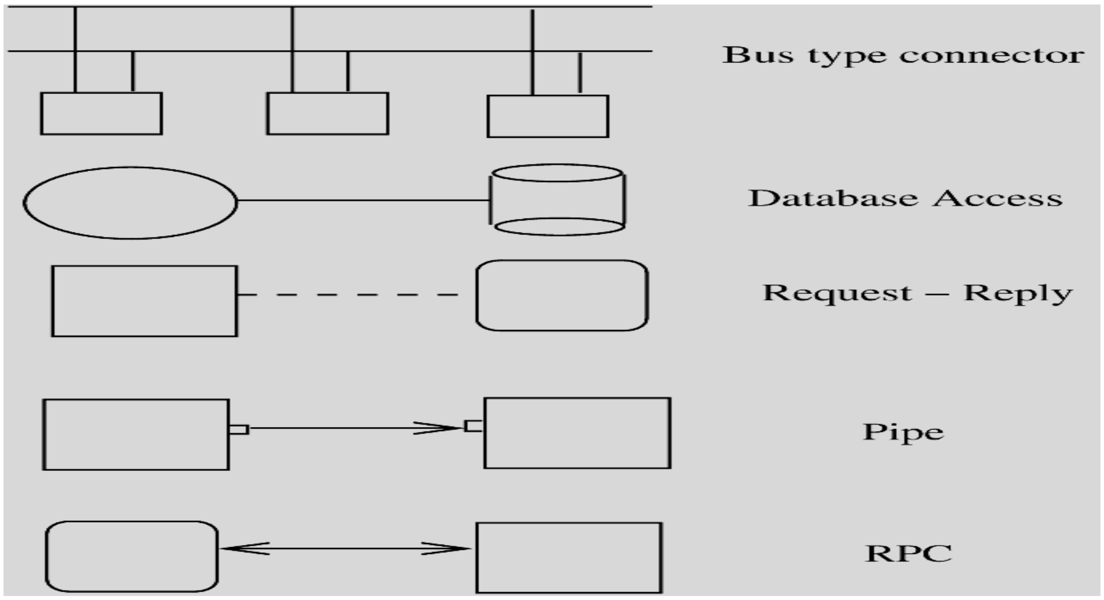

Figure 5.2: Connector examples.

5.3.2 Connectors

The different components of a system are likely to interact while the system is in operation to provide the services expected of the system. After all, components exist to provide parts of the services and features of the system, and these must be combined to deliver the overall system functionality. For composing a system from its components, information about the interaction between components is necessary.

Architecture Styles for C&C View

It should be clear that different systems will have different architecture. There are some general architectures that have been observed in many systems and that seem to represent general structures that are useful for architecture of a class of problems. These are called architectural styles. A style defines a family of architectures that satisfy the constraints of that style [6, 23, 76]. In this section we discuss some common styles for the C&C view which can be useful for a large set of problems [23, 76]. These styles can provide ideas for creating an architecture view for the problem at hand. Styles can also be combined to form richer views.

5.4.1 Pipe and Filter

Pipe-and-filter style of architecture is well suited for systems that primarily do data transformation whereby some input data is received and the goal of the system is to produce some output data by suitably transforming the input data. A system using pipe-and-filter architecture achieves the desired transformation by applying a network of smaller transformations and composing them in a manner such that together the overall desired transformation is achieved.

The pipe-and-filter style has only one component type called the filter. It also has only one connector type, called the pipe. A filter performs a data transformation, and sends the transformed data to other filters for further processing using the pipe connector. In other words, a filter receives the data it needs from some defined input pipes, performs the data transformation, and then sends the output data to other filters on the defined output pipes. A filter may have more than one input and more than one output. Filters can be independent and asynchronous entities, and as they are concerned only with the data arriving on the pipe, a filter need not know the identity of the filter that sent the input data or the identity of the filter that will consume the data they produce.

The pipe connector is a unidirectional channel which conveys streams of data received on one end to the other end. A pipe does not change the data in any manner but merely transports it to the filter on the receiver end in theorder in which the data elements are received. As filters can be asynchronous and should work without the knowledge of the identity of the producer or the consumer, buffering and synchronization needs to ensure smooth functioning of the producer-consumer relationship embodied in connecting two filters by a pipe is ensured by the pipe. The filters merely consume and produce data

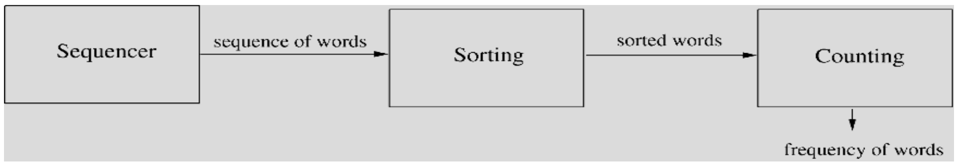

Figure 5.6: Pipe-and-filter example.

Shared-Data Style

In this style, there are two types of components—data repositories and data accessors. Components of data repository type are where the system stores shared data—these could be file systems or databases. These components pro- vide a reliable and permanent storage, take care of any synchronization needs for concurrent access, and provide data access support. Components of data accessors type access data from the repositories, perform computation on the data obtained, and if they want to share the results with other components, put the results back in the depository. In other words, the accessors are computa- tional elements that receive their data from the repository and save their data in the repository as well. These components do not directly communicate with each other—the data repository components are the means of communication and data transfer between them.

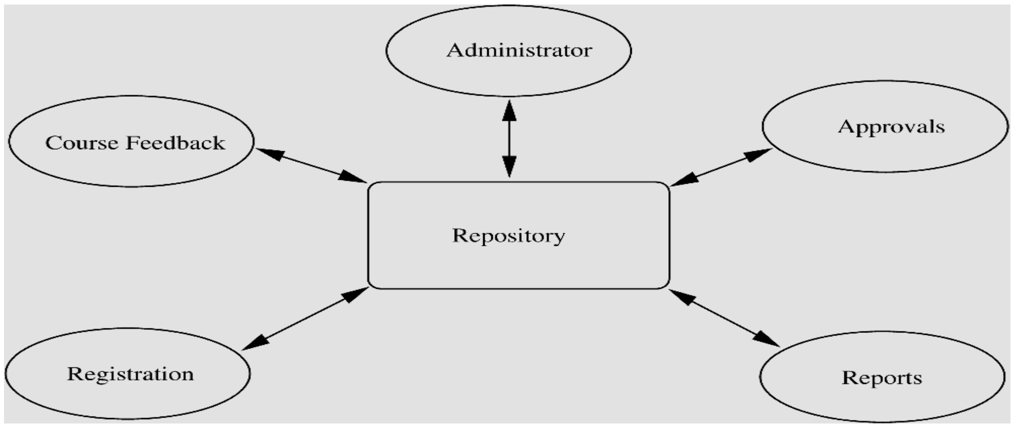

Figure 5.7: Shared data example.

Client-Server Style

Another very common style used to build systems today is the client-server style. Client-server computing is one of the basic paradigms of distributed com- puting and this architecture style is built upon this paradigm.

In this style, there are two component types—clients and servers. A con- straint of this style is that a client can only communicate with the server, and cannot communicate with other clients. The communication between a client component and a server component is initiated by the client when the client sends a request for some service that the server supports. The server receives the request at its defined port, performs the service, and then returns the results of the computation to the client who requested the service.

Some Other Styles

Publish-Subscribe Style In this style, there are two types of components. One type of component subscribes to a set of defined events. Other types of compo- nents generate or publish events. In response to these events, the components that have published their intent to process the event, are invoked. This type of style is most natural in user interface frameworks, where many events are defined (like mouse click) and components are assigned to these events. When that event occurs, the associated component is executed. As is the case with most connectors, it is the task of the runtime infrastructure to ensure that this type of connector (i.e., publish-subscribe) is supported. This style can be seen as a special case of the blackboard style, except that the repository aspect is not being used.

