Software Design

Introduction

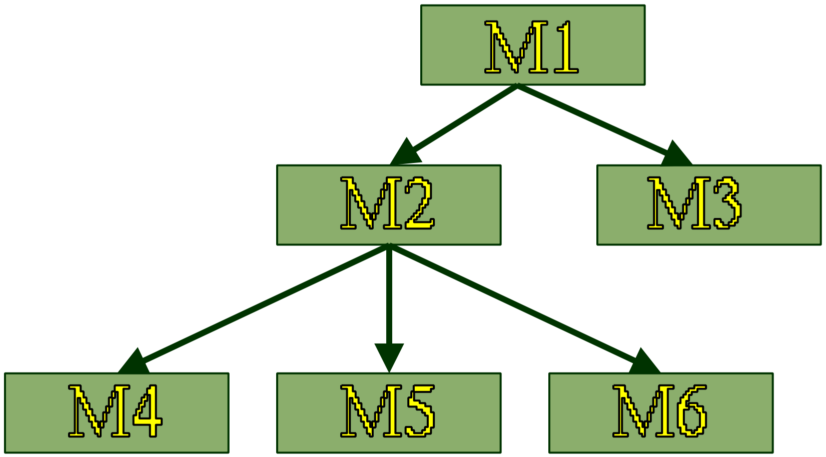
Design phase transforms SRS document:

into a form easily implementable in some programming language.

Items Designed During Design Phase

1. module structure,
2. control relationship among the modules
3. interface among different modules,
   data items exchanged among different modules,
4. data structures of individual modules,
5. algorithms for individual modules.

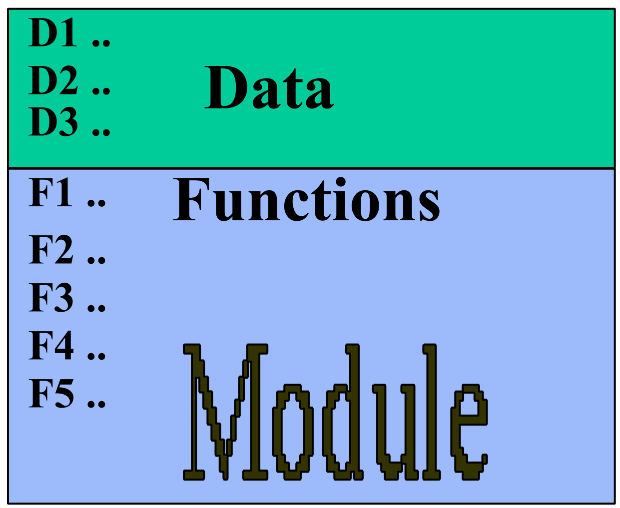
Module Structure



Introduction

A module consists of:

1. several functions
2. associated data structures.



Introduction

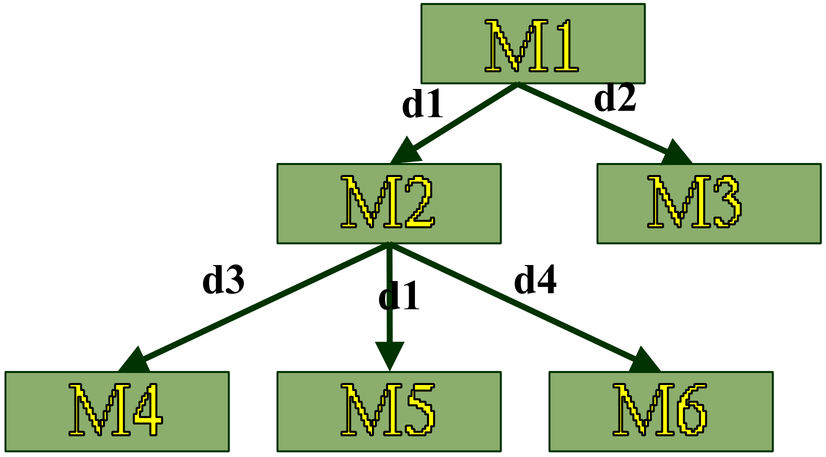
Design activities are usually classified into two stages:

1. preliminary (or high-level) design
2. detailed design.

