Software project lifecycle:

System feasibility analysis->Requirement Analysis & specification->**software architecture**-> Project planning -> Design -> coding -> Testing-> Deployment -> Maintanance

Software Architecture: of as system provides very high level view of a system in terms of parts of a system and how they are related to form a whole system.

- Depending on how the system is partitioned, we get **different architectural view** of the system.
  Architecture facilitates development of a high-quality system.
- There are three main **architectural views of a system**- module, component and connector, and allocation.

1. **In the module view**, system is viewed as structure of a programming module like packages, class and functions.
2. **Component and connector**: system is a collection of runtime entities called components, which interact with each other through the connectors.
3. **Allocation view**: describes how the different software units are allocated to hardware resources in the system.

Compoent and connector view:
Example: online survey of students.

Fig. Architecture of the survey system.

Client server and database are runtime components
Bidirectional arrow shows RPC [Remote procedure call]

Design Concepts:

- The design activity begins when **requirements document** and **architecture** for the software is developed.
- During design, we further refine the architecture. Design focuses on what **modules** system should have and **which modules** have to be developed.
- Module view may be the **module structure** of **each component in the architecture**. In that case, design exercise determines the module structure of the components.
- Design is plan for solution to the system.

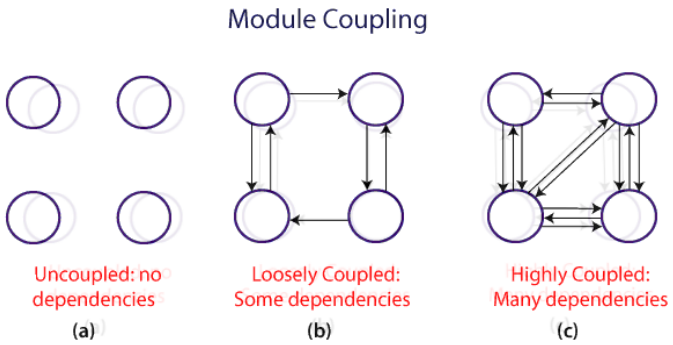
- Design process for software system has two levels:
  - Level 1: deciding which modules are needed for system, specification of these modules and how these modules should be interconnected. [**module design or high level design**]
  - Level 2: internal design of the modules, how the specifications of the module can be satisfied, is decided.[**detailed design or logic design**] (More detailed description of the logic and data structures so design is sufficiently complete for coding]
- Design methodology: is a systematic approach to creating design by applying of set of techniques and guidelines.

Design concepts: the design of the system is correct if a system built precisely according to the design satisfies the requirement of the system. To evaluate design, we have to specify evaluation criteria. Modularity is considered as main evaluation criterion.
System is **considered modular** if it consists of discrete modules so that each module can be implemeted separately, and a change to one module has minimal impact on other modules.
There are three modularization criterions:

1. Coupling
2. Cohesion
3. Open closed principle

1. Coupling

- Coupling between modules is the **strength of interconnections** between modules or measures of interdependence among the modules.
- Highly coupled modules are joined by **strong interconnection**.
- loosely coupled modules have **weak interconnections**. Independent modules have no interconnections.



Coupling increases with **complexity of the interface between modules**. To keep the coupling low we would like to **minimize the number of interface** per modules and **complexity** of the interface. Interface is used to **pass the information** to and from the modules.

For example: passing information to and from is done **through parameters.**
Coupling would **increase** if a module is used by other module via **indirect interface**.
More complex interface -> higher degree of coupling

Complexity of the **entry interface** of a procedure depends on the number of items being passed as parameters.
Type of information flow along the interfaces is third major factor affecting coupling.
There are two kind of information that can flow along an interface

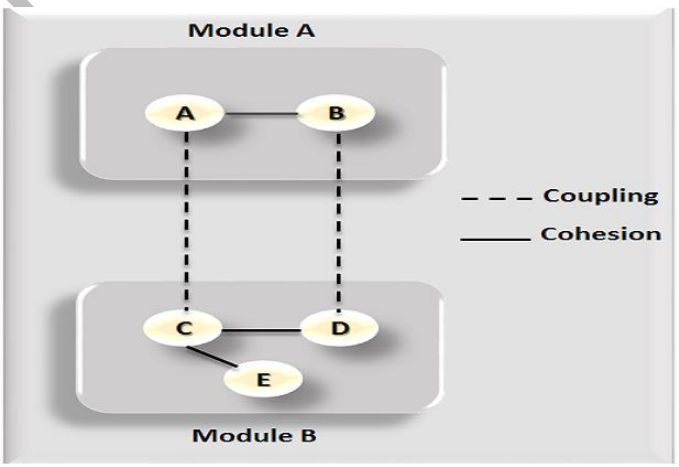
1. Data: Transfer of data information means module passess some input as a data to anothermodule and gets its return some data as output.
2. Control: passing and receiving control information means that the action of the module wil depend on this control information, which makes it more difficult to understand.

- Interface with **only data or control communication** result in lowest degree of coupling.
- **Coupling is high** if some data items and control information is passed between modules.[hybrid]

Effect of these three factors is summarized in table

	Interface complexity	**Types of connection**	**Type of communication**
Low	Simple obvious	To module By name	Data or control
High	Complicated Obscure	To internal elements	Hybrid

Table 1: Factors affecting coupling



In OO Systems, three different types of coupling exists between modules.

1. Interaction Coupling: occurs when methods of class invoking methods of other classes.
2. Component coupling: refers iteraction between two classes where class has variable of other class.
    a. A Class C can be component coupled with another class C1,if C has instance variable of type C1.
    b. Or C has method whose parameter is type of C1.
    c. Or C has method which has local variable of type C1.
3. Inheritance Coupling: due to inheritance relationship between the classes.

Cohesion: togehterness [inside module (various function)] (We need high cohesion)- functional strength of the module.

Cohesion of the module represents how tightly bound the internal elements module are to one another.

Greater the Cohesion of the module in the system lower the coupling between the modules.

There are seven levels of cohesion:

1. Coincidental Cohesion
2. Logical Cohesion
3. Temporal Cohesion
4. Procedural Cohesion
5. Communicational Cohesion
6. Sequential Cohesion
7. Functional Cohesion

1. **Coincidental Cohesion:**
    - The worst degree of cohesion is coincidental which is placed at the lowest level. **It occur when there is no meaningful relationship exist among the elements of a module**. so a module perform the tasks which are loosely related or almost unrelated to each other.
    - It can occur if an existing program is modularized by breaking it into pieces and making different pieces of module.
    - if a module is created to save duplicate code by combing some part of code which occurs at many different places that module is likely to have coincidental cohesion.
    - So the statements of the module do not any relationship with each other.
    - For example a module that checks a user security classification and also prints this week payroll is coincidentally cohesive.

2. Logical Cohesion:

- Logical cohesion is the next higher level of cohesion where **several logically related functions or data elements are placed in the same component.**
- For one module may read all kinds of inputs (from tape disk etc.) regardless of where the input is coming from or how it will be used input is the glue that holds this module together. Although more reasonable than confidential one, the elements of a logically cohesive module are not related functionally. In our example since the input can have different purposes for different modules, we perform many unrelated functions in one place.

3. Temporal Cohesion:

- Temporal cohesion is same as logical cohesion, except **that here the elements are related with time and executed together.** So the time concern is associated here.
- When a module performs tasks that must be executed with same duration of time, that module is called a logically cohesive module.
- elements of the modules in this case are time related. Temporal cohesion is higher than logical cohesion because the elements are executed simultaneously.

