Name : Muskan Lamba
Subject Name : Software Design Lab
Experiment No. : 8
Experiment Name : Study of component level design.
Roll No. : 17IT1064
Class/Div/Batch : TE/A/A4
Date of performance :
Date of submission :
Grade :
Sign :

Aim: To study component level design concept
Hardware & Software Required: PC Desktop, Rational Rose
Theory:

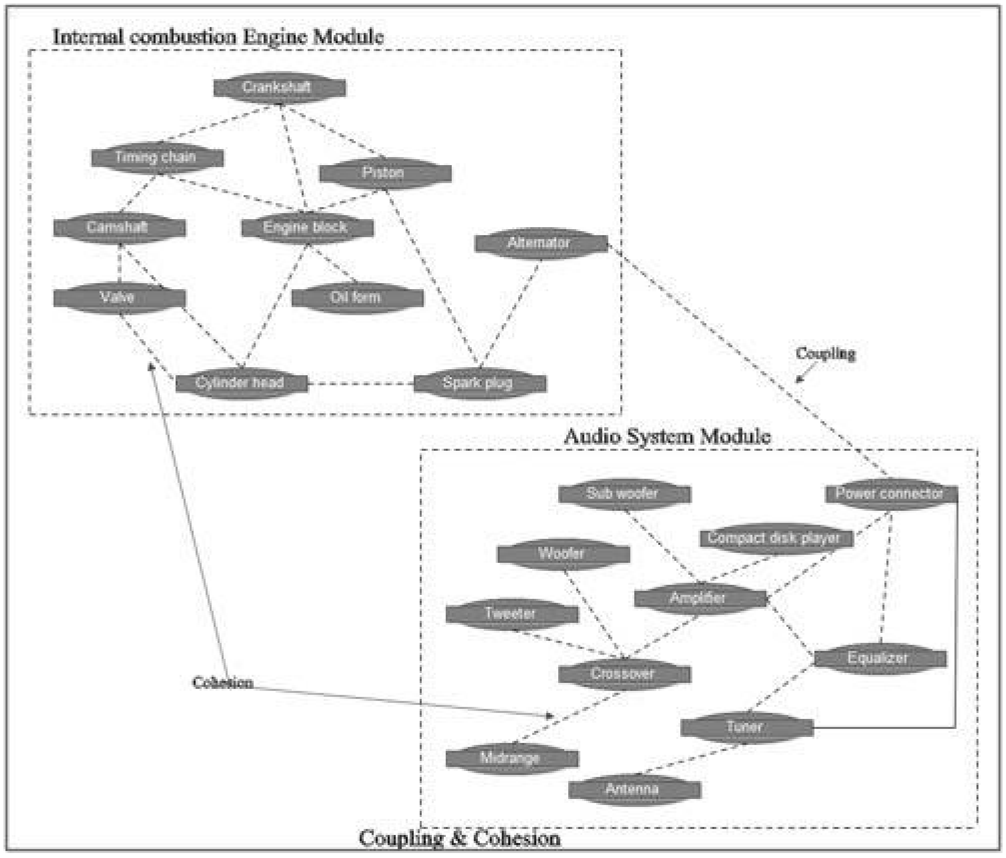
The component-level design can be represented by using different approaches. One approach is to use a programming language while other is to use some intermediate design notation such as graphical (DFD, flowchart, or structure chart), tabular (decision table), or text-based (program design language) whichever is easier to be translated into source code.

The component-level design provides a way to determine whether the defined algorithms, data structures, and interfaces will work properly. Note that a component (also known as module) can be defined as a modular building block for the software. However, the meaning of component differs according to how software engineers use it. The modular design of the software should exhibit the following sets of properties.

1. Provide simple interface: Simple interfaces decrease the number of interactions. Note that the number of interactions is taken into account while determining whether the software performs the desired function. Simple interfaces also provide support for reusability of components which reduces the cost to a greater extent. It not only decreases the time involved in design, coding, and testing but the overall software development cost is also liquidated gradually with several projects. A number of studies so far have proven that the reusability of software design is the most valuable way of reducing the cost involved in software development.
2. Ensure information hiding: The benefits of modularity cannot be achieved merely by decomposing a program into several modules; rather each module should be designed and developed in such a way that the information hiding is ensured. It implies that the implementation details of one module should not be visible to other modules of the program. The concept of information hiding helps in reducing the cost of subsequent design changes.

