System Design

Analysis and Design

- **Analysis**
  - ***do the right thing***

    (What exactly we want to do? We should be clear about that)
- **Design**
  - ***Do the thing right***

    (Whatever we want to do, we should do that correctly)

Analysis

- Analysis is a broad term. In Software development, we are primarily concerned with **two** forms of analysis.
- **Requirements Analysis** is discovering the requirements that a system must meet, in order to be successful.
- **System (Process/Object/Module) Analysis** is investigating the process/object in a domain to discover the information which is important to meet the requirements.

Design

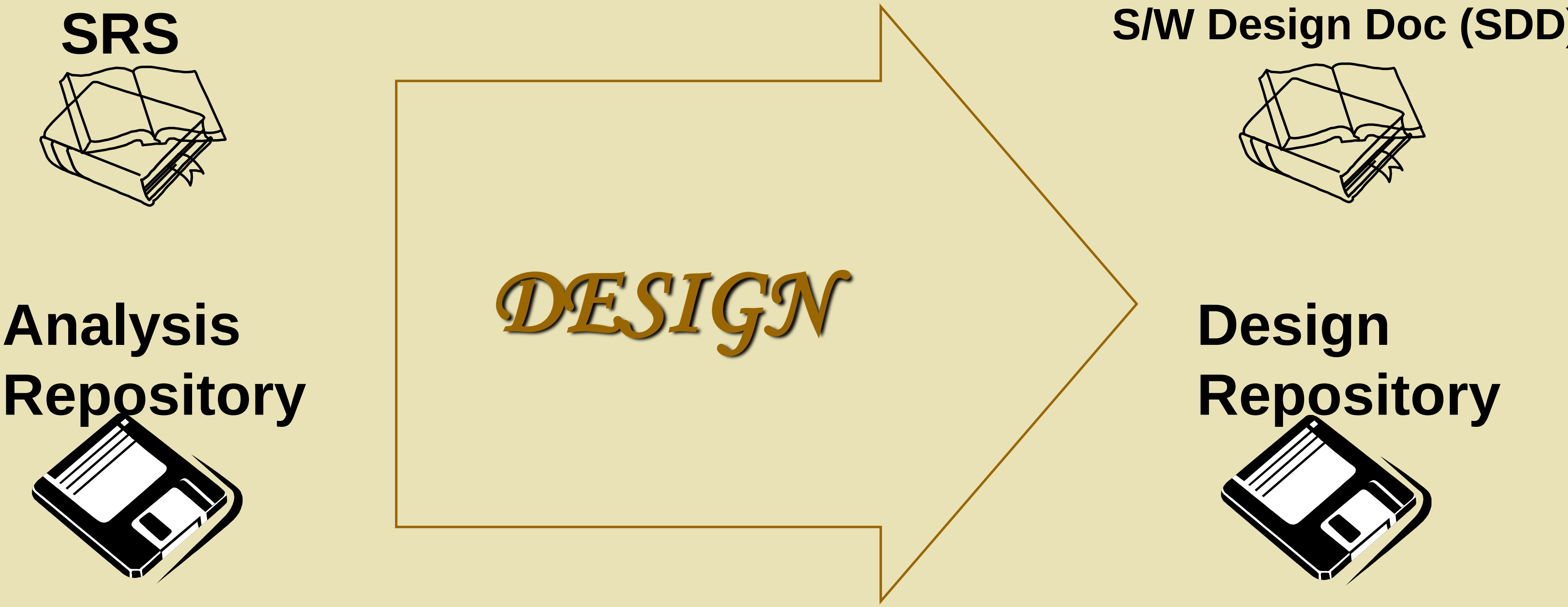
- Design emphasizes a **conceptual** solution that fulfills the requirements.
- A design is not an implementation, although a good design can be implemented very easily when it is complete.
- There are **subsets** of design, including architectural design, object/module design, and database design.