4. Procedural Cohesion:

- **When elements are grouped together in a module just to follow a sequential order, the component is procedurally cohesive**. So the elements of a module are evaluated with each other and must be executed in a specific order, and belong to a common procedural unit. For
- example a loop of decision statements in a module may be combined to form a separate module. procedurally cohesive modules often occur when modular structure is determined from some form of flow chart.

5. Communicational Cohesion:

- **Communicational cohesive has elements that are related by a reference to the same input or output data**.
- Here the elements concentrate on one area of a data structure. In a communication bound module the elements are together because they operate on the same input or output. Such modules may perform more than one function.
- This cohesion is sufficiently high and generally acceptable if alternative structures with higher cohesion cannot be easily identified. However, communicational cohesion often destroys the modularity and functional independence of the design.

6. Sequential Cohesion:

- If the output from one part of a module is input to the next part the module has sequential cohesion.
- So all the elements are related in such a way in a modules so that the output of one forms the input to another. But the sequential cohesion does not provide any

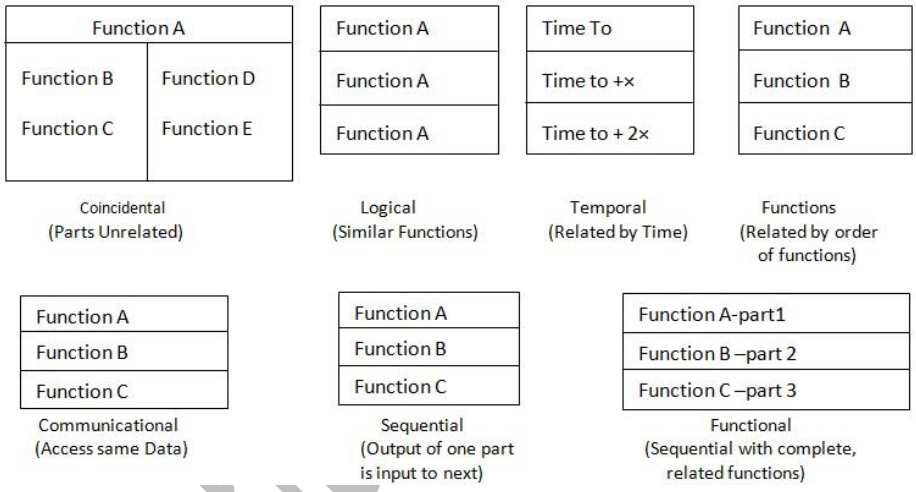
guidelines on how combine elements of a module. a sequentially bound module may contain several function or parts of different function.

7. Functional Cohesion:

- Our ideal is the functional cohesion which is the strongest cohesion. Here all the elements of the module are related to perform a single whole sole function. In this cohesion every processing elements is essential to the performance of a single function and all essential components are contained in one module.

For an example to find factorial of the number can be considered as a single function to be performed.

Fig 5.14. shown the example of various types of cohesion.



Cohesion in object oriented System:

Method Cohesion:

It focuses on **why the different code elements** of a method **are together within the method**. If each method implements the clearly defined function and all statements in the method contribute to implementing this function.(high cohesion)

Class cohesion: focuses on why different **attributes and methods** are **together** in this class. Cohesion is high if each class encapsulate a single concept.

Inheritance cohesion: focuses on why **different classes are together** in hierarchy. Two main reasons for inheritance are to **model generalization-specialization relationship** and **for code reuse**. Cohesion is considered high if hierarchy supports generalization-specialization relationship.

Open-closed principle: software entities should be open for extension [its behavior can be extended to accommodate new demands placed on this module due to changes in the requirements and system functionality] , but closed for modification[that existing source code of the module is not changed when making enhancement].

This principle is satisfied in object oriented design by using concepts like inheritance and polymorphisms.

Inheritance allows creating new classes that will extend the behavior of existing classes without changing the original class.

Example

Consider the application in which client object[type of Client] interact with printer object[type Printer] and invokes the methods to complete the printing. Class diagram is shown in fig.

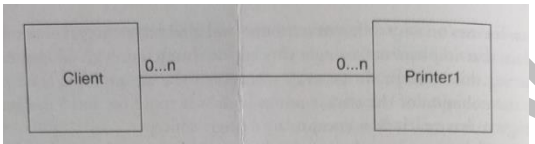


Figure 1 Example without using subtyping.

now suppose system has to be enhanced to allow another **printer** to be used by the client. *Under this design,to implement new change, a new class Printer2 will have to be created and code of the client class will be changed to allow using object of the Printer2 type as well. but this* ***design doesn't support the open-closed principle.***

So design for this system can be done in another manner that supports the open-closed principle.

Insteading of implementing printer1 class, we create an abstract class printer, that defines the interface of the printer and specifies all methods a printer object should support. Printer1 is implemented as specialization of Printer class. Here when printer2 is added as subclass of printer class.

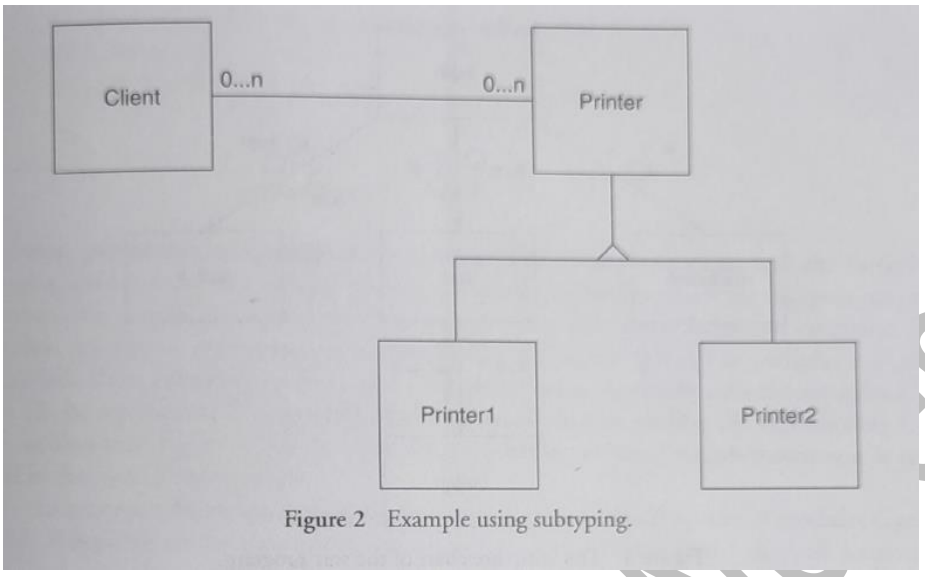


Figure 2 Example using subtyping.

Design:

1. Function Oriented Design
2. Object Oriented Design

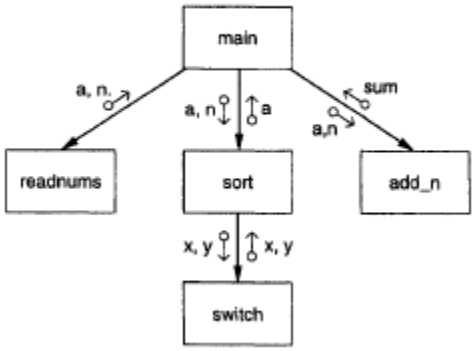
Function Oriented Design:

Structured Design methodology is used to design fuction oriented design. Structured design methodology uses structure chart as a graphical notation for function oriented design.

