Software Engineering

Design Principles

Presented by ***Pratanu Mandal***
of CSE 3A
Roll 54

JISCE

Introduction

In this presentation we deal with design principles of software projects.

Software design is a phase in software engineering, in which a blueprint is developed to serve as a base for constructing the software system.

IEEE defines software design as 'both a process of defining, the architecture, components, interfaces, and other characteristics of a system or component and the result of that process.'

Cyclomatic Complexity

Cyclomatic complexity is a software metric (measurement), used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code.

A *control flow graph* (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

The formula for obtaining cyclomatic complexity from CFG is as follows:

$$\boldsymbol{V\,(G) = \; E - N + 2}$$

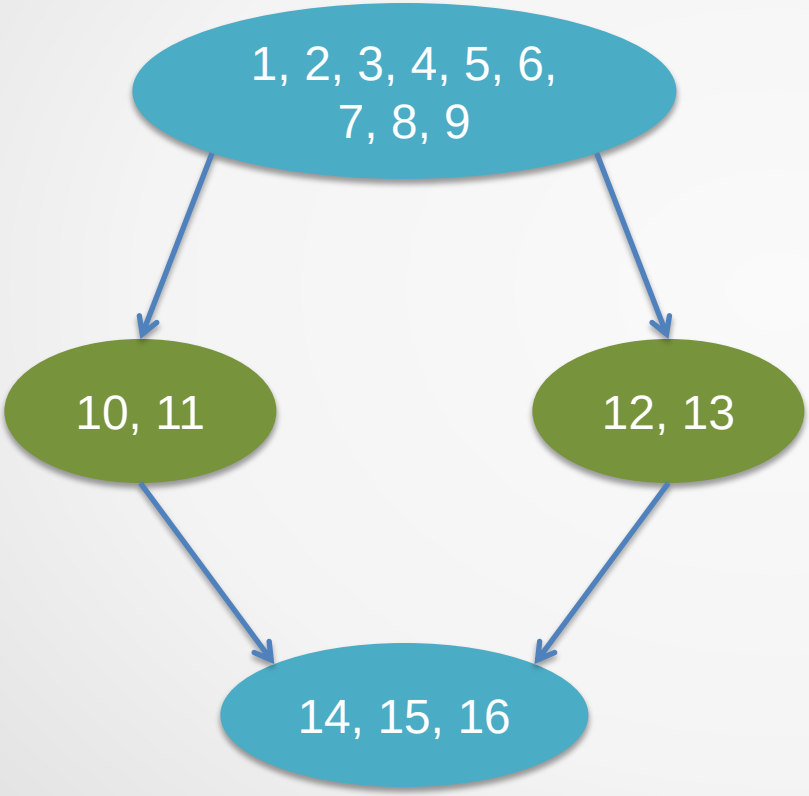
where, $E \rightarrow$ *no. of edges*
$N \rightarrow$ *no. of nodes*
$2 \rightarrow$ *constant*

Cyclomatic Complexity (Example)

Find Cyclomatic Complexity of a C program to find the largest of two numbers.

```
1     #include <stdio.h>
2
3     int main()
4     {
5         int a, b;
6
7         printf("Enter two numbers : ");
8         scanf("%d%d", &a, &b);
9
10        if(a > b)
11            printf("%d is larger \n", a);
12        else
13            printf("%d is larger \n", b);
14
15        return 0;
16    }
```

Cyclomatic Complexity (Example)



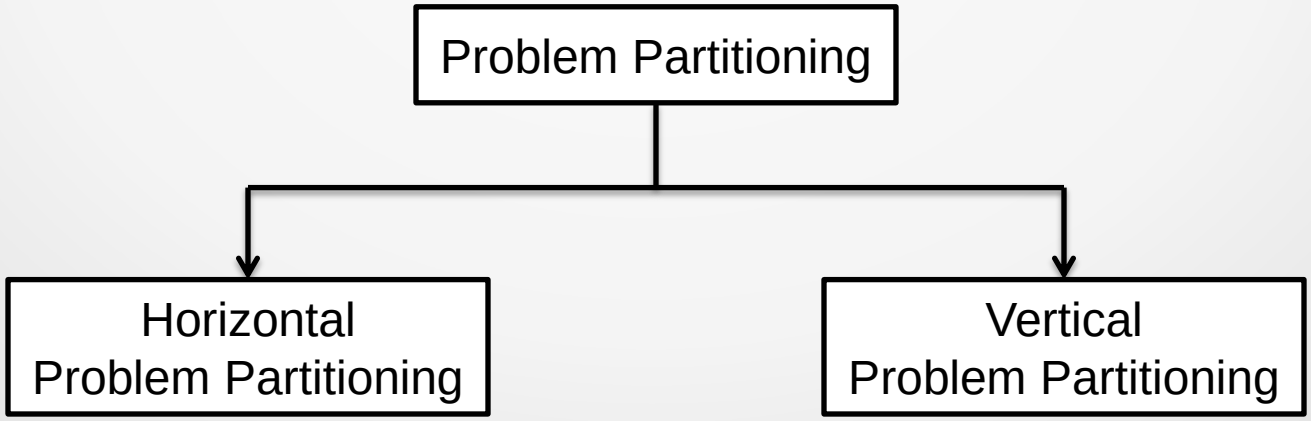
$$V(G) = E - N + 2$$
$$= 4 - 4 + 2$$
$$= \mathbf{2}$$