High-level design

Identify:

1. modules
2. control relationships among modules
3. interfaces among modules.



High-level design

- The outcome of high-level design:
  - program structure (or software architecture).

Detailed design

For each module, design:

1. data structure
2. algorithms

Outcome of detailed design:

module specification.

What Is Good Software Design?

- Should implement all functionalities of the system correctly.
- Should be easily understandable.
- Should be efficient.
- Should be easily able to change, i.e. easily maintainable.

What Is Good Software Design?

Understandability of a design is a major issue:

- determines goodness of design:
- a design that is easy to understand: also easy to maintain and change.

What Is Good Software Design?

Unless a design is easy to understand, tremendous effort needed to maintain it.

We already know that about 60% effort is spent in maintenance.

If the software is not easy to understand:

maintenance effort would increase many times.

Understandability

Use consistent and meaningful names
  for various design components,

Design solution should consist of:
  a cleanly decomposed set of modules (modularity),

Different modules should be neatly arranged in a hierarchy:
  in a neat tree-like diagram.

Modularity

Modularity is a fundamental attributes of any good design.

Decomposition of a problem cleanly into modules:

1. Modules are almost independent of each other
2. divide and conquer principle.

Modularity

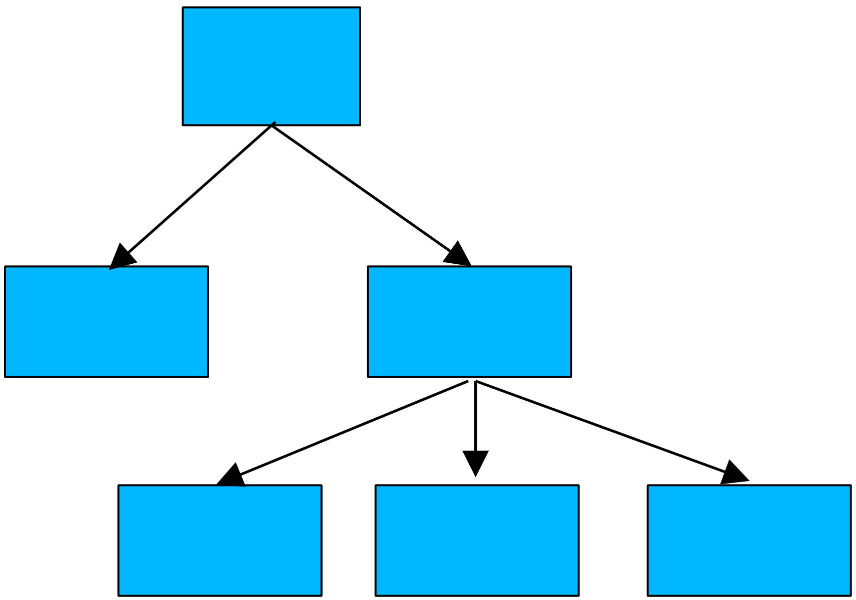
If modules are independent:

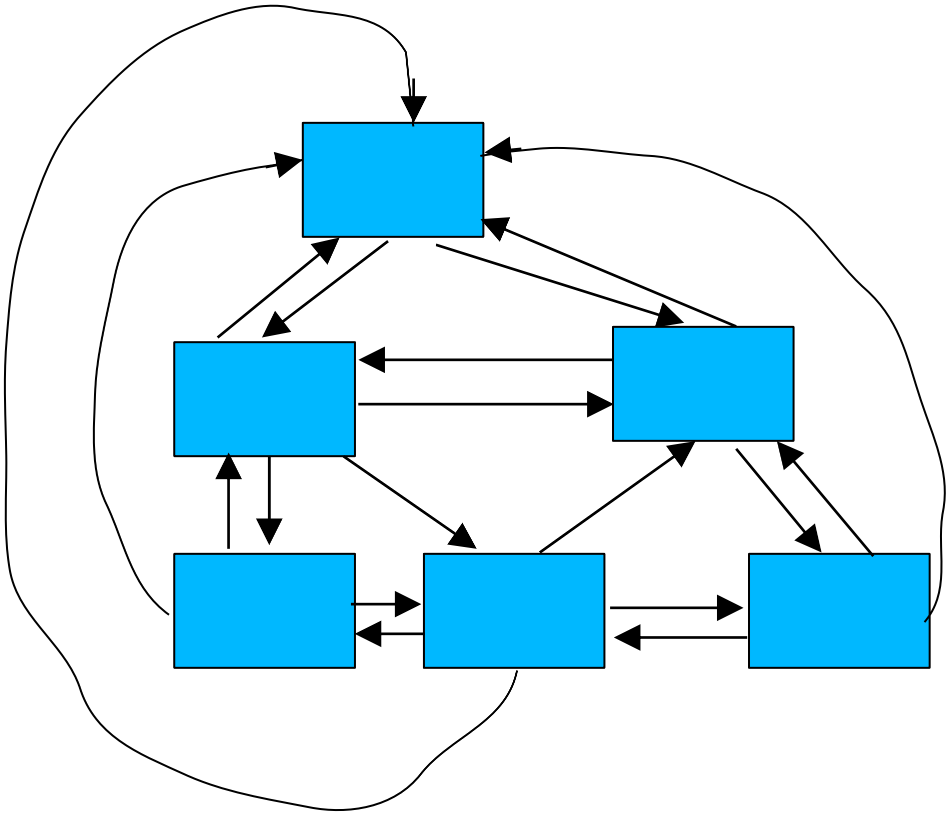
- modules can be understood separately.
- reduces the complexity greatly.

To understand why this is so,

- remember that it is very difficult to break a bunch of sticks but very easy to break the sticks individually.

Example of Cleanly and Non-cleanly Decomposed Modules





Modularity

In technical terms, modules should display:

1. high cohesion
2. low coupling.

Modularity

Neat arrangement of modules in a hierarchy means:

1. low fan-out

   Fan-out: a measure of the number of modules directly controlled by given module.

2. abstraction

Cohesion and Coupling

Cohesion is a measure of:

- functional strength of a module.

- A cohesive module performs a single task or function.

Coupling between two modules:

- a measure of the degree of interdependence or interaction between the two modules.

Cohesion and Coupling

A module having high cohesion and low coupling:

functionally independent of other modules:

A functionally independent module has minimal interaction with other modules.

Advantages of Functional Independence

Better understandability and good design:

Complexity of design is reduced,

- Different modules easily understood in isolation:
- modules are independent

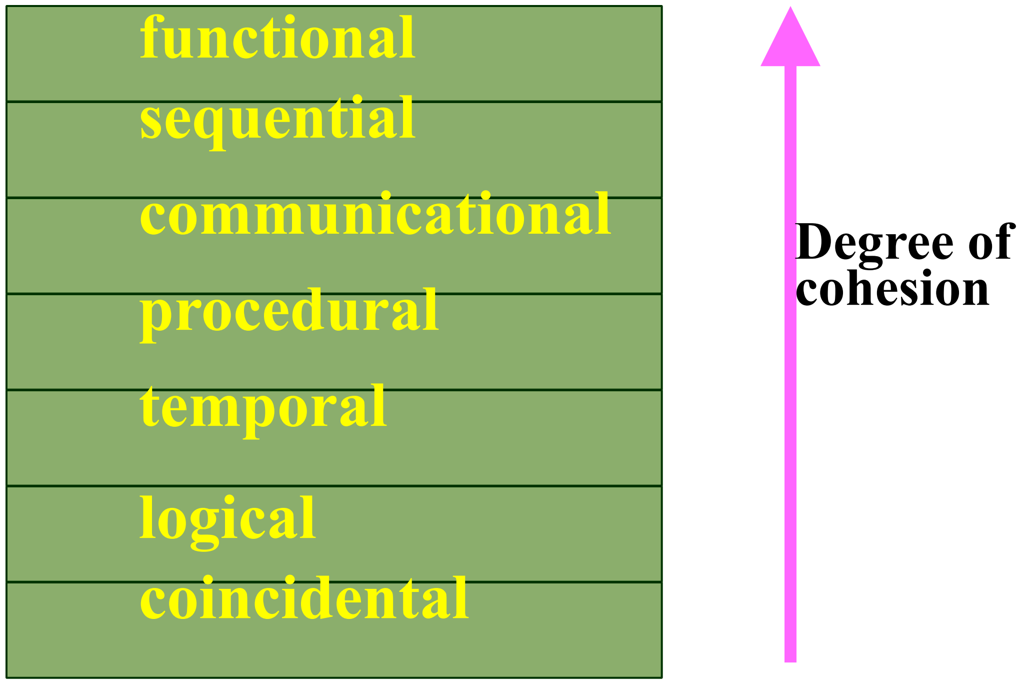
Advantages of Functional Independence

Functional independence reduces error propagation.

- degree of interaction between modules is low.
- an error existing in one module does not directly affect other modules.

Reuse of modules is possible.

Classification of Cohesiveness



Coincidental cohesion

The module performs a set of tasks:

which relate to each other very loosely, if at all.

- the module contains a random collection of functions.
- functions have been put in the module out of pure coincidence without any thought or design.

Logical cohesion

All elements of the module perform similar operations:
e.g. error handling, data input, data output, etc.

Temporal cohesion

The module contains tasks that are related by the fact that all the tasks must be executed in the same time span.

Procedural cohesion

If the set of functions of the module all part of a procedure (algorithm) in which certain sequence of steps have to be carried out in a certain order for achieving an objective,

e.g. the algorithm for decoding a message.

Communicational cohesion

If All functions of the module Refer to the same data structure,

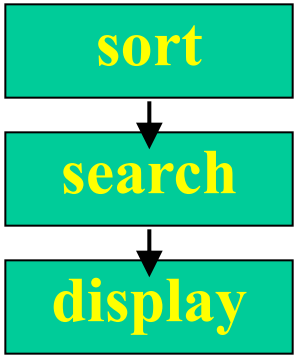
Example:

the set of functions defined on an array or a stack.

Sequential cohesion

If the Elements of a module forms different parts of a sequence, output from one element of the sequence is input to the next.

Example:



Functional cohesion

If the Different elements of a module cooperate to achieve a single function.

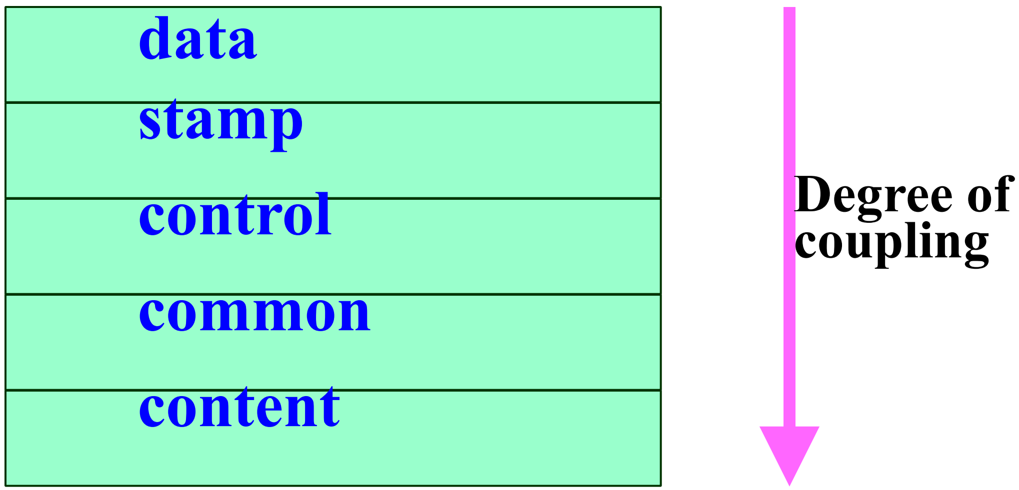
Coupling

Coupling indicates:

- how closely two modules interact or how interdependent they are.