Peer-to-peer style, or object-oriented style If we take a client-server style, and generalize each component to be a client as well as a server, then we have this style. In this style, components are peers and any component can request a service from any other component. The object-oriented computation model represents this style well. If we view components as objects, and connectors as method invocations, then we have this style. This model is the one that is primarily supported through middleware connectors like CORBA or .NET.

Communicating processes style Perhaps the oldest model of distributed com- puting is that of communicating processes. This style tries to capture this model of computing. The components in this model are processes or threads, which communicate with each other either with message passing or through shared memory. This style is used in some form in many complex systems which use multiple threads or processes.

5.5 Documenting Architecture Design

So far we have focused on representing views through diagrams. While de- signing, diagrams are indeed a good way to explore options and encourage discussion and brainstorming between the architects. But when the designing is over, the architecture has to be properly communicated to all stakehold- ers for negotiation and agreement. This requires that architecture be precisely documented with enough information to perform the types of analysis the dif- ferent stakeholders wish to make to satisfy themselves that their concerns have been adequately addressed. Without a properly documented description of the architecture, it is not possible to have a clear common understanding. Hence, properly documenting an architecture is as important as creating one. In this section, we discuss what an architecture document should contain. Our discus- sion is based on the recommendations in [6, 23, 54].

Just like different projects require different views, different projects will need different level of detail in their architecture documentation. In general, however, a document describing the architecture should contain the following:

– System and architecture contex
– Description of architecture views
– Across views documentation

We know that an architecture for a system is driven by the system objectives and the needs of the stakeholders. Hence, the first aspect that an architecture document should contain is identification of stakeholders and their concerns. This portion should give an overview of the system, the different stakeholders, and the system properties for which the architecture will be evaluated. A con- text diagram that establishes the scope of the system, its boundaries, the key actors that interact with the system, and sources and sinks of data can also be very useful. A context diagram is frequently represented by showing the system in the center, and showing its connections with people and systems, including sources and sinks of data.

Evaluating Architectures

Architecture of a software system impacts some of the key nonfunctional qual- ity attributes like modifiability, performance, reliability, portability, etc. The architecture has a much more significant impact on some of these properties than the design and coding choices. That is, even though choices of algorithms, data structures, etc., are important for many of these attributes, often they have less of an impact than the architectural choices. Clearly then, evaluating a proposed architecture for these properties can have a beneficial impact on the project—any architectural changes that are required to meet the desired goals for these attributes can be done during the architecture design itself.

Design Concepts

The design of a system is correct if a system built precisely according to the design satisfies the requirements of that system. Clearly, the goal during the design phase is to produce correct designs. However, correctness is not the sole criterion during the design phase, as there can be many correct designs. The goal of the design process is not simply to produce a design for the system. Instead, the goal is to find the best possible design within the limitations im- posed by the requirements and the physical and social environment in which the system will operate.

A software system cannot be made modular by simply chopping it into a set of modules. For modularity, each module needs to support a well-defined abstraction and have a clear interface through which it can interact with other modules. To produce modular designs, some criteria must be used to select modules so that the modules support well-defined abstractions and are solv- able and modifiable separately. Coupling and cohesion are two modularization criteria, which are often used together. We also discuss the open-closed princi- ple, which is another criterion for modularity.

Coupling

Two modules are considered independent if one can function completely with- out the presence of the other. Obviously, if two modules are independent, they are solvable and modifiable separately. However, all the modules in a system cannot be independent of each other, as they must interact so that together they produce the desired external behavior of the system. The more connections between modules, the more dependent they are in the sense that more knowl- edge about one module is required to understand or solve the other module. Hence, the fewer and simpler the connections between modules, the easier it is to understand one without understanding the other. The notion of coupling [79, 88] attempts to capture this concept of "how strongly" different modules are interconnected.

Coupling between modules is the strength of interconnections between mod- ules or a measure of interdependence among modules. In general, the more we must know about module A in order to understand module B, the more closely connected A is to B. "Highly coupled" modules are joined by strong interconnections, while "loosely coupled" modules have weak interconnections. Independent modules have no interconnections. To solve and modify a module separately, we would like the module to be loosely coupled with other mod- ules. The choice of modules decides the coupling between modules. Because the modules of the software system are created during system design, the coupling between modules is largely decided during system design and cannot be reduced during implementation.

Coupling increases with the complexity and obscurity of the interface be- tween modules. To keep coupling low we would like to minimize the number of interfaces per module and the complexity of each interface. An interface of a module is used to pass information to and from other modules. Coupling is reduced if only the defined entry interface of a module is used by other modules, for example, passing information to and from a module exclusively through pa- rameters. Coupling would increase if a module is used by other modules via an indirect and obscure interface, like directly using the internals of a module or using shared variables.
Complexity of the interface is another factor affecting coupling. The more complex each interface is, the higher will be the degree of coupling. For example, complexity of the entry interface of a procedure depends on the number of items being passed as parameters and on the complexity of the items.
The type of information flow along the interfaces is the third major factor affecting coupling. There are two kinds of information that can flow along an interface: data or control. Passing or receiving control information means that the action of the module will depend on this control information, which makes it more difficult to understand the module and provide its abstraction

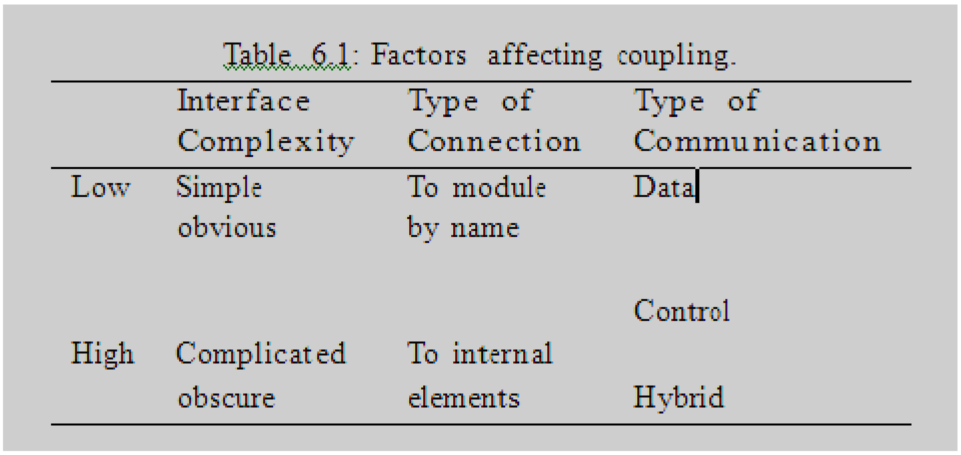
Table 6.1: Factors affecting coupling.

	Interface Complexity	Type of Connection	Type of Communication
Low	Simple obvious	To module by name	Data
			Control
High	Complicated obscure	To internal elements	Hybrid

The manifestation of coupling in OO systems is somewhat different as objects are semantically richer than functions. In OO systems, three different types of coupling exist between modules [30]:– Interaction coupling– Component coupling– Inheritance coupling

Interaction coupling occurs due to methods of a class invoking methods of other classes. In many ways, this situation is similar to a function calling an- other function and hence this coupling is similar to coupling between functional modules discussed above. Like with functions, the worst form of coupling here is if methods directly access internal parts of other methods.
Component coupling refers to the interaction between two classes where a class has variables of the other class. Three clear situations exist as to how this can happen. A class C can be component coupled with another class C1, if C has an instance variable of type C1, or C has a method whose parameter is of type C1, or if C has a method which has a local variable of type C1.
Inheritance coupling is due to the inheritance relationship between classes. Two classes are considered inheritance coupled if one class is a direct or indirect subclass of the other. If inheritance adds coupling, one can ask the question why not do away with inheritance altogether. The reason is that inheritance may reduce the overall coupling in the system

Cohesion

We have seen that coupling is reduced when the relationships among elements in different modules are minimized. That is, coupling is reduced when elements in different modules have little or no bonds between them. Another way of achieving this effect is to strengthen the bond between elements of the same module by maximizing the relationship between elements of the same module. Cohesion is the concept that tries to capture this intramodule

Cohesion of a module represents how tightly bound the internal elements of the module are to one another. Cohesion of a module gives the designer an idea about whether the different elements of a module belong together in the same module. Cohesion and coupling are clearly related. Usually, the greater the cohesion of each module in the system, the lower the coupling between modules is. This correlation is not perfect, but it has been observed in practice. There are several levels of cohesion:

– Coincidental– Logical– Temporal

– Procedural– Communicational

– Sequential– Functional

Coincidental is the lowest level, and functional is the highest. Coincidental cohesion occurs when there is no meaningful relationship among the elements of a module.
A module has **logical** cohesion if there is some logical relationship between the elements of a module, and the elements perform functions that fall in the same logical class.