Functional Independence

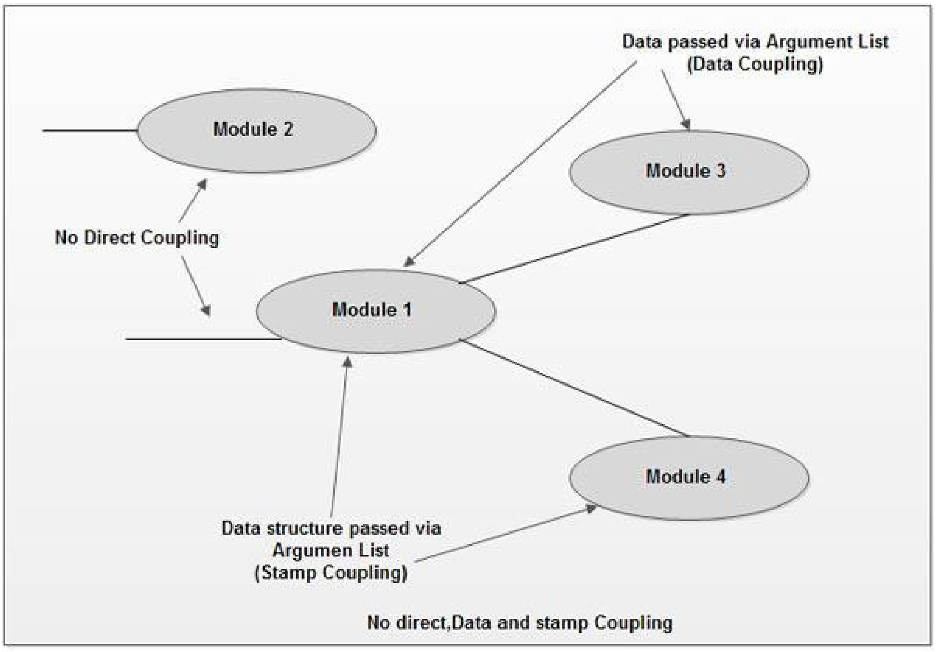
Functional independence is the refined form of the design concepts of modularity, abstraction, and information hiding. Functional independence is achieved by developing a module in such a way that it uniquely performs given sets of function without interacting with other parts of the system. The software that uses the property of functional independence is easier to develop because its functions can be categorized in a systematic manner. Moreover, independent modules require less maintenance and testing activity, as secondary effects caused by design modification are limited with less propagation of errors. In short, it can be said that functional independence is the key to a good software design and a good design results in high-quality software. There exist two qualitative criteria for measuring functional independence, namely, coupling and cohesion.



Coupling

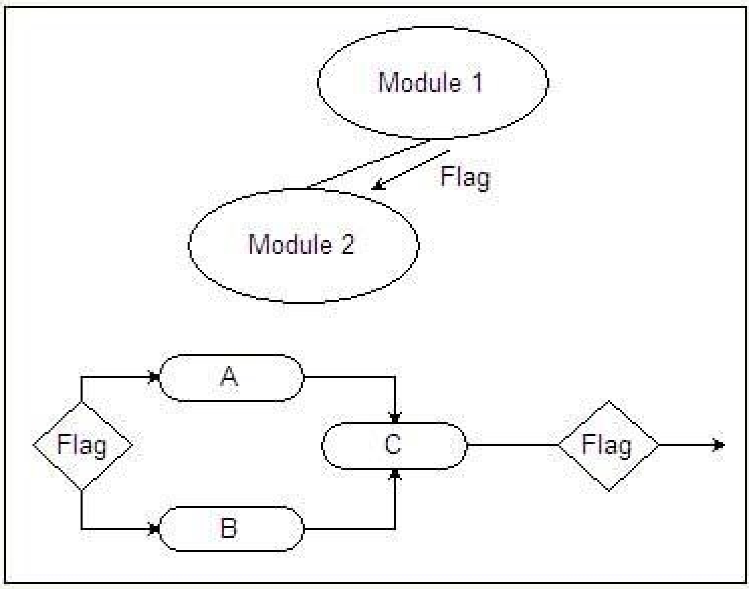
Coupling measures the degree of interdependence among the modules. Several factors like interface complexity, type of data that pass across the interface, type of communication, number of interfaces per module, etc. influence the strength of coupling between two modules. For better interface and well-structured system, the modules should be loosely coupled in order to minimize the 'ripple effect' in which modifications in one module results in errors in other modules. Module coupling is categorized into the following types.

No direct coupling: Two modules are said to be 'no direct coupled' if they are independent of each other.