Structured Chart:

- 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 i**nput** 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 c**an be shown to be **data or control.**
- As an example consider the structure of the following program, whose structure is shown




```
main()
{
        int sum, n, N, a[MAX];
        readnums(a, &N);
        sort(a,N);
        scanf(&n);
        sum = add_n(a, n) ;
        printf(sum );
}
readnums ( a , N)
int a [ ] , *N ;
        {
        ;
        }
sort( a , N)
int a[] , N;
{         ;
        if (a[ i ] > a [ t ])
                switch(a i ] ,a[t]);
        ;
}

/* Add the first n numbers of a */
add_n(a, n)
int a [ ] , n ;
{
        :
}
```

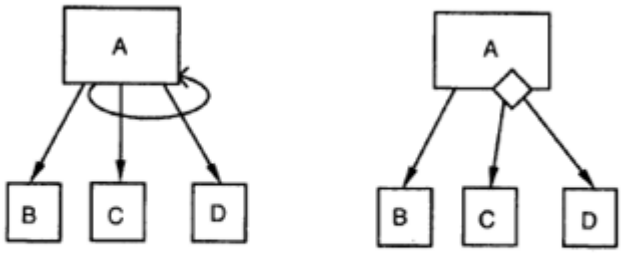


Figure 6.2: Iteration and decision representation.

- 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 6.2.
- All the subordinate modules activated within a common loop are enclosed in the same looping arrow.

Type of modules

Input module: module that obtain information from the subordinate and pass it to the superordinate is called input module.

Output modules: which takes input from superordinate and pass it to the subordinate is called output module.

Transform modules: which transforms data from one to other form

Coordinate modules: manage the flow of data to and from different subordinates.

Structured design methodology

Introduction: - 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**. Here we describe the structured design methodology for developing system designs.

There are four major steps in this strategy:

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

1. Restate the problem as a data flow diagram

Introduction:-

- During design activity, we are no longer modeling the **problem domain**, but rather are dealing with the **solution domain** and developing a model for the eventual system.
- That is, the **DFD during design represents how the data will flow in the system** when it is built.
- In this modeling, the major transforms or functions in the software are decided, and the DFD shows the major transforms that the software will have and how the data will flow through different transforms.
- So, drawing a DFD for design is a very **creative activity** in which the designer visualizes the eventual system and its processes and data flows.
- As the system does not yet exist, the **designer has complete freedom in creating a DFD that will solve the problem stated in the SRS.** The general rules of drawing a DFD remain the same; we show what transforms are needed in the software and are not concerned with the logic for implementing them.
- Consider the example of the simple automated teller machine that allows customers to withdraw money. A DFD for this ATM is shown in Figure

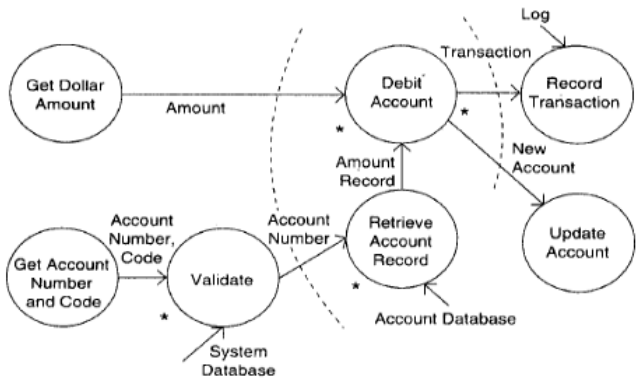


Fig. Data flow diagram for an ATM

- There are two major streams of input data in this diagram.
- The first is the **account number and the code**, and the second is **the amount** to be debited. The DFD is self-explanatory.

- Used * at different places in the DFD. For example, the transform "validate," which verifies if the account number and code are valid, needs not only the account number and code, but also information from the system database to do the validation.
- And the transform debit account has two outputs, one used for recording the transaction and the other to update the account.

2. Identify the Most Abstract Input and Output Data Elements

Introduction: -

- The goal of this second step is to **separate** the **transforms in the data flow diagram** that convert the **input or output to the desired format** from the ones that perform the actual transformations.
- For this separation, once the data flow diagram is ready, the next step is to **identify the highest abstract level of input and output.**
- The most abstract input data elements are **those data elements in the data flow diagram that are farthest removed** from the physical inputs but can still be considered inputs to the system.
- The most abstract input data elements often have little resemblance to the actual physical data. These are often the data elements obtained after operations like error checking, data validation, proper formatting, and conversion are complete.
- Most abstract input (MAI) data elements **are recognized by starting from the physical inputs and traveling toward the outputs in the data flow diagram, until the data elements are reached that can no longer be considered incoming**.
- The aim is to go as far as possible from the physical inputs, without losing the incoming nature of the data element. This process is performed for each input stream. Identifying the most abstract data items represents a value judgment on the part of the designer, but often the choice is obvious.
- We identify the most abstract output data elements (MAO) by **starting from the outputs in the data flow diagram and traveling toward the inputs.** These are the data elements that are most removed from the actual outputs but can still be considered outgoing.
- The MAO data elements may also be considered the **logical output data items**, and the transforms in the data flow diagram after these data items are basically to convert the logical output into a form in which the system is required to produce the output.
- The two **most abstract inputs are the dollar amount and the validated account numbe**r.
- The validated account number is the most abstract input, rather than the account number read in, as it is still the input—but with a guarantee that the account number is valid.
- The two abstract outputs are obvious.
- The abstract inputs and outputs are marked in the data flow diagram.

3. First-Level Factoring

- 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.
- Each of these modules is an **input module,** whose purpose is to deliver to the main module the **most abstract data item** for which it is created.
- Similarly, for each most abstract output data item, a **subordinate module that** is an **output module** that accepts data from the main module is specified.
- Each of the arrows connecting these **input and output subordinate** modules are **labeled** with the respective abstract data item flowing in the proper direction.
- Let us examine the data flow diagram of the ATM. It has **two most abstract inputs**, **two most abstract outputs, and two central transforms**. Drawing a module for each of these, we get the structure chart shown in Figure 6.7.
- As we can see, the first-level factoring is straightforward, after the most abstract input and output data items are identified in the data flow diagram.
- The main module is the overall **control module**, which will form the **main program or procedure** in the implementation of the design.
- It is a **coordinate module** that invokes the **input modules to get the most abstract data items**, passes these to the appropriate transform modules, and delivers the results of the transform modules to other transform modules until the most abstract data items are obtained. These are then passed to the output modules.

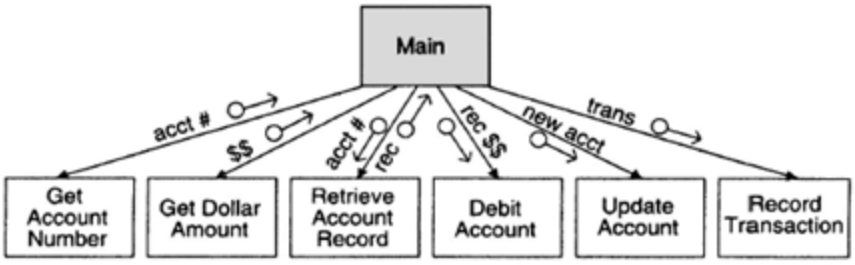


Figure 6.7: First-level factoring for ATM.