Temporal cohesion is the same as logical cohesion, except that the elements are also related in time and are executed together.
A procedurally cohesive module contains elements that belong to a common procedural unit
A module with **communicational** cohesion has elements that are related by a reference to the same input or output data

If we have a sequence of elements in which the output of one forms the input to another, sequential cohesion does not provide any guidelines on how to combine them into modules. **Functional** cohesion is the strongest cohesion. In a functionally bound mod- ule, all the elements of the module are related to performing a single function

Cohesion in object-oriented systems has three aspects [30]:

– Method cohesion– Class cohesion– Inheritance cohesion

Method cohesion is the same as cohesion in functional modules. It focuses on why the different code elements of a method are together within the method. The highest form of cohesion is if each method implements a clearly defined function, and all statements in the method contribute to implementing this function.

Class cohesion focuses on why different attributes and methods are together in this class. The goal is to have a class that implements a single concept or abstraction with all elements contributing toward supporting this concept. In general, whenever there are multiple concepts encapsulated within a class, the cohesion of the class is not as high as it could be, and a designer should try to change the design to have each class encapsulate a single concept.

Inheritance cohesion focuses on the reason why classes are together in a hierarchy. The two main reasons for inheritance are to model generalization- specialization relationship, and for code reuse. Cohesion is considered high if the hierarchy supports generalization-specialization of some concept (which is likely to naturally lead to reuse of some code).

The Open-Closed Principle

This is a design concept which came into existence more in the OO context. Like with cohesion and coupling, the basic goal here is again to promote build- ing of systems that are easily modifiable, as modification and change happen frequently and a design that cannot easily accommodate change will result in systems that will die fast and will not be able to easily adapt to the changing world.

The basic principle, as stated by Bertrand Meyer, is “Software entities should be open for extension, but closed for modification”[66]. A module being “open for extension” means that its behavior can be extended to accommodate new demands placed on this module due to changes in requirements and sys- tem functionality. The module being “closed for modification” means that the existing source code of the module is not changed when making enhancements.

Then how does one make enhancements to a module without changing the existing source code? This principle restricts the changes to modules to extension only, i.e. it allows addition of code, but disallows changing of existing code. If this can be done, clearly, the value is tremendous. Code changes involve heavy risk and to ensure that a change has not “broken” things that were working often requires a lot of regression testing. This risk can be minimized if no changes are made to existing code. But if changes are not made, how will enhancements be made? This principle says that enhancements should be made by adding new code, rather than altering old code.

This principle can be satisfied in OO designs by properly using inheritance and polymorphism. Inheritance allows creating new classes that will extend the behavior of existing classes without changing the original class. And it is this property that can be used to support this principle. As an example, consider an application in which a client object (of type Client) interacts with a printer object (of class Printer1) and invokes the necessary methods for completing its printing needs. The class diagram for this will be as shown in Figure 6.1.

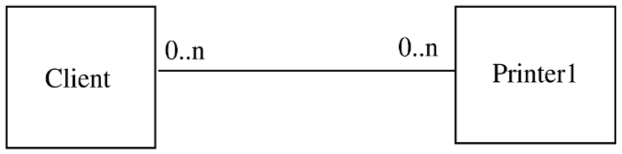

Figure 6.1: Example without using subtyping.

In this design, the client directly calls the methods on the printer object for printing something. Now suppose the system has to be enhanced to allow another printer to be used by the client. Under this design, to implement this change, a new class Printer2 will have to be created and the code of the client class will have to be changed to allow using object of Printer2 type as well. This design does not support the open-closed principle as the Client class is not closed against change.

Function-Oriented Design

Creating the software system design is the major concern of the design phase. Many design techniques have been proposed over the years to provide some discipline in handling the complexity of designing large systems. The aim of design methodologies is not to reduce the process of design to a sequence of mechanical steps but to provide guidelines to aid the designer during the design process. We discuss the structured design methodology [79, 88] for developing function-oriented system designs. The methodology employs the structure chart notation for creating the design. So before we discuss the methodology, we describe this notation

Structure Charts

Graphical design notations are frequently used during the design process to represent design or design decisions, so the design can be communicated to stakeholders in a succinct manner and evaluated. For a function-oriented design, the design can be represented graphically by structure charts.

The structure of a program is made up of the modules of that program together with the interconnections between modules. Every computer program has a structure, and given a program its structure can be determined. The structure chart of a program is a graphic representation of its structure. In a structure chart a module is represented by a box with the module name written in the box. An arrow from module A to module B represents that module A invokes module B. B is called the subordinate of A, and A is called the super ordinate of B. The arrow is labeled by the parameters received by B as input and the parameters returned by B as output, with the direction of flow of the input and output parameters represented by small arrows. The parameters can be shown to be data (unfilled circle at the tail of the label) or control (filled circle at the tail). As an example, consider the structure of the following,

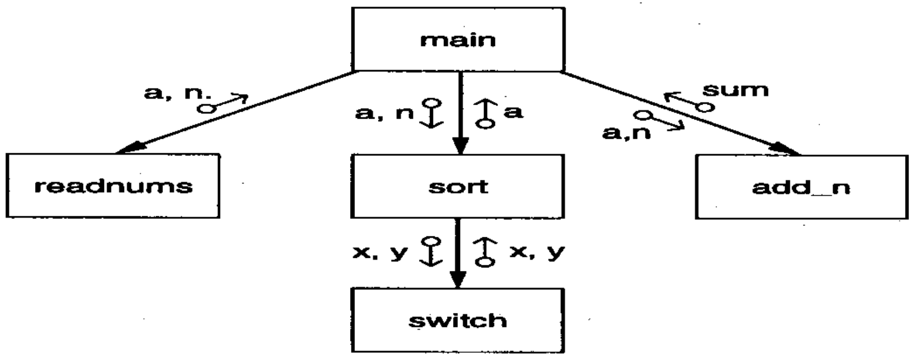

Figure 6.3: The structure chart of the sort program.

In general, procedural information is not represented in a structure chart, and the focus is on representing the hierarchy of modules. However, there are situations where the designer may wish to communicate certain procedural information explicitly, like major loops and decisions. Such information can also be represented in a structure chart. For example, let us consider a situation where module A has subordinates B, C, and D, and A repeatedly calls the modules C and D. This can be represented by a looping arrow around the arrows joining the subordinates C and D to A, as shown in Figure 6.4. All the subordinate modules activated within a common loop are enclosed in the same looping arrow.

Major decisions can be represented similarly. For example, if the invocation of modules C and D in module A depends on the outcome of some decision, that is represented by a small diamond in the box for A, with the arrows joining C and D coming out of this diamond, as shown in Figure 6.4.

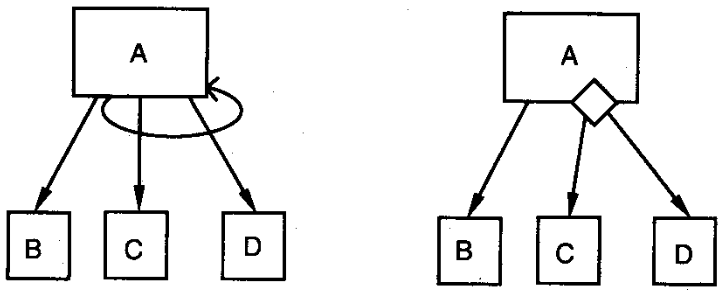

Figure 6.4: Iteration and decision representation.