- The degree of coupling between two modules depends on their interface complexity.

Classes of coupling



Data coupling

Two modules are data coupled,

if they communicate by an elementary data item that is passed as a parameter between the two, eg an integer, a floa, character etc.

Stamp coupling

Two modules are stamp coupled,

if they communicate via a composite data item:

- such as a record in PASCAL

or a structure in C.

Control coupling

It exists between two modules.

If Data from one module is used to direct order of instruction execution in another.

Example of control coupling:

- a flag set in one module and tested in another module.

Common Coupling

Two modules are common coupled, if they share some global data items.

Content coupling

Content coupling exists between two modules:

if they share code,

e.g, branching from one module into another module.

The degree of coupling increases from data coupling to content coupling.

Neat Hierarchy

Control hierarchy represents organization of modules. control hierarchy is also called <u>program structure.</u>

Characteristics of Module Structure

Depth:

number of levels of control

Width:

overall span of control.

Fan-out:

a measure of the number of modules directly controlled by given module.

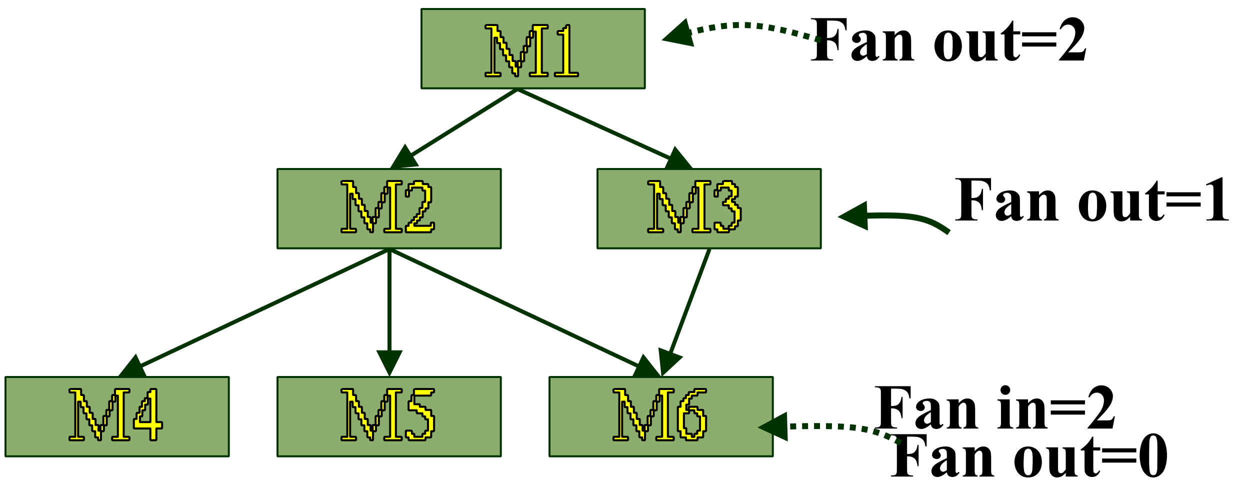
Characteristics of Module Structure

Fan-in:

indicates how many modules directly invoke a given module.

High fan-in represents code reuse and is in general encouraged.

Module Structure



Goodness of Design

A design having modules:

- with high fan-out numbers is not a good design:
- a module having high fan-out lacks cohesion.