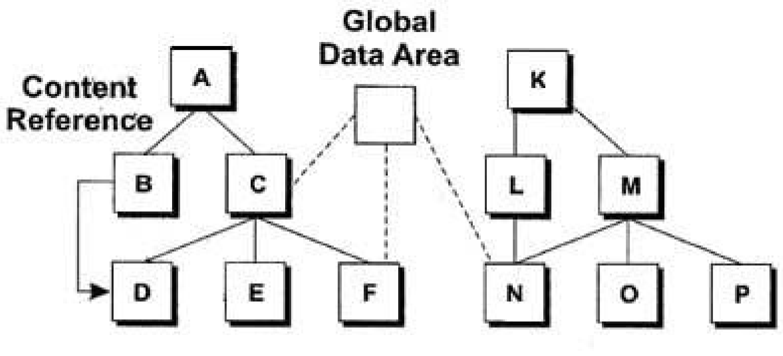


1. Data coupling: Two modules are said to be 'data coupled' if they use parameter list to pass data items for communication.
2. Stamp coupling: Two modules are said to be 'stamp coupled' if they communicate by passing a data structure that stores additional information than what is required to perform their functions.
3. Control coupling: Two modules are said to be 'control coupled' if they communicate (pass a piece of information intended to control the internal logic) using at least one 'control flag'. The control flag is a variable whose value is used by the dependent modules to make decisions.

When a component is to be accessed or shared across execution contexts or network links, techniques such as serialization or marshalling are often employed to deliver the component to its destination.



Content coupling: Two modules are said to be 'content coupled' if one module modifies data of some other module or one module is under the control of another module or one module branches into the middle of another module.



Content and Common Coupling

Common coupling: Two modules are said to be 'common coupled' if they both reference a common data block.

Cohesion

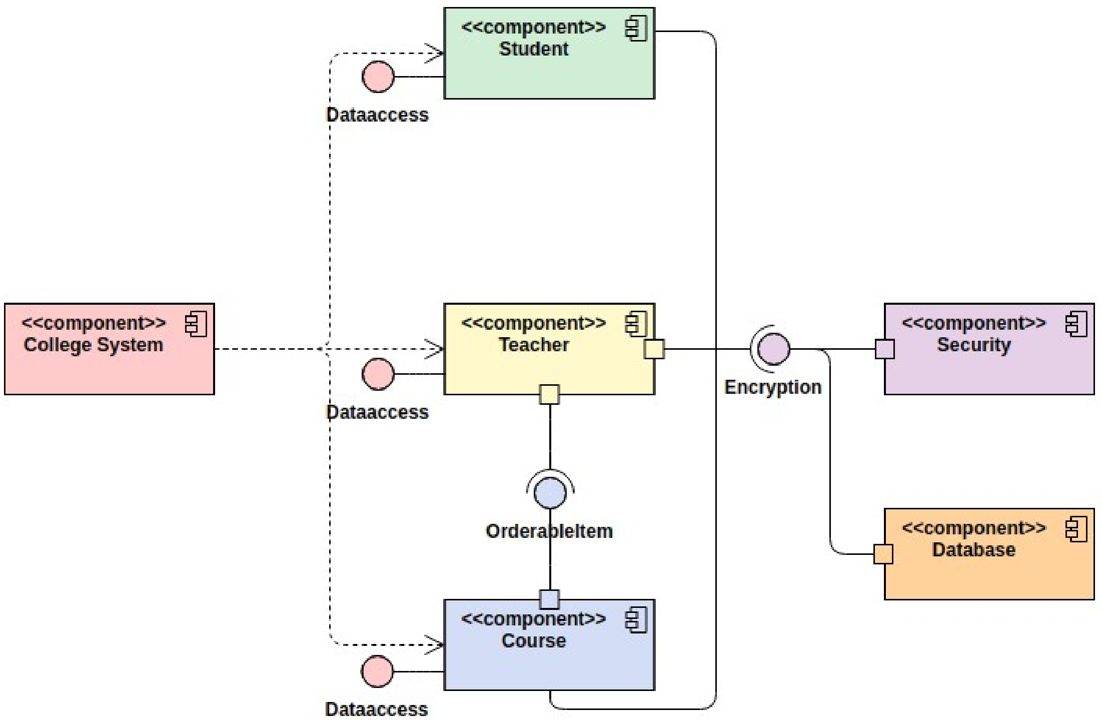
Cohesion measures the relative functional strength of a module. It represents the strength of bond between the internal elements of the modules. The tighter the elements are bound to each other, the higher will be the cohesion of a module. In practice, designers should avoid a low level of cohesion when designing a module. Generally, low coupling results in high cohesion and vice versa. Various types of cohesion are listed below.

1. Functional cohesion: In this, the elements within the modules are concerned with the execution of a single function.
2. Sequential cohesion: In this, the elements within the modules are involved in activities in such a way that the output from one activity becomes the input for the next activity.
3. Communicational cohesion: In this, the elements within the modules perform different functions, yet each function references the same input or output information.

4. Procedural cohesion: In this, the elements within the modules are involved in different and possibly unrelated activities.
5. Temporal cohesion: In this, the elements within the modules contain unrelated activities that can be carried out at the same time.
6. Logical cohesion: In this, the elements within the modules perform similar activities, which are executed from outside the module.

   Coincidental cohesion: In this, the elements within the modules perform activities with no meaningful relationship to one another

Results:



Conclusion and Discussion:

Component Level Design Development Process was successfully studied.