Problem Partitioning

When solving complex problems, it is impractical to treat them as huge monoliths, due to the limitations of the human mind.

Instead, we use the time-tested principle of **"divide and conquer"**.

The goal is to divide the problem into manageably small pieces (modules) that can be solved separately.

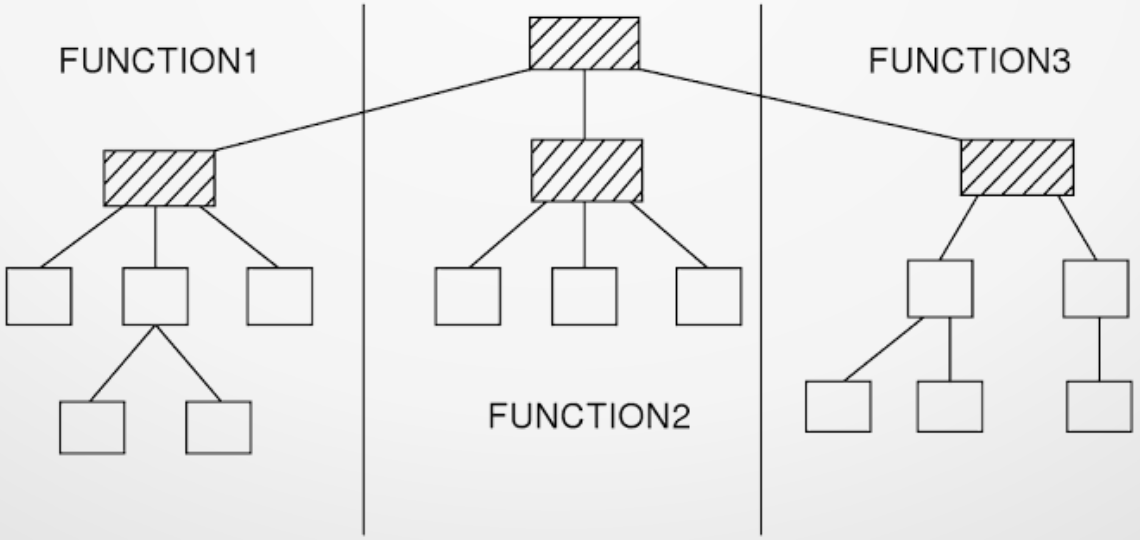


Problem Partitioning (Cont.)

Horizontal Partitioning

Horizontal Partitioning defines separate branches of *modular hierarchy* for each *major program function*.

The simplest approach to horizontal partitioning defines three partitions – Input, Data Transformation, and Output.



Problem Partitioning (Cont.)

Horizontal Partitioning (Cont.)

Advantages

- Software is easier to test
- Software is easier to maintain
- Software is easier to extend
- Propagation of fewer side effects in case of change or errors