4. Factoring the Input, Output, and Transform Branches

- Introduction:- The first-level factoring 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.

Example:

Problem of determining the number of different words in an input file.

1. Restate the problem as a data flow diagram

- This problem has only one **input data stream, the input fil**e, while the desired output is the **count of different words in the file**.
- To transform the input to the desired output, the first thing we do is form a list of all the words in the file.
- It is best to then **sort the list,** as this will make identifying different words easier.
- This sorted fist is then used to **count the number of different words**, and the output of this transform is the **desired count**, which is **then printed**.
- This sequence of data transformation is what we have in the data flow diagram.

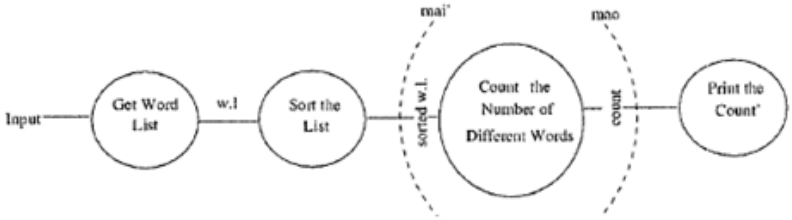


Figure 6.5: DFD for the word-counting problem.

2. Identify the Most Abstract Input and Output Data Elements

- The choice of the most abstract input is obvious.
- We start following the input. First, **the input file is converted into a word list**, which is essentially the **input in a different form**.
- **The sorted word list is still basically the input**, as it is still the **same list, in a different order**.
- This appears to be the **most abstract input** because the next data (i.e., count) is not just another form of the input data.
- The choice of the **most abstract output is even more obvious**(Clear)
- count is the natural choice (a **data that is a form of input will not usually be a candidate for the most abstract output**).
- Thus we have **one central transform**, **count-number-of-different-words**, which has one input and one output data item.

3. First-level factoring

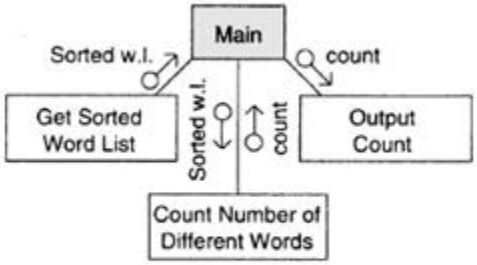


Figure 6.6: First-level factoring.

- The structure after the first-level factoring of the word-counting problem (its data flow diagram was given earlier) is shown in Figure 6.6.
- In this example, there is **one input module**, which **returns the sorted word list to the main module**.
- The output module takes **from the main module the value of the count**.
- There is only **one central transform** in this example, and a module is drawn for that.
- Data items traveling to and from this transformation module are the same as the data items going in and out of the central transform.

4. Factoring the Input, Output, and Transform Branches

- Factoring of input modules will usually **not yield any output subordinate modules.**
- The factoring of the input module **get-sorted-list in** the first-level structure is shown in Figure 6.8.
- The transform producing the **input returned by this module** (i.e., the sort transform) is treated as a central transform.
- Its input is the word list. Thus, in the first factoring we have an input module to get the list and a transform module to sort the list.
- The **input module** can be **factored further**, as the **module needs to perform two functions**, **getting a word and then adding it to the list**. Note that the looping arrow is used to show the iteration.

Figure 6.8: Factoring the input module.

- The **central transform** can be factored by creating a **subordinate transform** module for each of the transforms in this data flow diagram.
- This process **can be repeated** for the new transform modules that are created, until we reach atomic modules.
- The factoring of the **central transform count-the-number-of-different-words** is shown in Figure 6.9.This was a relatively simple transform, and we did not need to draw the data flow diagram.
- To determine the **number of words**, we have to **get a word repeatedly**,

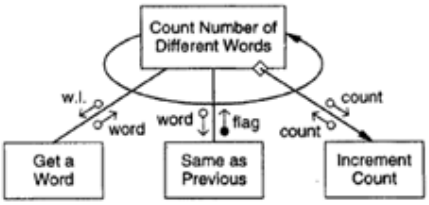


Figure 6.9: Factoring the central transform.

- Determine if it is the same as the previous word (for a sorted list, this checking is sufficient to determine if the word is different from other words), and then count the word if it is different. For each of the three different functions, we have a subordinate module, and we get the structure shown in Figure 6.9.

Object Oriented Design:

An OO model closely represents the problem domain, which makes it easier to produce and understand designs.

UML:

- Unified Modeling Language (UML) is a **graphical notation** for expressing object oriented designs. 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 a 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 all the requirements of the system.
- Due to the **ability of UML** to create **different models**, it has become an aid for understanding the system, designing the system, as well as a notation for representing design.

Class Diagram

Introduction:-

- The class diagram of UML is the central piece in a design or model.
- 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
    - **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.
    - **Associations between classes**—what types of associations exist between different classes.
    - **Subtype, super type 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**. First letter in the class should be uppercase.
- The **middle part** lists the **key attributes** or **fields** of the class.
- These attributes are the state holders for the objects of the class.
- The **name of the attributes** starts with **a lowercase,** and if multiple words are joined, then each new word **starts with an uppercase**.
- The bottom part lists the **methods or operations** of the class.
- These represent the **behavior** that the class can provide.

- Naming conventions are same as for attributes but to show that it is a function, the names end with **"( )".** Sometimes, designers may like to specify the **responsibility** of a class.
- The responsibility is what the entire class is meant to do using its **attributes and methods**. If responsibility needs to be specified, it is typically done by having a **4th part at the bottom** of the class box and specifying the responsibility in it as plain text.
- Example of a class, **an interface**, and a **class with some tagged values** is shown in Figure 7.9.

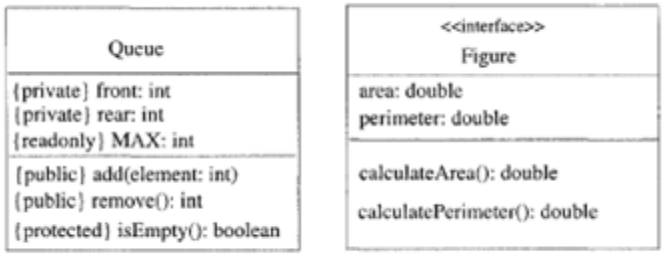


Figure 7.9: Class, stereotypes, and tagged values.

- The divided-box notation is to **describe the key features of a class** as a stand alone entity.
- To model a system or an application, we must represent relationship between classes.
- One common relationship is the **generalization-specialization** relationship between classes, which finally gets reflected as the inheritance hierarchy.
- All **properties of the superclass** are inherited by the **subclass**, so a subclass contains its own properties as well as those of the superclass.
- The generalization-specialization relationship is specified by having **arrows** coming from the **subclass to the superclass**, with the **empty triangle shaped arrow-head touching** to the superclass.
- When there are **multiple subclasses of a class**, this may be specified by having **one arrow head on the superclass,** and then drawing fines from this to the different subclasses.
- In this hierarchy, often **specialization is done on the basis of some discriminator**—a distinguishing property that is used to specialize superclass into different subclasses.
- Using the discriminator, objects of the superclass type are **partitioned i**nto sets of objects of different subclass types.
- The discriminator used for the **generalization-specialization relationship** can be specified by labeling the arrow.