Structured Design Methodology

No design methodology reduces design to a series of steps that can be mechanically executed. All design methodologies are, at best, a set of guidelines that, if applied, will most likely produce a design that is modular and simple.

The basic principle behind the structured design methodology, as with most other methodologies, is problem partitioning. Structured design methodology partitions the system at the very top level into various subsystems, one for managing each major input, one for managing each major output, and one for each major transformation. The modules performing the transformation deal with data at an abstract level, and hence can focus on the conceptual problem of how to perform the transformation without bothering with how to obtain clean inputs or how to present the output.

The actual transfor- mation in the system is frequently not very complex—it is dealing with data and getting it in proper form for performing the transformation or producing the output in the desired form that requires considerable processing.

This partitioning is at the heart of the structured design methodology. There are four major steps in the methodology:

1. Restate the problem as a data flow diagram
2. Identify the input and output data elements
3. First-level factoring
4. Factoring of input, output, and transform branches

Restate the Problem as a Data Flow Diagram To use this methodology, the first step is to construct the data flow diagram for the problem. We studied data flow diagrams in Chapter 3. However, there is a fundamental difference between the DFDs drawn during requirements analysis and those drawn during structured design. In the requirements analysis, a DFD is drawn to model the problem domain. The analyst has little control over the problem, and hence his task is to extract from the problem all the information and then represent it as a DFD.

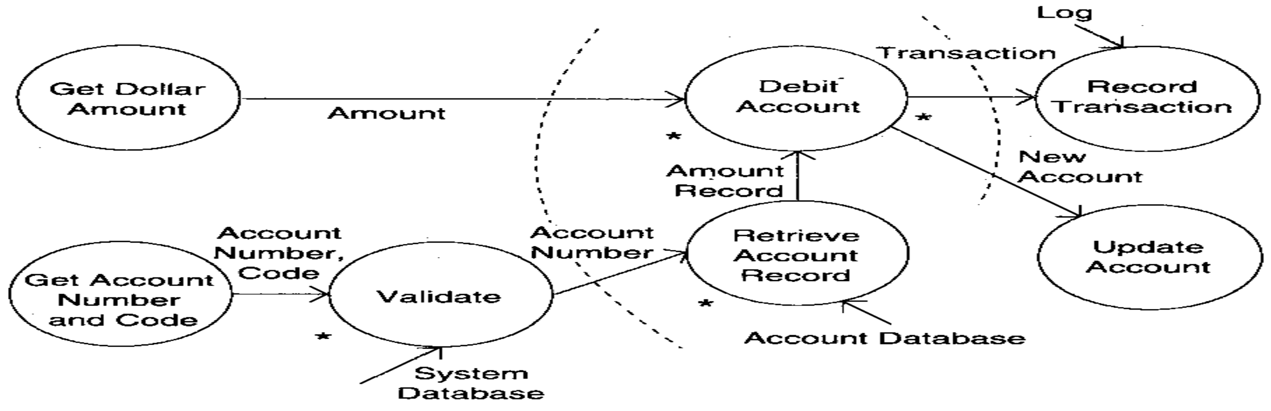

Figure 6.5: Data flow diagram of an ATM.

First-Level Factoring Having identified the central transforms and the most abstract input and output data items, we are ready to identify some modules for the system. We first specify a main module, whose purpose is to invoke the subordinates. The main module is therefore a coordinate module. For each of the most abstract input data items, an immediate subordinate module to the main module is specified.

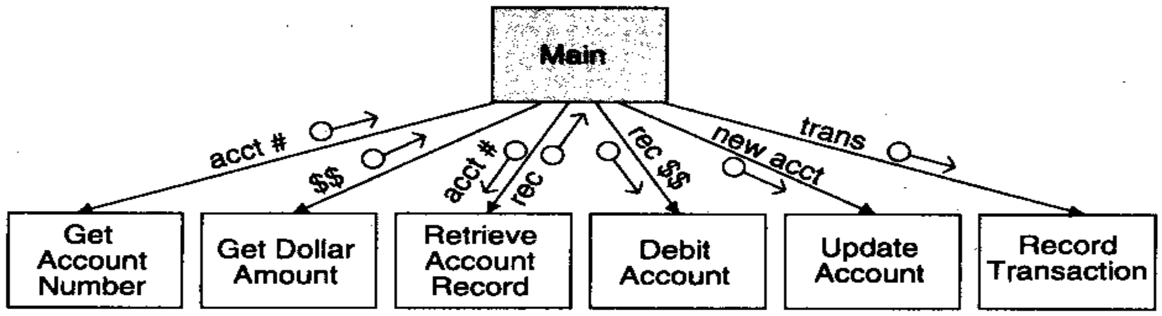

Figure 6.6: First-level factoring for ATM
Factoring the Input, Output, and Transform Branches The first-level factor- ing results in a very high level structure, where each subordinate module has a lot of processing to do. To simplify these modules, they must be factored into subordinate modules that will distribute the work of a module. Each of the input, output, and transformation modules must be considered for factoring.

Object-Oriented Design

Object-oriented (OO) approaches for software development have become ex- tremely popular in recent years. Much of the new development is now being done using OO techniques and languages. There are many advantages that OO systems offer. An OO model closely represents the problem domain, which makes it easier to produce and understand designs. As requirements change, the objects in a system are less immune to these changes, thereby permitting changes more easily. Inheritance and close association of objects in design to problem domain entities encourage more re-use, i.e., new applications can use existing modules more effectively, thereby reducing development cost and cycle time. Object-oriented approaches are believed to be more natural and provide richer structures for thinking and abstraction.

OO Concepts

Here we very briefly discuss the main concepts behind object-orientation. Readers familiar with an OO language will be familiar with these concepts.

Classes and Objects Classes and objects are the basic building blocks of an OO design, just like functions (and procedures) are for a function-oriented design. Objects are entities that encapsulate some state and provide services to be used by a client, which could be another object, program, or a user.

A major advantage of encapsulation is that access to the encapsulated data is limited to the operations defined on the data. Hence, it becomes much easier to ensure that the integrity of data is preserved, something very hard to do if any program from outside can directly manipulate the data structures of an object. Encapsulation and separation of the interface and its implementation, also allows the implementation to be changed without affecting the clients as long as the interface is preserved.

Objects represent the basic runtime entities in an OO system; they occupy space in memory that keeps its state and is operated on by the defined opera- tions on the object. A class, on the other hand, defines a possible set of objects. We have seen that objects have some attributes, whose values constitute much of the state of an object. What attributes an object has are defined by the class of the object. Similarly, the operations allowed on an object or the services it provides, are defined by the class of the object

The relationship between a class and objects of that class is similar to the relationship between a type and elements of that type. A class represents a set of objects that share a common structure and a common behavior, whereas an object is an instance of a class.

Inheritance and Polymorphism Inheritance is a relation between classes that allows for definition and implementation of one class based on the definition of existing classes [62]. When a class B inherits from another class A, B is referred to as the subclass or the derived class and A is referred to as the superclass or the base class. In general, a subclass B will have two parts: a derived part and an incremental part

Unified Modeling Language (UML)

UML is a graphical notation for expressing object-oriented designs [35]. It is called a modeling language and not a design notation as it allows representing various aspects of the system, not just the design that has to be implemented. For an OO design, a specification of the classes that exist in the system might suffice. However, while modeling, during the design process, the designer also tries to understand how the different classes are related and how they interact to provide the desired functionality. This aspect of modeling helps build designs that are more likely to satisfy the requirements of the system.

