Object Oriented Software Engineering

Unit –II

Define Cohesion. List the types of Cohesions. Explain each one with suitable example.

Cohesion refers to the degree to which the elements inside a module belong together.In one sense, it is a measure of the strength of relationship between the methods and data of a class and some unifying purpose or concept served by that class. In another sense, it is a measure of the strength of relationship between the class's methods and data themselves.

Types

1. **Coincidental Cohesion**

A module has coincidental cohesion if it performs multiple, completely unrelated operations. An example of a module with coincidental cohesion is a module named print_the_ next_line, reverse_the_string_of_characters_comprising_the_second_argument,add_7_to_the_fifth_argume nt, convert_the_fourth_argument_to_floating_point.

2. **Logical Cohesion**

A module has logical cohesion when it performs a series of related operations, one of which is selected by the calling module. All the following are examples of modules with logical cohesion. Example 1 Module new_operation, which is invoked as follows:

function_code = 7;

new_operation (function_code, dummy_1, dummy_2, dummy_3); // dummy_1, dummy_2, and dummy_3 are dummy variables, // not used if function_code is equal to 7

Example 2 An object that performs all input and output.

Example 3 A module that edits insertions, deletions, and modifications of master fi le records.

Example 4 A module with logical cohesion in an early version of OS/VS2 that performed 13 different operations; its interface contained 21 pieces of data.

3. **Temporal Cohesion**

A module has temporal cohesion when it performs a series of operations related in time. An example of a module with temporal cohesion is one named open_old_master_ file, new_master_file, transaction_file, and print_file; initialize_sales_region_table; read_first_transaction_record_and_first_old_master_fi le_record.

4. Procedural Cohesion

A module has procedural cohesion if it performs a series of operations related by the sequence of steps to be followed by the product. An example of a module with procedural cohesion is read_part_number_from_database_and_update_repair_record_ on_maintenance_file.

5. **Communicational Cohesion**

A module has communicational cohesion if it performs a series of operations related by the sequence of steps to be followed by the product and if all the operations are performed on the same data. Two examples of modules with communicational cohesion are update_record_in_database_and_write_ it _to_the_audit_trail , and calculate_new_ trajectory_and_send_ it _to_the_printer

6. **Functional Cohesion**

A module that performs exactly one operation or achieves a single goal has functional cohesion . Examples of such modules are get_temperature_of_furnace , compute_ orbital_of_electron , write_to_diskette , and calculate_sales_commission. A module with functional cohesion often can be reused because the one operation it performs often needs to be performed in other products.

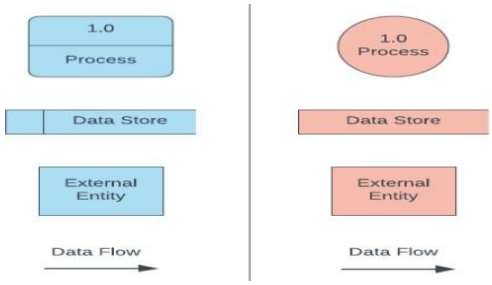
7. **Informational Cohesion**

A module has informational cohesion if it performs a number of operations, each with its own entry point, with independent code for each operation, all performed on the same data structure.

Q 2.What is influence of coupling on maintenance?

Tight coupling increases the maintenance cost as it is difficult and changes to one component would affect all other components that are connected to it. So, code refactoring becomes difficult as you would need to refactor all other components in the connected-chain so that the functionality doesn't break. This process is cumbersome and takes a lot of tedious effort and time.

Q 3..List the symbols used in DFD.


1.0
Process
Data Store
External Entity
Data Flow
1.0
Process
Data Store
External Entity
Data Flow

Q.What do you mean by data flow analysis?

Data flow analysis (DFA) is a classical design technique for achieving modules with high cohesion. It can be used in conjunction with most analysis techniques. Here, DFA is presented in conjunction with structured systems analysis.

The input to the technique is a data flow diagram. A key point is that, once the DFD has been completed, the software designer has precise and complete information regarding the input to and output from the product.

The point at which the input loses the quality of being input and simply becomes internal data operated on by the product is termed the point of highest abstraction of input . The point of highest abstraction of output is similarly the first point in the flow of data at which the output can be identified.

FIGURE 14.1 A data flow diagram showing flow of data and operations of product.