Disadvantages

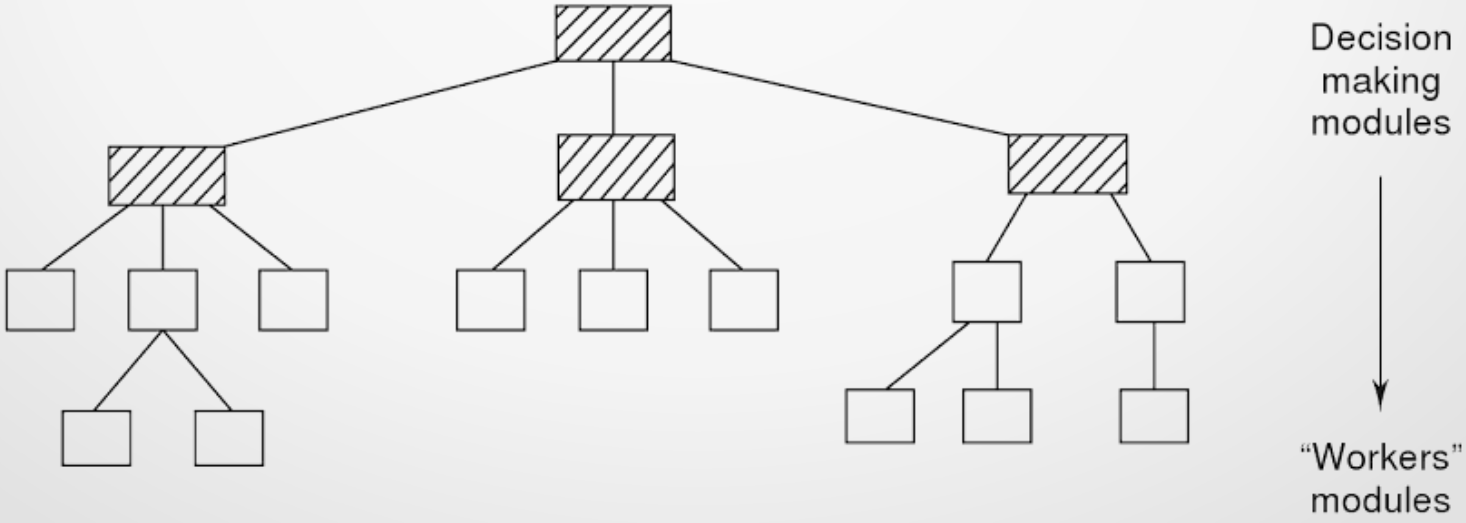
- Causes more data to be passed across module interfaces
- Can complicate overall control of program flow

Problem Partitioning (Cont.)

Vertical Partitioning

Vertical Partitioning, often called factoring, suggests that control and work should be distributed from ***top-down*** in the program structure.

Top level modules should perform control function and do actual processing work. Modules that reside low in the structure should be the workers, performing all input, compilation, and output tasks.



Problem Partitioning (Cont.)

Vertical Partitioning (Cont.)

Advantages

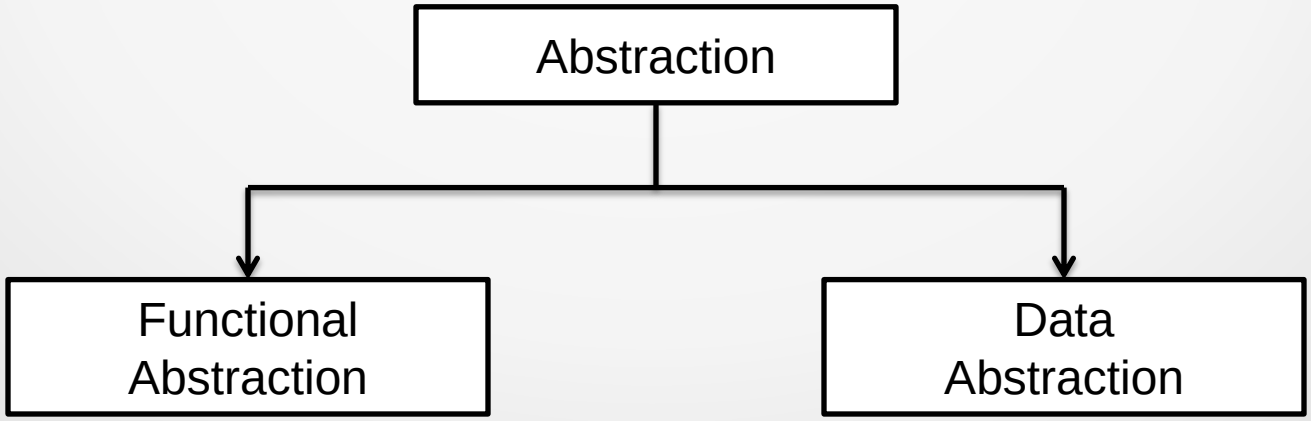
- Good at dealing with changes
- Easy to maintain the changes
- Reduces change impact and error propagation

Abstraction

Abstraction is the concept of ***information hiding***.

It is the process of describing the external behaviour of a component without bothering with the internal details that produce that behaviour.

It allows the designer to consider a component at an abstract level without worrying about the implementation details.



Abstraction (Cont.)