Class Diagram The class diagram of UML is the central piece in a design or model. As the name suggests, these diagrams describe the classes that are there in the design. As the final code of an OO implementation is mostly classes, these diagrams have a very close relationship with the final code. There are many tools that translate the class diagrams to code skeletons, thereby avoiding errors that might get introduced if the class diagrams are manually translated to class definitions by programmers. A class diagram defines

1. Classes that exist in the system—besides the class name, the diagrams are capable of describing the key fields as well as the important methods of the classes.

2. Associations between classes—what types of associations exist between dif- ferent classes.

3. Subtype, supertype relationship—classes may also form subtypes giving type hierarchies using polymorphism. The class diagrams can represent these hierarchies also.

A class itself is represented as a rectangular box which is divided into three areas. The top part gives the class name. By convention the class name is a word with the first letter in uppercase.

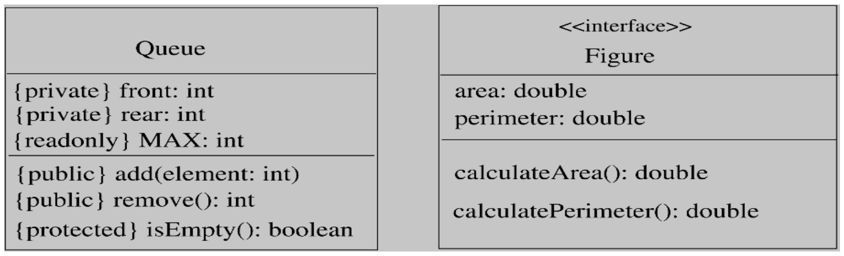

Figure 6.12: Class, stereotypes, and tagged values.

The generalization-specialization relationship is specified by having arrows coming from the subclass to the superclass, with the empty triangle-shaped arrowhead touching the superclass. Often, when there are multiple subclasses of a class, this may be specified by having one arrowhead on the superclass, and then drawing lines from this to the different subclasses.

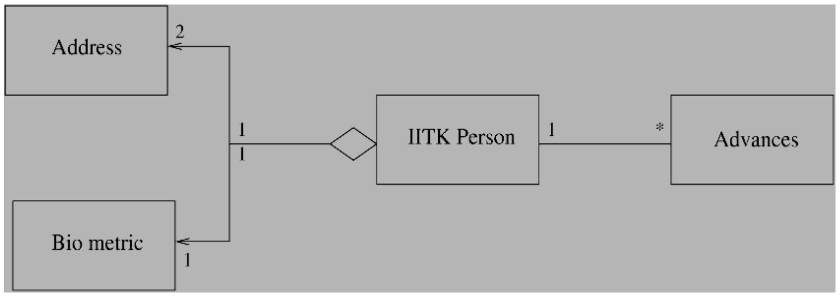

Figure 6.14: Aggregation and association among class

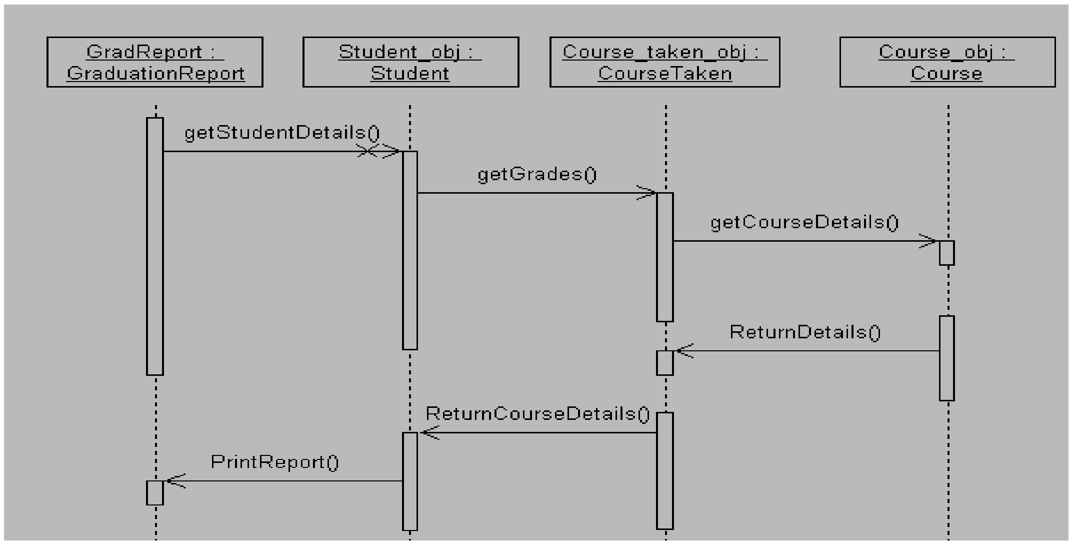

Figure 6.15: Sequence diagram for printing a graduation report.

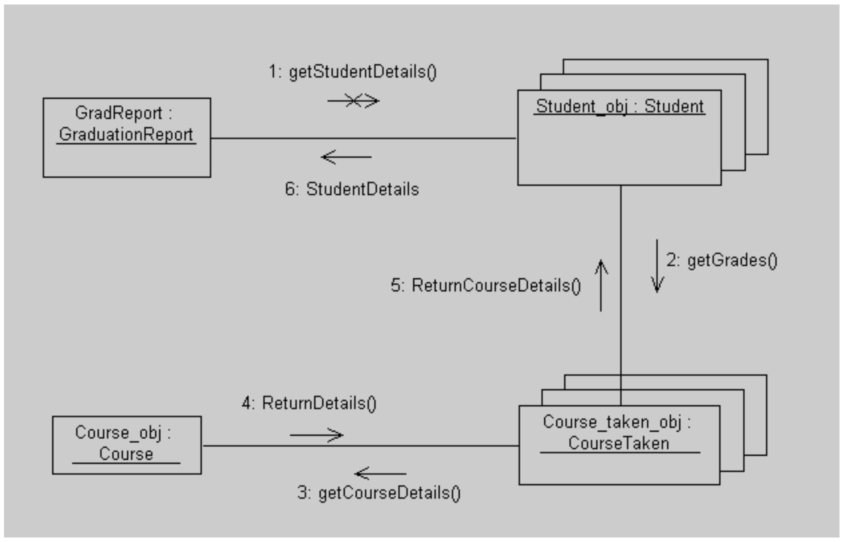

Figure 6.16: Collaboration diagram for printing a graduation report

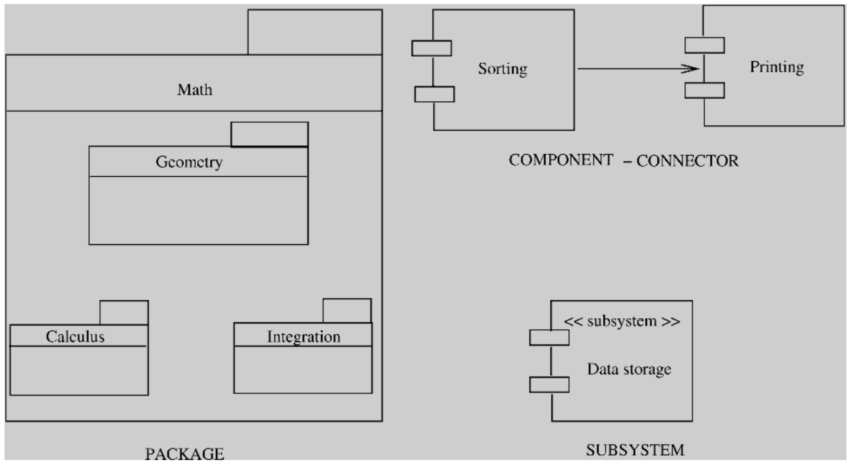

Figure 6.17: Subsystems, Components, and package

Detailed Design

In the previous two sections we discussed two different approaches for system design—one based on functional abstraction and one based on objects. In sys- tem design we concentrate on the modules in a system and how they interact with each other. Once the modules are identified and specified during the high- level design, the internal logic that will implement the given specifications can be designed, and is the focus of this section.