Example: the IITKPerson class represents all people belonging to the IITK. These are broadly divided into two subclasses—Student and Employee, as both these types have many different properties and different behavior. Similarly, students have two different subclasses, UnderGraduate and PostGraduate, both requiring some different attributes and having different constraints. The Employee class has subtypes representing the

faculty, staff, and research staff.

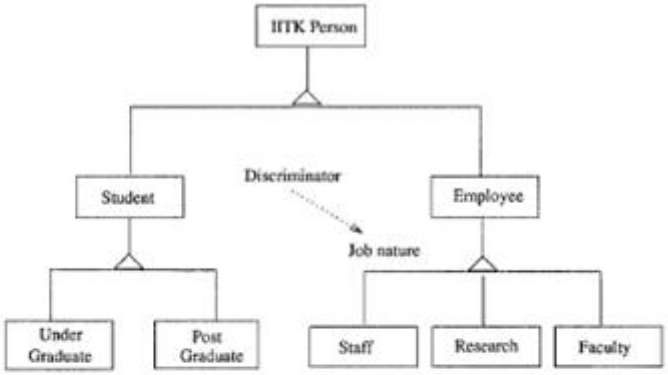


Figure 7.10: A class hierarchy.

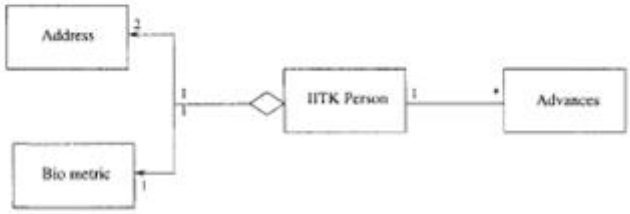


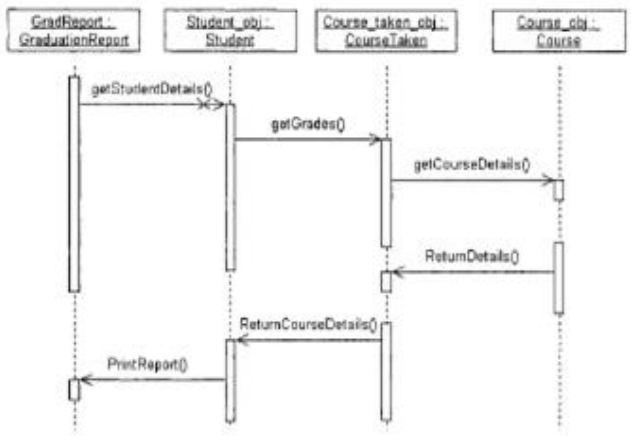
Figure 7.11: Aggregation and association among classes.

- An **association** between two classes means that an object of one class needs some services from objects of the other class to perform its own service.
- The association is shown by **a line between the two classes**.
- An association may have **a name** which can be specified by **labeling the association line**.
- And if the roles of the two ends of the association need to be **named**, that can also be done.
- In an association, an **end may also have multiplicity allowing relationships like 1 to 1, or 1 to many be modeled.**
  - **fixed multiplicity:** is represented by putting a number at that end;
  - **zero or many multiplicity** : is represented by a *.

- Another type of relationship is the **part-whole relationship** which represents the situation when an object is composed of many parts, each part itself is an object.
- This situation represents **containment or aggregation**—i.e. object of a class are contained inside the object of another class.
- For representing this aggregation relationship, the class which represents the **"whole"** is shown at the **top** and a **line** emanating from a **little diamond** connecting it to classes which represent the parts.
- Often in an implementation this relationship is implemented in the same manner as an association, hence, this relationship is also sometimes modeled as an **association**.

Sequence and Collaboration Diagrams

- Class diagrams represent the **static structure of the system**.
- Do not represent the **dynamic behavior** of the system.
- **Sequence diagrams or collaboration diagrams** together called interaction diagrams.
- **An interaction diagram** typically captures the **behavior of a use case** and models how the different objects in the system collaborate to implement the use case.
- **A sequence diagram** shows the **series of messages exchanged between some objects**, and their temporal ordering, when objects collaborate to provide some desired system functionality.
- The sequence diagram is generally drawn to model the **interaction between objects for a particular use case**.
- In a sequence diagram (and also in collaboration diagrams), it is objects that participate and not classes.
- When capturing dynamic behavior, the roles of classes are limited as during execution it is objects that exist.
- In a sequence diagram, all the **objects that participate** in the interaction are shown at the top as boxes with object names.
- For each object, **a vertical bar representing** its **lifeline** is drawn downwards.
- A message from **one object to another is represented as an arrow** from the lifeline of one to the lifeline of the other.
- **Each message** is labeled with the **message name**, which typically should be the name of a method in the class of the target object.



- An object can also make a self call, which is shown as an message **starting and ending in the same objects lifeline.**

- To clarify the sequence of messages and relative timing of each, **time** is represented as **increasing as one moves farther** away downwards from the object name in the object life.
- Time is represented by the y-axis, increasing downwards.
- Using the lifeline of **objects and arrows**, one can model objects lives and how messages flow from one object to another.
- frequently a message is sent from one object to another only under some condition.
- This condition can be represented in the sequence diagram by specifying it within brackets before the message name. If a message is sent to multiple receiver objects, then this multiplicity is shown by having **a "*"** before the message name.
- Each message has a return, which is when the operation finishes and returns the value (if any) to the invoking object.
- Though often this message can be implied, sometimes it may be desirable to show the return message explicitly. This is done by showing a dashed arrow.
- A sequence diagram for an example is shown in Figure. This example is for printing the graduation report for students.
- The object for **GradReport (**which has the responsibility for printing the report) sends a message to the Student objects for the relevant information, which request the CourseTaken objects for the courses the student has taken. These objects get information about the courses from the Course objects.
- A **collaboration diagram** also shows how **objects communicate**. Instead of using a timeline-based representation that is used by sequence diagrams, a collaboration diagram looks more like a state diagram.
- Each object is represented in the diagram, and the messages sent from one object to another are shown as **numbered arrows** from **one object to the other**.
- The chronological ordering of messages is captured by message numbering, in contrast to a sequence diagram where ordering of messages is shown pictorially.
- Two types of interaction diagrams are semantically equivalent and have the same representation power.
- The collaboration diagram for the above example is shown in Figure.
- As a system has many functions, each involving different objects in different ways, there will be a **dynamic model** for each of these functions or use cases.
- In other words, whereas one **class diagram** can capture the structure of the system's code, for the dynamic behavior many diagrams are needed.
- Many systems may be performing many functions and it may not be feasible or practical to draw the interaction diagram for each of these.
- Typically, during design, **interaction diagram** of some key **use cases or functions** will be drawn to make sure that the classes that exist can indeed support the desired use cases, and to understand their dynamics. An interaction diagram represents interactions of objects for one of the many scenarios.