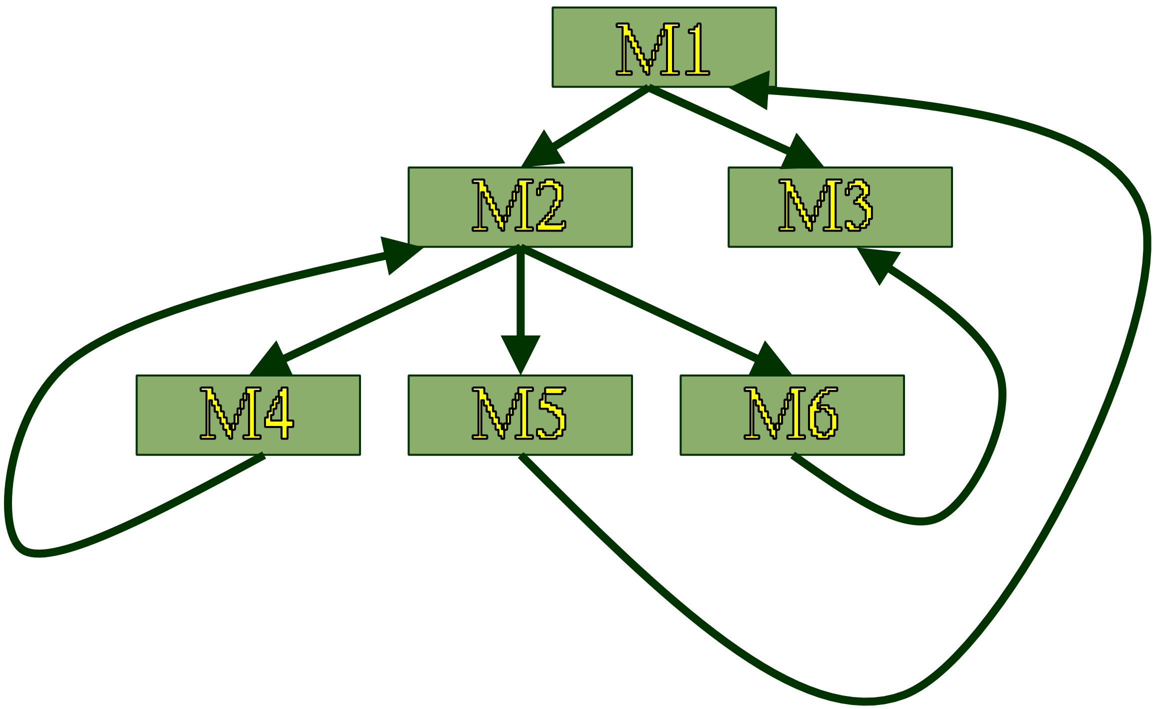
Visibility and Layering

A module A is said to be visible by another module B,

if A directly or indirectly calls B.

The layering principle requires modules at a layer can call only the modules immediately below it.

Bad Design



Abstraction

- The principle of abstraction requires:
  - lower-level modules do not invoke functions of higher level modules.
  - Also known as layered design.

High-level Design

High-level design maps functions into modules {fi} {mj} such that:

1. Each module has high cohesion
2. Coupling among modules is as low as possible
3. Modules are organized in a neat hierarchy

Design Approaches

Two fundamentally different software design approaches:

Function-oriented design

Object-oriented design

Function-Oriented Design

A system is looked upon as something that performs a set of functions.

Starting at this high-level view of the system:

- each function is successively refined into more detailed functions.
- Functions are mapped to a module structure.

Function-Oriented Design

Each subfunction:

split into more detailed subfunctions and so on.

Object-Oriented Design

System is viewed as a collection of objects (i.e. entities).

- each object manages its own state information.

Object-Oriented Design Example

Library Automation Software:

- each library member is a separate object with its own data and functions.

- Functions defined for one object:

  cannot directly refer to or change data of other objects.

Object-Oriented Design

Objects have their own internal data: defines their state.

Similar objects constitute a class.

- each object is a member of some class.
- Classes may inherit features from a super class.

Conceptually, objects communicate by message passing.

Object-Oriented versus Function-Oriented Design

Unlike function-oriented design,

in OOD the basic abstraction is not functions such as "sort", "display", "track", etc.,

but real-world entities such as "employee", "picture", "machine", "radar system", etc.

Object-Oriented versus Function-Oriented Design

In OOD:

- software is not developed by designing functions such as:
  1. update-employee-record,
  2. get-employee-address, etc.

  but by designing objects such as:
  1. employees,
  2. departments, etc.

Summary

We characterized the features of a good software design by introducing the concepts of:

fan-in, fan-out,
cohesion, coupling,
abstraction, etc.

Summary

Two fundamentally different approaches to software design:

- function-oriented approach
- object-oriented approach