Logic/Algorithm Design

The basic goal in detailed design is to specify the logic for the different modules that have been specified during system design. Specifying the logic will require developing an algorithm that will implement the given specifications. Here we consider some principles for designing algorithms or logic that will implement the given specifications.

The term algorithm is quite general and is applicable to a wide variety of areas. For software we can consider an algorithm to be an unambiguous proce- dure for solving a problem

State Modeling of Classes

For object-oriented design, the approach just discussed for obtaining the de- tailed design can be used for designing the logic of methods. But a class is not a functional abstraction and cannot be viewed as merely a collection of functions (methods).

The technique for getting a more detailed understanding of the class as a whole, without talking about the logic of different methods, has to be different from the refinement-based approach. An object of a class has some state and many operations on it. To better understand a class, the relationship between the state and various operations and the effect of interaction of various opera- tions have to be understood

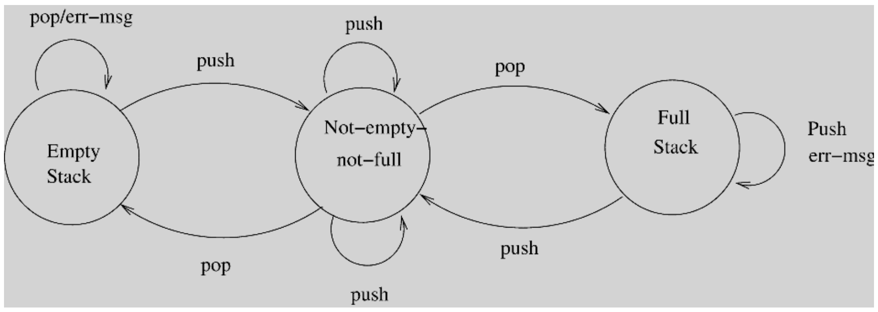

Figure 6.23: FSA model of a stack.

The finite state modeling of objects is an aid to understand the effect of various operations defined on the class on the state of the object. A good un- derstanding of this can aid in developing the logic for each of the operations. To develop the logic of operations, regular approaches for algorithm development can be used. The model can also be used to validate if the logic for an operation is correct. As we will see later, a state model can be used for generating test cases for validation.

Verification

The output of the design activity should be verified before proceeding with the activities of the next phase. If the design is expressed in some formal notation for which analysis tools are available, then through tools it can be checked for internal consistency (e.g., those modules used by another are defined, the interface of a module is consistent with the way others use it, data usage is consistent with declaration, etc.) If the design is not specified in a formal, executable language, it cannot be processed through tools, and other means for verification have to be used.

WHAT IS THE DIFFERENCE BETWEEN VERIFICATION AND VALIDATION PROCESS.

Verification	Validation
Verification is the process to find whether the software meets the specified requirements for particular phase.	The validation process is checked whether the software meets requirements and expectation of the customer.
It estimates an intermediate product.	It estimates the final product.
The objectives of verification is to check whether software is constructed according to requirement and design specification.	The objectives of the validation is to check whether the specifications are correct and satisfy the business need.
It describes whether the outputs are as per the inputs or not.	It explains whether they are accepted by the user or not.
Verification is done before the validation.	It is done after the verification.
Plans, requirement, specification, code are evaluated during the verifications.	Actual product or software is tested under validation.
It manually checks the files and document.	It is a computer software or developed program based checking of files and document.

Metrics

Here we discuss some of the metrics that can be extracted from a design and that could be useful for evaluating the design. We do not discuss the standard metrics of effort or defect that are collected (as per the project plan) for project monitoring.

Size is always a product metric of interest. For size of a design, the total number of modules is a commonly used metric. (By using an average size of a module, from this metric the final size in LOC can be estimated and compared with project estimates.)

Another metric of interest is complexity. A possible use of complexity met- rics at design time is to improve the design by reducing the complexity of the modules that have been found to be most complex. This will directly improve the testability and maintainability.
Complexity Metrics for Function-Oriented Design

Network Metrics Network metrics is a complexity metric that tries to capture how “good” the structure chart is. As coupling of a module increases if it is called by more modules, a good structure is considered one that has exactly one caller. That is, the call graph structure is simplest if it is a pure tree. The more the structure chart deviates from a tree, the more complex the system. Deviation of the tree is then defined as the graph impurity of the design [87]. Graph impurity can be defined as

$$\text{Graph impurity} = n - e - 1$$

The module design complexity, Dc , is defined as

$$Dc = size * (inf\ low * outf\ low)^2 .$$

The term (inf low ∗outf low) refers to the total number of combinations of input source and output destination. This term is squared, as the interconnection between the modules is considered a more important factor (compared to the
internal complexity) determining the complexity of a module
The module size is considered an insignificant factor, and complexity Dc for a module is defined as
$Dc = f\ an\ \ in * f\ an\ \ out + inf\ low * outf\ low$
where fan in represents the number of modules that call this module and fan out is the number of modules this module calls.

Testing

Testing Concepts

In this section we will first define some of the terms that are commonly used when discussing testing. Then we will discuss some basic issues relating to how testing is performed, and the importance of psychology of the tester.

Error, Fault, and Failure

While discussing testing we commonly use terms like error, fault, failure etc. Let us start by defining these concepts

The term **error** is used in two different ways. It refers to the discrepancy between a computed, observed, or measured value and the true, specified, or theoretically correct value. That is, error refers to the difference between the actual output of a software and the correct output.

Fault is a condition that causes a system to fail in performing its required function. A fault is the basic reason for software malfunction and is practically synonymous with the commonly used term **bug**, or the somewhat more general term **defec**t. The term error is also often used to refer to defects

Failure is the inability of a system or component to perform a required function according to its specifications.

Test Case, Test Suite, and Test Harness

So far we have used the terms test case or set of test cases informally. Let us define them more precisely. A test case (often called a test) can be considered as comprising a set of test inputs and execution conditions, which are designed to exercise the SUT in a particular manner

A group of related test cases that are generally executed together to test some specific behavior or aspect of the SUT is often referred to as a test suite.

Psychology of Testing

As mentioned, in testing, the software under test (SUT) is executed with a set of test cases. As discussed, devising a set of test cases that will guarantee that all errors will be detected is not feasible. Moreover, there are no formal or precise methods for selecting test cases.

Levels of Testing

Testing is usually relied upon to detect the faults remaining from earlier stages, in addition to the faults introduced during coding itself. Due to this, different levels of testing are used in the testing process; each level of testing aims to test different aspects of the system.

The basic levels are unit testing, integration testing, system testing, and acceptance testing. These different levels of testing attempt to detect different types of faults. The relation of the faults introduced in different phases, and the different levels of testing are shown in Figure 8.1.

The first level of testing is called **unit testing**, which we discussed in the previous chapter. Unit testing is essentially for verification of the code produced by individual programmers, and is typically done by the programmer of the module. Generally, a module is offered by a programmer for integration and use by others only after it has been unit tested satisfactorily.

The next level of testing is often called **integration testing**. In this, many unit tested modules are combined into subsystems, which are then tested. The goal here is to see if the modules can be integrated properly. Hence, the emphasis on testing interfaces between modules. This testing activity can be considered testing the design.

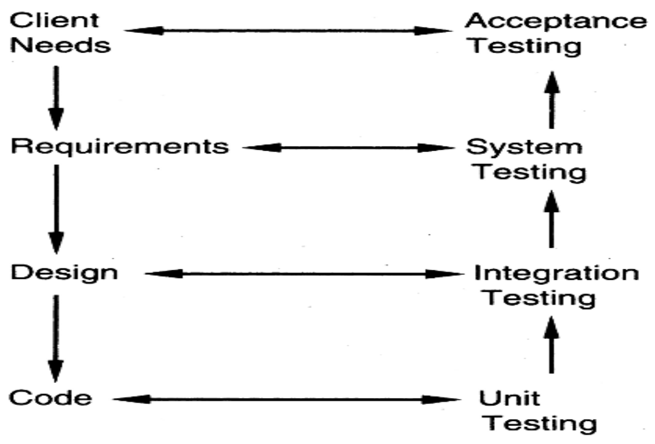
$$\begin{tikzcd}
\text{Client Needs} \arrow[r, leftrightarrow] \arrow[d] & \text{Acceptance Testing} \\
\text{Requirements} \arrow[r, leftrightarrow] \arrow[d] & \text{System Testing} \arrow[u] \\
\text{Design} \arrow[r, leftrightarrow] \arrow[d] & \text{Integration Testing} \arrow[u] \\
\text{Code} \arrow[r, leftrightarrow] & \text{Unit Testing} \arrow[u]
\end{tikzcd}$$

Figure 8.1: Levels of testing.