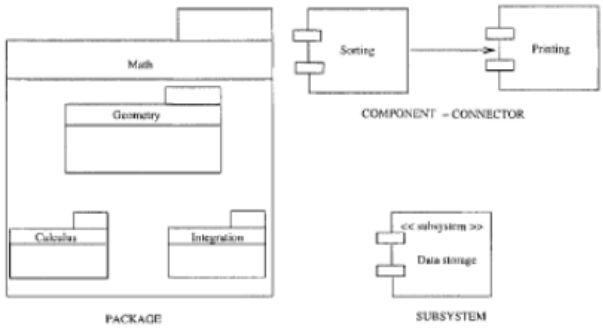


Figure 7.14: Subsystems, Components, and packages.

Design Methodololgy

Introduction: -

- A methodology basically uses the concepts to provide guidelines and notation for the design activity. Though methodologies are useful, they do not reduce the activity of design to a sequence of steps that can be followed mechanically.
- Due to this, the overall approach and the principles behind it are often more useful than the details of the **methodologie**s.
- In fact, most **experienced designers** tailor the methodology to suit their way of thinking and working. Even though it is one of the earlier methodologies, its basic concepts are still applicable.
- We assume that during **architecture design** the system has been broken into high-level subsystems or components.
- The problem we address is how to **produce an object-oriented design for a subsystem**, which can itself be viewed as a system.
- A complete OO design should be such that in the **implementation phase** only further details about methods or attributes need to be added.
- In OO design, the **OO analysis forms the starting step**. Using the model produced during analysis, a detailed model of the final system is built.
- In an object-oriented approach, the separation between **analysis and design** is not very clear and depends on the perception.
- The OMT methodology that we discuss for design considers **dynamic modeling and functional modeling parts of the analysis**.
- As these two models have little impact on **the object model produced in OOA** or on the SRS, we view these modeling as part of the **design activity**. Hence, performing the object modeling can be viewed as the first step of design.
- With this point of view, the design methodology for producing an OO design consists of the following sequence of steps:

  - identify classes and relationship between them.

- Develop the dynamic model and use it to define operations on classes
- Develop the functional model and use it to define operations on classes
- Identify internal classes and operations
- Optimize and package

- Any methodology can be followed, as long as the **output of the modeling activity is the class diagram** representing the problem structure.
- Hence, the first step of the design is generally performed during **the requirements phase** when the problem is being modeled for producing the SRS.
- The basic **goal is to produce a class diagram** of the problem domain.
- This requires identification of object types in the problem domain, the structures between classes (both inheritance and aggregation), attributes of the different classes, associations between the different classes, and the services each class needs to provide to support the system

Dynamic Modelling

- The dynamic model of a system aims to specify **how the state of various objects changes when events occur.**
- An event is something that happens at some time instance. For an object, an event is essentially a request for an operation..
- An event typically is an occurrence of something and has no time duration associated with it. **Each event has** an **initiator and a responder**.
- Events can be internal to the system, in which case the **event initiator and the event responder** are both within the system.

- A scenario is **a sequence of events** that occur in a particular execution of the system.
- With **use cases,** dynamic modeling involves preparing **interaction diagrams** for the important scenarios, identifying events on classes, ensuring that events can be supported, and perhaps build state models for the classes.
- It is best to start by modeling scenarios being triggered by external events. The scenarios should not necessarily cover all possibilities, but the major ones should be considered.

- First the **main success scenarios** should be modeled, then scenarios for **"exceptional"** cases should be modeled.
- An "**exception" scenario** could be if the ordered item was not available or if the customer cancels his order. From each scenario, events have to be identified.
- Events are interactions with the outside world and object-to object interactions.
- All the events that have the same effect on the flow of control in the system are grouped as a single event type.

- Each event type is then allocated to the **object classes** that initiate it and that service the event. With this done, a scenario can be represented as a sequence (or collaboration) diagram showing the events that will take place on the different objects if the execution corresponding to the scenario takes place.
- Once the **main scenarios** are modeled, various events on objects that are needed to support executions corresponding to the various scenarios are known.
- This information is then used to expand our view of the classes in the design. The main reason for performing dynamic modeling is that scenarios and sequence diagrams extend the initial design.
- for each event in the sequence diagrams, there will be an operation on the object on which the event is invoked.
- So, by using the scenarios and sequence diagrams we can further refine our view of the objects and add operations that are needed to support some scenarios but may not have been identified during initial modeling.
- For example, from the event trace diagram in Figure, we can see that "placeOrder" and "getBill" will be two operations required on the object of type Order if this interaction is to be supported. The effect of these different events on a class itself can be modeled using the state diagrams.

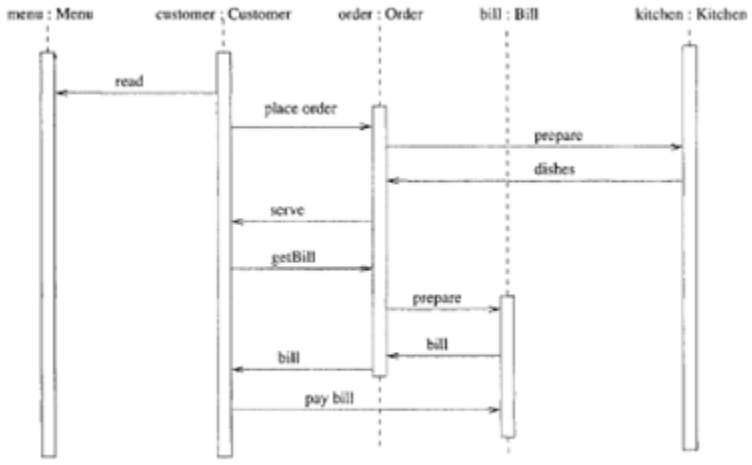


Figure 7.15: A sequence diagram for the restaurant.

Functional Modeling

A functional model of a system specifies how the output values are computed in the system from the input values, without considering the control aspects of the computation. This represents the functional view of the system—the mapping from inputs to outputs and the various steps involved in the mapping. Generally, when the transformation from the inputs to outputs is complex, consisting of many steps, the functional modeling is likely to be useful. In systems where the transformation of inputs to outputs is not complex, the functional model is likely to be straightforward.

The functional model of a system (either the problem domain or the solution domain) can be represented by a data flow diagram (DFD). The basic purpose of doing functional modeling, when the goal is to obtain an object-oriented design for the system, is to use the model to make sure that the object model can perform the transformations required from the system.

As processes represent operations and in an object-oriented system, most of the processing is done by operations on classes, all processes should show up as operations on classes. Some operations might appear as single operations on an object; others might appear as multiple operations on different classes, depending on the level of abstraction of the DFD. If the DFD is sufficiently detailed, most processes will occur as operations on classes. The DFD also specifies the abstract signature of the operations by identifying the inputs and outputs.

Internal Classes and operation:

Introduction:-The classes identified so far are the ones that come from the problem domain. The methods identified on the objects are the ones needed to satisfy all the interactions with the environment and the user and to support the desired functionality. However, the final design is a blueprint for implementation. Hence, implementation issues have to be considered. While considering implementation issues, algorithm and optimization issues arise. These issues are handled in this step.

First, each class is critically evaluated to see if it is needed in its present form in the final implementation. Some of the classes might be discarded if the designer feels they are not needed during implementation. Then the implementation of operations on the classes is considered. For this, rough algorithms for implementation might be considered. While doing this, a complex operation may get defined in terms of lower-level operations on simpler classes. In other words, effective implementation of operations may require heavy interaction with some data structures and the data structure to be considered an object in its own right. These classes that are identified while considering implementation concerns are largely support classes that may be needed to store intermediate results or to model some aspects of the object whose operation is to be implemented. The classes for these objects are called container classes. Once the implementation of each class and each operation on the class has been considered and it has been satisfied that they can be implemented, the system design is complete. The detailed design might also uncover some very low-level objects, but most such objects should be identified during system design.