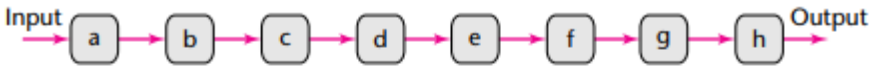
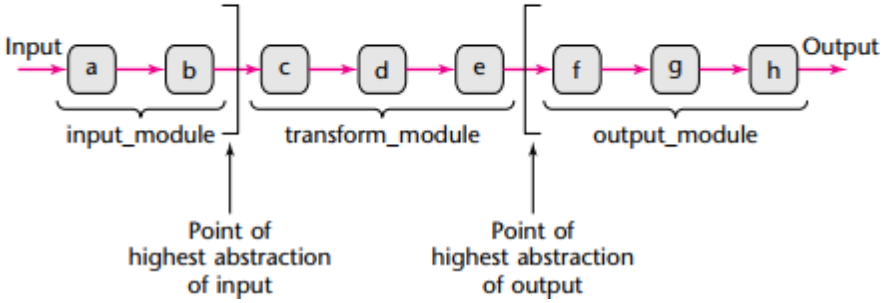


FIGURE 14.2 Points of highest abstraction of input and output.



Data flow analysis is a way of achieving high cohesion. The aim of composite/structured design is high cohesion but also low coupling. To achieve the latter, sometimes it is necessary to make minor modifi cations to the design. For example, because DFA does not take coupling into account.

Q. Explain advantages of following :
i) Reusability
ii) Portability
iii) Inter operability.

A definition of software reuse is the process of creating software systems from predefined

software components.

The advantage of software reuse:

- The systematic development of reusable components.
- The systematic reuse of these components as building blocks to create new systems.

A reusable component may be code, but the bigger benefits of reuse come from a broader and higher-level view of what can be reused. Software specifications, designs, tests cases, data, prototypes, plans, documentation, frameworks, and templates are all candidates for reuse.

Software reuse can cut software development time and costs. The major advantages for software reuse are to:

- Increase software productivity.
- Shorten software development time.
- Improve software system interoperability.
- Develop software with fewer people.
- Move personnel more easily from project to project.
- Reduce software development and maintenance costs.
- Produce more standardized software.
- Produce better quality software and provide a powerful competitive advantage.

2.Portability

The major reason why portability is essential is that the life of a software product generally is longer than the life of the hardware for which it was first implemented. Good software products can have a life of 15 years or more, whereas hardware frequently is changed every 4 years. Therefore, good software can be implemented, over its lifetime, on three or more different hardware configurations.

- *Users* may benefit from portable software because it should be cheaper, and should work in a wider range of environments.
- *Developers* should benefit from portable software because implementations in multiple environments are often desired over the lifetime of a successful product, and these should be easier to develop and easier to maintain.
- *Vendors* should find software portability desirable because ported implementations of the same product for multiple environments should be easier to support, and should increase customer loyalty.
- *Managers* should find advantages in portable software since it is likely to lead to reduced maintenance costs and increased product lifetime, and to simplify product enhancement when multiple implementations exist. However, managers must be convinced that the cost to get the first implementation out the door may not be the only cost that matters!

Q.Explain different requirement elicitation techniques.

Interviewing

The members of the requirements team meet with members of the client organization until they are convinced that they have elicited all relevant information from the client and future users of the target software product.

There are two basic types of questions.

A closed-ended question requires a specific answer. For example, the client might be asked how many salespeople the company employs or how fast a response time is required.

Open-ended questions are asked to encourage the person being interviewed to speak out. For instance, asking the client, "Why is your current software product unsatisfactory?" may explain many aspects of the client's approach to business. Some of these facts might not come to light if the question were closed ended.

Similarly, there are two basic types of interviews, structured and unstructured.

In a structured interview , specific preplanned questions are asked, frequently closed ended. In an unstructured interview , the interviewer may start with one or two prepared closed ended questions, but subsequent questions are posed in response to the answers he or she receives from the person being interviewed.

ii. One way of gaining knowledge about the activities of the client organization is to send a questionnaire to the relevant members of the client organization. This technique is useful when the opinions of, say, hundreds of individuals need to be determined.

iii. A different way of eliciting requirements is to examine the various forms used by the business. For example, a form in a printing works might reflect press number, paper roll size, humidity, ink temperature, paper tension, and so on. The various fields in this form shed light on the flow of print jobs and the relative importance of the steps in the printing process.

iv. Another way of obtaining such information is by direct observation of the users, that is, by members of the requirements team observing and writing down the actions of the employees while they perform their duties.

Q.

List and discuss various impediments to reuse of software.

All too many software professionals would rather rewrite a routine from scratch than reuse a routine implemented by someone else, the implication being that a routine cannot be any good unless they implemented it themselves, otherwise known as the not invented here (NIH) syndrome

Many developers would be willing to reuse a routine provided they could be sure that the routine in question would not introduce faults into the product.