Functional Abstraction

In functional abstraction, a module is specified by the function it performs. It forms the basis of partitioning in *function-oriented approaches*.

For example, a module to compute the log of a value can be abstractly represented by the log function.

Advantages

- It reduces code size
- It avoids cut-and-paste
- Bugs can be fixed in one place instead of many

Abstraction (Cont.)

Data Abstraction

Data abstraction forms the basis for *object-oriented design*. In this design, data is treated as objects with some predefined operations on them.

The operations defined on a data object are the only operations that can be performed on it. From outside an object, the internals of the object are hidden; only the operations on the object are visible.

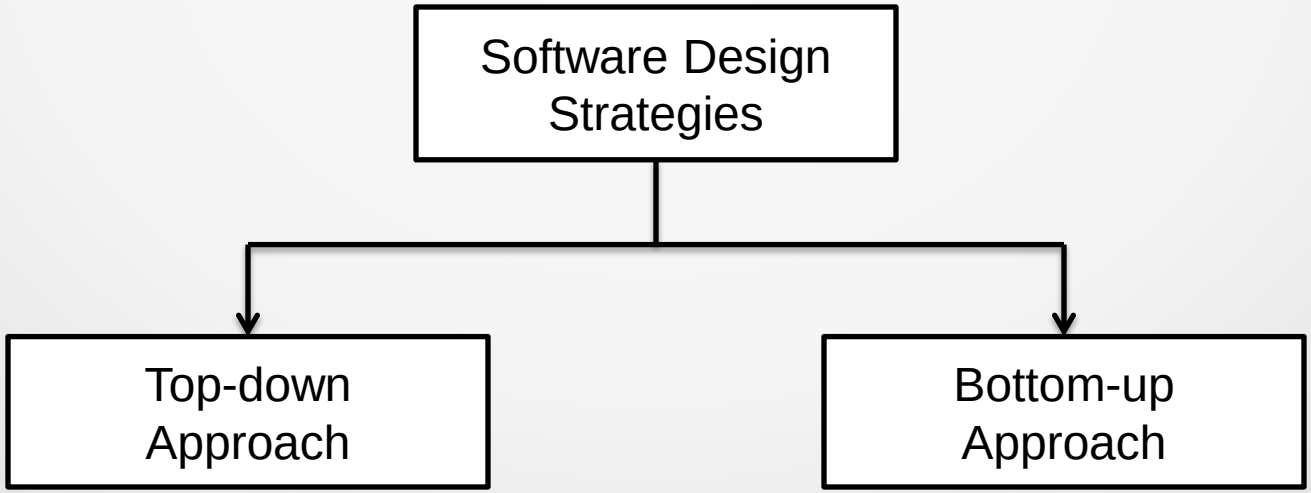
Advantages

- Reduction of complexity
- Development of large programs possible
- Easier to maintain independently

Software Design Strategies

A system consists of components, which have components of their own. Indeed a system is a hierarchy of components.

The highest-level component correspond to the total system. To design such a hierarchy there are two possible approaches: top-down and bottom-up.



Software Design Strategies (Cont.)

Top-down Approach

A top-down approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved.

Top-down design methods often result in some form of ***stepwise refinement***. Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly.

Most design methodologies are based on the top-down approach.

Software Design Strategies (Cont.)

Bottom-up Approach

A bottom-up approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these low-level components.

Bottom-up methods work with ***layers of abstraction***. Starting from the very bottom, operations that provide a layer of abstraction are implemented. The operations of this layer are then used to implement more powerful operations and a still higher level of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

Software Design Strategies (Cont.)

A top-down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch.

A bottom-up approach is more suitable if a system is to be built from an existing system.

Pure top-down or pure bottom-up approaches are often not practical. Instead, some combination of both of them is often used for practical applications.

Conclusion

Through this presentation we have learnt about some of the design principles of software engineering and their necessity.

Following these design principles enable us to create better and more manageable and scalable software in the real world.

It is my hope that we shall benefit from what we have learnt here by applying them in our own software projects.

References

Books :

- “An Integrated Approach to Software Engineering” by “Pankaj Jalote”
- "Software Engineering" by "Bharat Bhushan Agarwal, Sumit Prakash Tayal"

Online Sources :

- https://en.wikipedia.org/wiki/Cyclomatic_complexity
- http://ecomputernotes.com/software-engineering/principles-of-software-design-and-concepts

Any Questions?

Thank you.

Presented by ***Pratanu Mandal***
of **CSE 3A**
Roll **54**