Optimize and Package

Introduction:-In the design methodology used, the basic structure of the design was created during analysis. As analysis is concerned with capturing and representing various aspects of the problem, some in efficiencies may have crept in. In this final step, the issue of efficiency is considered, keeping in mind that the final structures should not deviate too much from

the logical structure produced by analysis, as the more the deviation, the harder it will be to understand a design.

Adding Redundant Associations. The association in the initial design may make it very inefficient to perform some operations. In some cases, these operations can be made more efficient by adding more associations. Consider the example where a Company has a relationship to a person. A person may have an attribute languages-spoken, which lists the languages the person can speak. If the company sometimes needs to determine all its employees who know a specific language, it has to access each employee object to perform this operation.

Saving Derived Attributes.This operation can be made more efficient by adding an index in the Company object for different languages, thereby adding a new relationship between the two types of objects. This association is largely for efficiency. For such situations, the designer must consider each operation and determine how many objects in an association are accessed and how many are actually selected. If the hit ratio is low, indexes can be considered. Saving Derived Attributes. A derived attribute is one whose value can be determined from the values of other attributes. As such an attribute is not independent, it may not have been specified in the initial design. However, if it is needed very often or if its computation is complex, its value can be computed and stored once and then accessed later. This may require new objects to be created for the derived attributes.

Use of Generic Types. A language like C allows "generic" classes to be declared where the base type or the type of some attribute is kept "generic" and the actual type is specified only when the object is actually defined .The approach of C does not support true generic types, and this type of definition is actually handled by the compiler. By using generic types, the code size can be reduced. For example, if a list is to be used in different contexts, a generic list can be defined and then instantiated for an integer, real, and char types.

Adjustment of Inheritance. Sometimes the same or similar operations are defined in various classes in a class hierarchy. By making the operation slightly more general (by extending interface or its functionality), it can be made a common operation that can be "pushed" up the hierarchy. The designer should consider such possibilities. Note that even if the same operation has to be used in only some of the derived classes, but in other derived classes the logic is different for the operation, inheritance can still be used effectively. The operation can be pushed to the base class and then redefined in those classes where its logic is different.

Another way to increase the use of inheritance, which promotes reuse, is to see if abstract classes can be defined for a set of existing classes and then the existing classes considered as a derived class of that. This will require identifying common behavior and properties among various classes and abstracting out a meaningful common superclass. Note that this is useful only if the abstract superclass is meaningful and the class hierarchy is "natural." A

superclass should not be created simply to pack the common features on some classes together in a class.

Word counting problem

Introduction:-

1. Identify classes and relationship between them.

The initial analysis clearly shows that there is a **File object**, which is an **aggregation of many Word objects**. There is a **Counter object**, which keeps **track of the number of different words**. As it does not belong "naturally" to either the class Word nor the class File, it will have to be "forced" into one of the classes. For this reason, we have kept Counter as a separate object. The basic problem statement finds only these **three objects**. Further analysis for services reveals that some **history mechanism is** needed to check if the word is unique.

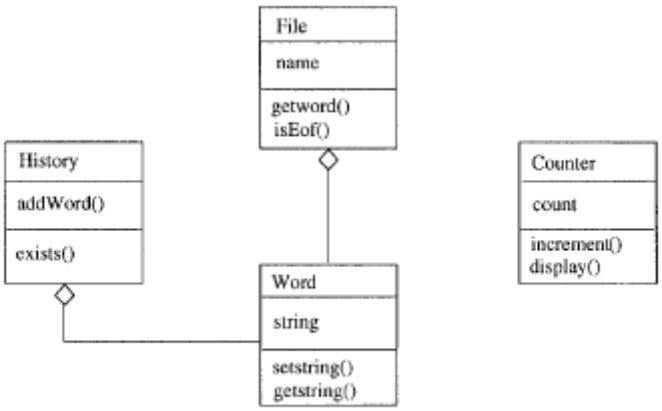


Figure 7.16: Class diagram for the word counting problem.

2. Develop the dynamic model and use it to define operations on classes

This is essentially a batch processing problem, where a **file is given as input** and some output is given by the system. Use case and scenario for this problem are straightforward. For example, the scenario for **the "normal"** case can be:

- System prompts for the file name; user enters the file name.
- System checks for existence of the file.
- System reads the words from the file.
- System prints the count

From this simple scenario, **no new operations are uncovered**, and our object diagram stays unchanged.

3. Develop the functional model and use it to define operations on classes

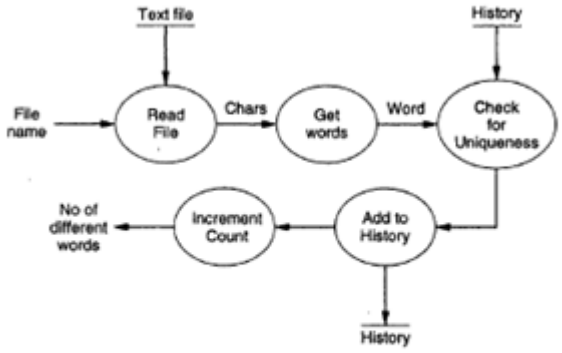


Figure 7.17: Functional Model for the word counting problem.

One possible functional model is shown in Figure 7.17.

1. The model **reinforces the need for some object where the history of what words have been seen is recorded.**
2. This object is used to **check the uniqueness of the words.**
3. It also shows that various operations like **increment (), isunique(), and addToHistoryO** are needed.
4. These operations should appear as operations in classes or should be supported by a combination of operations. In this example, most of these processes are reflected as **operations on classes** and are already incorporated in the design.

5. Optimize and package

1. **Implementation and optimization** concerns are used to enhance the object model.
2. The **history mechanism** will be implemented by a **binary search tree**.
3. Instead of the **class History**, we have a different **class B tree**.
4. For the **class Word**, various operations are needed to compare different words. Operations are also needed to set the string value for a word and retrieve it.
5. The final class **diagram is similar** in structure to the one shown in Figure 7.16, except for these changes.
6. The final step of the design activity is to **specify this design**. This is not a part of the design methodology, but it is an **essential step, as the design specification** is what forms the major part of the **design document**.
7. The design specification, as mentioned earlier, should specify all the classes that are in the design, all methods of the classes along with their interfaces.

```
class Word  {
    private :
        char *string; // string representing the word
    public:
        bool operator == ( Word ); // Checks for equality
        bool operator < ( Word );
        bool operator > ( Word );
        Word operator = ( Word ); // The assignment operator
        void setWord ( char * ); // Sets the string for the word
        char *getWord ( ); // gets the string for the word
};

class File {
    private:
        FILE inFile;
        char *fileName;
    public:
        Word getWord ( ); // get a word; Invokes operations of Word
        bool isEof ( ); // Checks for end of file
        void fileOpen ( char * );
};

class Counter {
    private:
        int counter;
    public:
        void increment ( );

        void display ( );
  };

  class Btree: GENERIC in <ELEMENT_TYPE>  {
      private:
          ELEMENT_TYPE element;
          Btree < ELEMENT_TYPE > *left;
          Btree < ELEMENT_TYPE > *right;
      public:
          void insert( ELEMENT_TYPE ); // to insert an element
          bool lookup( ELEMENT_TYPE ); // to check if an element exists
  };
```

Detailed design

- Useful for complex and important modules
- Done by programmer as part of the personal process of developing code.
- Logic/Algorithm Design:
- Basic Goal: specify the logic for the different modules.
- Write an algorithm

To design Algorithm :

Step 1: finalize Statement of the problem from System design.