A large organization may have hundreds of thousands of potentially useful components. How should these components be stored for effective later retrieval?

Reuse can be expensive. Legal issues can arise with contract software. In terms of the type of contract usually drawn up between a client and a software development organization, the software product belongs to the client.

Q.

Define and explain 'software portability'. Give some examples. List its advantages.

More precisely, *portability* may be defined as follows: Suppose a product P is compiled by compiler C and then runs on the **source computer**, namely, hardware configuration H under operating system O. A product P' is needed that functionally is equivalent to P but must be compiled by compiler C' and run on the **target computer**, namely, hardware configuration H' under operating system O'. If the cost of converting P into P' is significantly less than the cost of coding P' from scratch, then P is said to be *portable*.

Overall, the problem of porting software is nontrivial because of incompatibilities

The major reason why portability is essential is that the life of a software product generally is longer than the life of the hardware for which it was first implemented. Good software products can have a life of 15 years or more, whereas hardware frequently is changed every 4 years.

Therefore, good software can be implemented, over its lifetime, on three or more different hardware configurations. One way to solve this problem is to buy upwardly compatible hardware. The only expense is the cost of the hardware; the software need not be changed.

Nevertheless, in some cases it may be economically more sound to port the product to different hardware entirely. For example, the first version of a product may have been implemented 7 years ago on a mainframe.

Although it may be possible to buy a new mainframe on which the product can run with no changes, it may be considerably less expensive to implement multiple copies of the product on a network of personal computers, one on the desk of each user. In this instance, if the software has been implemented in a way that would promote portability, then porting the product to the personal computer network makes good financial sense.

Q.Provide details of dynamic modeling.

The dynamic model represents the time–dependent aspects of a system. It is concerned with the temporal changes in the states of the objects in a system. The main concepts are −

- State, which is the situation at a particular condition during the lifetime of an object.
- Transition, a change in the state
- Event, an occurrence that triggers transitions
- Action, an uninterrupted and atomic computation that occurs due to some event, and
- Concurrency of transitions.

A state machine models the behavior of an object as it passes through a number of states in its lifetime due to some events as well as the actions occurring due to the events. A state machine is graphically represented through a state transition diagram.

State

The state is an abstraction given by the values of the attributes that the object has at a particular time period. It is a situation occurring for a finite time period in the lifetime of an object, in which it fulfils certain conditions, performs certain activities, or waits for certain events to occur. In state transition diagrams, a state is represented by rounded rectangles.

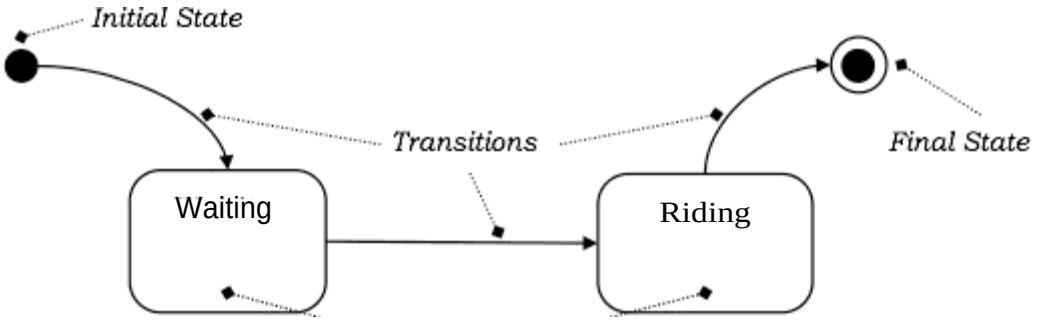
Parts of a state

- **Name** − A string differentiates one state from another. A state may not have any name.
- **Entry/Exit Actions** − It denotes the activities performed on entering and on exiting the state.
- **Internal Transitions** − The changes within a state that do not cause a change in the state.
- **Sub–states** − States within states.

The five parts of a transition are −

- **Source State** − The state affected by the transition.
- **Event Trigger** − The occurrence due to which an object in the source state undergoes a transition if the guard condition is satisfied.
- **Guard Condition** − A Boolean expression which if True, causes a transition on receiving the event trigger.
- **Action** − An un-interruptible and atomic computation that occurs on the source object due to some event.
- **Target State** − The destination state after completion of transition.

Suppose a person is taking a taxi from place X to place Y. The states of the person may be: Waiting (waiting for taxi), Riding (he has got a taxi and is travelling in it), and Reached (he has reached the destination). The following figure depicts the state transition.


Initial State
Transitions
Final State
Waiting
Riding