The next levels are **system testing and acceptance testing**. Here the entire software system is tested. The reference document for this process is the requirements document, and the goal is to see if the software meets its require- ments. This is often a large exercise, which for large projects may last many weeks or months. This is essentially a validation exercise, and in many situa- tions it is the only validation activity. Acceptance testing is often performed with realistic data of the client to demonstrate that the software is working satisfactorily. It may be done in the setting in which the software is to even- tually function. Acceptance testing essentially tests if the system satisfactorily solves the problems for which it was commissioned.

regression testing, some test cases that have been executed on the old system are maintained, along with the output produced by the old system. These test cases are executed again on the modified system and its output compared with the earlier output to make sure that the system is working as before on these test cases. This frequently is a major task when modifications are to be made to existing systems.

Complete regression testing of large systems can take a considerable amount of time, even if automation is used. If a small change is made to the system, often practical considerations require that the entire test suite not be executed, but regression testing be done with only a subset of test cases.

Testing Process

The basic goal of the software development process is to produce software that has no errors or very few errors. Testing is a quality control activity which focuses on identifying defects (which are then removed). We have seen that different levels of testing are needed to detect the defects injected during the various tasks in the project. And at a level multiple SUTs may be tested.

Test Plan

In general, in a project, testing commences with a test plan and terminates with successful execution of acceptance testing. A test plan is a general docu- ment for the entire project that defines the scope, approach to be taken, and the schedule of testing, as well as identifies the test items for testing and the personnel responsible for the different activities of testing. The test planning can be done well before the actual testing commences and can be done in par- allel with the coding and design activities. The inputs for forming the test plan are: (1) project plan, (2) requirements document, and (3) architecture. A test plan should contain the following:

– Test unit specification
– Features to be tested
– Approach for testing
– Test deliverables
– Schedule and task allocation

As seen earlier, different levels of testing have to be performed in a project. The levels are specified in the test plan by identifying the test units for the project. A test unit is a set of one or more modules that form a software under test (SUT).

Test Case Design

The test plan focuses on how the testing for the project will proceed, which units will be tested, and what approaches (and tools) are to be used during the various stages of testing. However, it does not deal with the details of testing a unit, nor does it specify which test cases are to be used.

Test case design has to be done separately for each unit. Based on the approach specified in the test plan, and the features to be tested, the test cases are designed and specified for testing the unit. Test case specification gives, for each unit to be tested, all test cases, inputs to be used in the test cases, conditions being tested by the test case, and outputs expected for those test cases.

Test Case Execution

With the specification of test cases, the next step in the testing process is to execute them. This step is also not straightforward. The test case specifications only specify the set of test cases for the unit to be tested. However, executing the test cases may require construction of driver modules or stubs. It may also require modules to set up the environment as stated in the test plan and test case specifications.

Black-Box Testing

As we have seen, good test case design is the key to suitable testing of the SUT. The goal while testing a SUT is to detect most (hopefully all) of the defects, through as small a set of test cases as possible. Due to this basic goal, it is important to select test cases carefully—best are those test cases that have a high probability of detecting a defect, if it exists, and also whose execution will give a confidence that no failures during testing implies that there are few (hopefully none) defects in the software.

There are two basic approaches to designing the test cases to be used in testing: black-box and white-box. In black-box testing the structure of the program is not considered. Test cases are decided solely on the basis of the requirements or specifications of the program or module, and the internals of the module or the program are not considered for selection of test cases. In this section, we will present some techniques for generating test cases for black-box testing. White-box testing is discussed in the next section

Equivalence Class Partitioning

Because we cannot do exhaustive testing, the next natural approach is to divide the input domain into a set of equivalence classes, so that if the program works correctly for a value, then it will work correctly for all the other values in that class. If we can indeed identify such classes, then testing the program with one value from each equivalence class is equivalent to doing an exhaustive test of the program.

Equivalence classes are usually formed by considering each condition speci- fied on an input as specifying a valid equivalence class and one or more invalid equivalence classes. For example, if an input condition specifies a range of val- ues (say, $0 < count < Max$), then form a valid equivalence class with that range and two invalid equivalence classes, one with values less than the lower bound of the range (i.e., $count < 0$) and the other with values higher than the higher bound ($count > Max$).

Once equivalence classes are selected for each of the inputs, then the issue is to select test cases suitably. There are different ways to select the test cases. One strategy is to select each test case covering as many valid equivalence classes as it can, and one separate test case for each invalid equivalence class.

Boundary Value Analysis

It has been observed that programs that work correctly for a set of values in an equivalence class fail on some special values. These values often lie on the boundary of the equivalence class. Test cases that have values on the boundaries of equivalence classes are therefore likely to be "high-yield" test cases, and selecting such test cases is the aim of boundary value analysis. v In the first strategy, we select the different boundary values for one variable, and keep the other variables at some nominal value. And we select one test case consisting of nominal values of all the variables. In this case, we will have $6n + 1$ test cases. For two variables X and Y, the 13 test cases will be as shown in Figure 8.4.

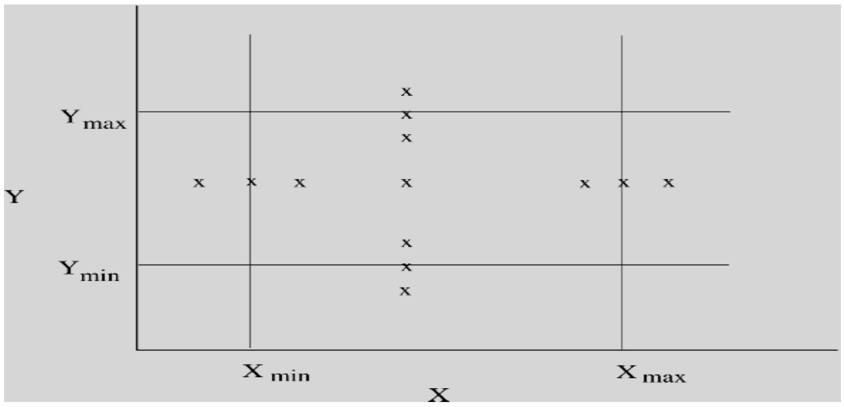

Figure 8.4: Test cases for boundary value analysis.

Pairwise Testing

There are generally many parameters that determine the behavior of a software system. These parameters could be direct input to the software or implicit settings like those for devices. These parameters can take different values, and for some of them the software may not work correctly.

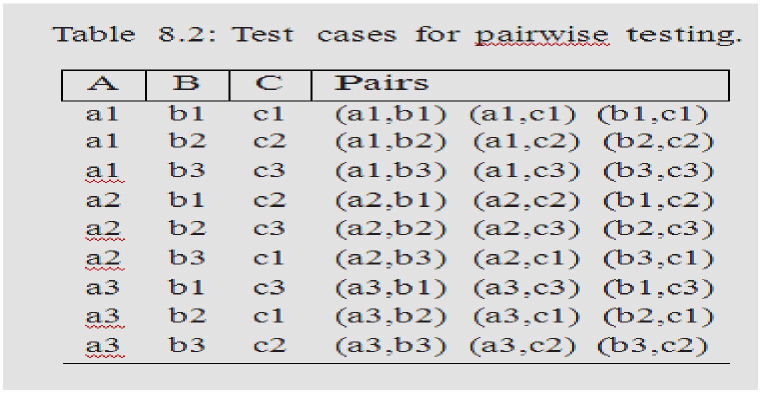
Table 8.2: Test cases for pairwise testing.