Step 2: Design Algorithm

Data structure and program structure are decided.

- **Stepwise refinement technique** is used.
- Breaks the logic design problem into series of steps.
- Specification of module into abstract statements in algorithm.
- Successive refinement terminates with good algorithm having all steps.
- Decomposition process is used.

State Modeling of classes

- To better understand a class, understanding of **relationship between the state** and **various operations**.
- To understand the behavior of a class is to view it as **finite state automaton.**
- **State diagram:** relates **events and states** by showing how states changes when event is performed.

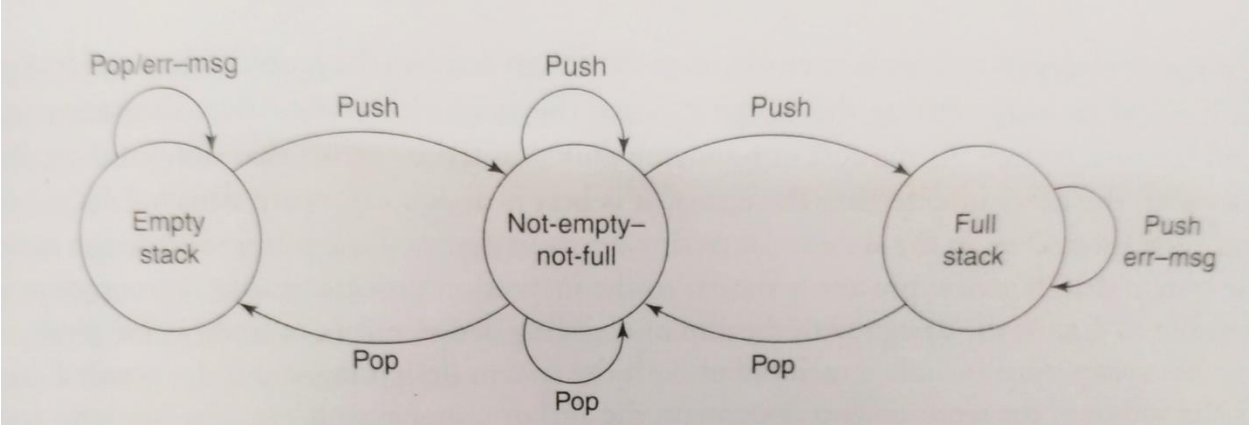


Figure 23 FSA model of a stack.

Verification

- Design satisfy the requirements of good quality
- Purpose is to find the design errors
- (inspection process)

Metrics:

Metrics are used to **evaluate a design** that can be extracted from design.

- **Size:** Number of Modules x Average LOC expected per module
  - *Or you can generate LOC estimates for each individual module*
- **Complexity**
  - *Network Metrics*
  - *Information Flow Metrics*

1. Network metrics:

Network metrics focus on the **structure chart of a system**

- They attempt to define how "**good" the structure or network** is in an effort to quantify the complexity of the call graph
- The simplest structure occurs if the **call graph is a tree**, with each node having two children
  - *As a result, the **graph impurity metric** is defined as **nodes - edges - 1***
  - *In the case of a tree, this **metric produces the result zero** since there is always one more node in a tree than edges*
  - *This metric is designed to make you examine nodes that have **high coupling** and see if there are ways to reduce this coupling*

2. Information flow metrics

- Information flow metrics attempt to **define the complexity of a system** in terms of the **total amount of information** flowing **through its modules**
- There are two information flow metrics and how they can be used to classify modules
  - ***Approach 1***
    - A module's complexity depends on its **intramodule complexity** and its **intermodule** complexity
    - intramodule complexity is approximated by the (estimated**) size of the module in lines of code**
    - intermodule complexity is determined by the **total amount of information** (abstract data elements) **flowing into a module** (inflow) and the total amount of information **flowing out of a module** (outflow)
    - The module design complexity D_c is defined as
    - **D_c = size * (inflow*outflow)^2**
    - The term **(inflow * outflow)^2** refers to the **total number of input and output combinations**, and this number is **squared** since the interconnections between modules are considered more important to determining the complexity of a module than its code size
  - ***Approach 2***
    - **Approach 1** depends largely on the **amount of information** flowing **in and out of the module**
    - ***Approach 2*** is a variant that also considers the **number of modules connected to a particular module**; in addition, the **code size** of a module is considered insignificant with respect to a module's complexity
    - The module design complexity D_c is defined as **D_c = (fan_in * fan_out) + (inflow*outflow)**
    - **fan_in** above refers to the **number of modules that call this module**, **fan_out** is the number of modules **called by this module**
  - ***Classification***
    - Neither of these metrics is any good, unless they can tell us when to consider a module "**too complex**"
    - To this end, an approach was developed to compare a module's complexity against the complexity of the other modules in its system
    - **avg_complexity** is defined as the average complexity of the modules in the current design
    - **std_deviation** is defined as the **standard deviation** in the design complexity of the modules in the **current design**
    - A module can be classified as ***error prone***, ***complex***, **or** ***normal*** using the following **conditions**
      - D_c is the complexity of a **particular module**
      - A module is **error prone** if **D_c > avg_complexity + std_deviation**
      - A module is **complex** if **avg_complexity < D_c < avg_complexity + std_deviation**
      - Otherwise a module is considered **normal**

Complexity metrics for Object oriented design:

OO metrics focus on identifying the **complexity of classes in an OO design**

1. Weighted Methods per Class
2. Depth of Inheritance Tree
3. Number of Children
4. Coupling Between Class
5. Response for a Class
6. Lack of Cohesion in Methods

1. Weighted Methods Per Class:

- The complexity of a class depends on the **number of methods** it has and the **complexity of those methods.**
- For a class with methods M1, M2, …, Mn, determine a complexity value for each method, c1, c2, …, cn using any metric that estimates complexity for functions (estimated size, interface complexity, data flow complexity, etc.)
- **WMC = Σci** ; this metric has been shown to have a reasonable correlation with **fault proneness**

2. Depth of Inheritance Tree

- DIT of class C is depth from the root class.
- DIT is significant in **predicting fault proneness**
- basic idea: the **deeper** in the tree, the more methods a particular class has, making it harder to change

3. Number of Children

- Immediate number of subclasses of C
- Gives a sense of reuse of C's features
- Most classes have a **NOC of 0**; one study showed, however, that classes with a **high NOC** had a tendency to be less **fault prone** than others

4. Coupling between classes

- **Number of classes** to which this **class is coupled**
- Two classes are coupled if **methods of one use methods** or **attributes of another**
- A study has shown that the CBC metric is significant in predicting the fault proneness of classes.

5. Response for a Class

- CBC metric **does not quantify the strength** of the connections its class has with other classes (it only counts them)
- The response for a class metric attempts to quantify this by capturing the total number of methods that can be invoked from an object of this class
- Thus even if a class has a CBC of "1" , its RFC value may be much higher
- A study has shown that the **higher a class's RFC value is**, the larger the **probability that class will contain defects.**