A	B	C	Pairs
a1	b1	c1	(a1,b1) (a1,c1) (b1,c1)
a1	b2	c2	(a1,b2) (a1,c2) (b2,c2)
a1	b3	c3	(a1,b3) (a1,c3) (b3,c3)
a2	b1	c2	(a2,b1) (a2,c2) (b1,c2)
a2	b2	c3	(a2,b2) (a2,c3) (b2,c3)
a2	b3	c1	(a2,b3) (a2,c1) (b3,c1)
a3	b1	c3	(a3,b1) (a3,c3) (b1,c3)
a3	b2	c1	(a3,b2) (a3,c1) (b2,c1)
a3	b3	c2	(a3,b3) (a3,c2) (b3,c2)

Pairwise testing is a practical way of testing large software systems that have many different parameters with distinct functioning expected for different values. An example would be a billing system (for telephone, hotel, airline, etc.) which has different rates for different parameter values. It is also a practical approach for testing general-purpose software products that are expected to run on different platforms and configurations, or a system that is expected to work with different types of systems.

Special Cases

It has been seen that programs often produce incorrect behavior when inputs form some special cases. The reason is that in programs, some combinations of inputs need special treatment, and providing proper handling for these special cases is easily overlooked. For example, in an arithmetic routine, if there is a division and the divisor is zero, some special action has to be taken, which could easily be forgotten by the programmer. These special cases form particularly good test cases, which can reveal errors that will usually not be detected by other test cases.

State-Based Testing

There are some systems that are essentially stateless in that for the same inputs they always give the same outputs or exhibit the same behavior. Many batch processing systems, computational systems, and servers fall in this category. In hardware, combinatorial circuits fall in this category. At a smaller level, most functions are supposed to behave in this manner. There are, however, many systems whose behavior is state-based in that for identical inputs they behave differently at different times and may produce different outputs. The reason for different behavior is that the state of the system may be different.A state model for a system has four components:

– States. Represent the impact of the past inputs to the system.

– Transitions. Represent how the state of the system changes from one state to another in response to some events.

– Events. Inputs to the system.

– Actions. The outputs for the events.

The state model shows what state transitions occur and what actions are performed in a system in response to events. When a state model is built from the requirements of a system, we can only include the states, transitions, and actions that are stated in the requirements or can be inferred from them. If more information is available from the design specifications, then a richer state model can be built.

White-Box Testing

In the previous section we discussed black-box testing, which is concerned with the function that the tested program is supposed to perform and does not deal with the internal structure of the program responsible for actually imple- menting that function. Thus, black-box testing is concerned with functionality rather than implementation of the program To test the structure of a program, structural testing aims to achieve test cases that will force the desired coverage of different structures. Various criteria have been proposed for this. Unlike the criteria for functional testing, which are frequently imprecise, the criteria for structural testing are generally quite precise as they are based on program structures, which are formal and precise. Here we will discuss one approach to structural testing: control flow-based testing, which is most commonly used in practice. **Control Flow-Based Criteria**

Most common structure-based criteria are based on the control flow of the program. In these criteria, the control flow graph of a program is considered and coverage of various aspects of the graph are specified as criteria. Hence, before we consider the criteria, let us precisely define a control flow graph for a program.

Let the control flow graph (or simply flow graph) of a program P be G. A node in this graph represents a block of statements that is always executed together, i.e., whenever the first statement is executed, all other statements are also executed. An edge (i, j) (from node i to node j) represents a possible transfer of control after executing the last statement of the block represented by node i to the first statement of the block represented by node j. A node corresponding to a block whose first statement is the start statement of P is called the start node of G, and a node corresponding to a block whose last statement is an exit statement is called an exit node [73]. A path is a finite sequence of nodes (n1 , n2 , ..., nk), k > 1, such that there is an edge (ni , ni+1) for all nodes ni in the sequence (except the last node nk). A complete path is a path whose first node is the start node and the last node is an exit node.

A more general coverage criterion is branch coverage, which requires that each edge in the control flow graph be traversed at least once during testing. In other words, branch coverage requires that each decision in the program be evaluated to true and false values at least once during testing. Testing based on branch coverage is often called branch testing.

The trouble with branch coverage comes if a decision has many conditions in it (consisting of a Boolean expression with Boolean operators and and or). In such situations, a decision can evaluate to true and false without actually exercising all the conditions. For example, consider the following function that checks the validity of a data item. The data item is valid if it lies between 0 and 100.

```
int check(x)

int x;

{

        if ((x >= ) && (x <= 200))

                check = True;
        else check = False;
}
```

As the path coverage criterion leads to a potentially infinite number of paths, some efforts have been made to suggest criteria between the branch coverage and path coverage. The basic aim of these approaches is to select a set of paths that ensure branch coverage criterion and try some other paths that may help reveal errors. One method to limit the number of paths is to consider two paths the same if they differ only in their subpaths that are caused due to the loops. Even with this restriction, the number of paths can be extremely large.

Test Case Generation and Tool Support

Once a coverage criterion is decided, two problems have to be solved to use the chosen criterion for testing. The first is to decide if a set of test cases satisfy the criterion, and the second is to generate a set of test cases for a given criterion. Deciding whether a set of test cases satisfy a criterion without the aid of any tools is a cumbersome task, though it is theoretically possible to do manually. For almost all the structural testing techniques, tools are used to determine whether the criterion has been satisfied. Generally, these tools will provide feedback regarding what needs to be tested to fully satisfy the criterion.

There are many tools available for statement and branch coverage, the crite- ria that are used most often. Both commercial and freeware tools are available for different source languages. These tools often also give higher-level coverage data like function coverage, method coverage, and class coverage. To get the coverage data, the execution of the program during testing has to be closely monitored. This requires that the program be instrumented so that required data can be collected. A common method of instrumenting is to insert SOME statements called probes in the program. The sole purpose of the probes is to generate data about program execution during testing that can be used to compute the coverage. With this, we can identify three phases in generating coverage data:

1. Instrument the program with probes
2. Execute the program with test cases
3. Analyze the results of the probe data

Probe insertion can be done automatically by a preprocessor. The execution of the program is done by the tester. After testing, the coverage data is displayed by the tool—sometimes graphical representations are also shown.

Metrics

We have seen that during testing the software under test is executed with a set of test cases. As the quality of delivered software depends substantially on the quality of testing, a few natural questions arise while testing:

– How good is the testing that has been done?

– What is the quality or reliability of software after testing is completed? During testing, the primary purpose of metrics is to try to answer these and
other related questions. We will discuss some metrics that may be used for this purpose.

Coverage Analysis

One of the most commonly used approaches for evaluating the thoroughness of testing is to use some coverage measures. We have discussed above some of the common coverage measures that are used in practice—statement coverage and branch coverage. To use these coverage measures for evaluating the quality of testing, proper coverage analysis tools will have to be employed which can inform not only the coverage achieved during testing but also which portions are not yet covered.

Reliability

After testing is done and the software is delivered, the development is con- sidered over. It will clearly be desirable to know, in quantifiable terms, the reliability of the software being delivered. As reliability of software depends considerably on the quality of testing, by assessing reliability we can also judge the quality of testing. Alternatively, reliability estimation can be used to decide whether enough testing has been done. In other words, besides characterizing an important quality property of the product being delivered, reliability esti- mation has a direct role in project management—it can be used by the project manager to decide whether enough testing has been done and when to stop testing.

Defect Removal Efficiency

Another analysis of interest is defect removal efficiency, though this can only be determined sometime after the software has been released. The purpose of this analysis is to evaluate the effectiveness of the testing process being employed, not the quality of testing for a project. This analysis is useful for improving the testing process in the future.

Usually, after the software has been released to the client, the client will find defects, which have to be fixed (generally by the original developer, as this is often part of the contract). This defect data is also generally logged