Overview of Design

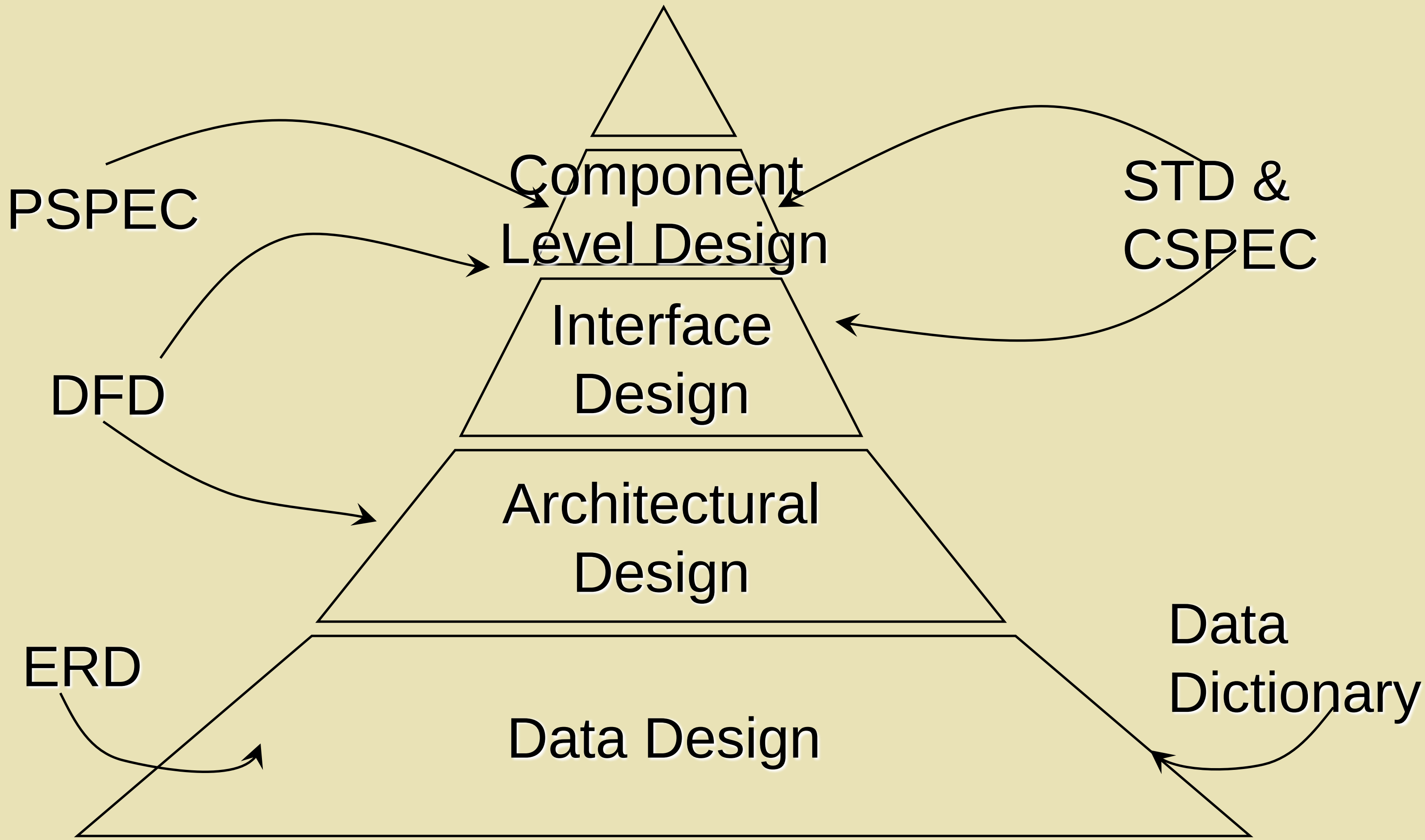
A meaningful engineering representation of something that is to be built.

- A **design** is a solution to a problem that is **independent** of any particular implementation (i.e. any **specific language**).
- *Why is it important?* A house would never be built without a blueprint. Why should software? Without design the system may fail with small changes, is difficult to test and cannot be assessed for quality.
- *What is the work product?* A design specification document.
- Design is difficult because design is an **abstraction** of the solution which has not yet been created/implemented.

Design Phase



Design Model



Items Designed During Design Phase

- module structure,
- control relationship among the modules
  - call relationship or invocation relationship
- interface among different modules,
  - data items exchanged among different modules,
- data structures of individual modules,
- algorithms for individual modules.

Definition

The practice of taking a specification of externally observable behavior and adding details needed for actual computer system implementation, including human interaction, task/function management, and data management details.

Coad and Yourdon (1991)

Definition

... a process of inventing and selecting modules that meet the objectives for a software system. Input includes an understanding of the following:

1. Requirements.
2. Any constraints.
3. Design criteria.

Stevens (1991)

The output of the design effort is composed of the following:

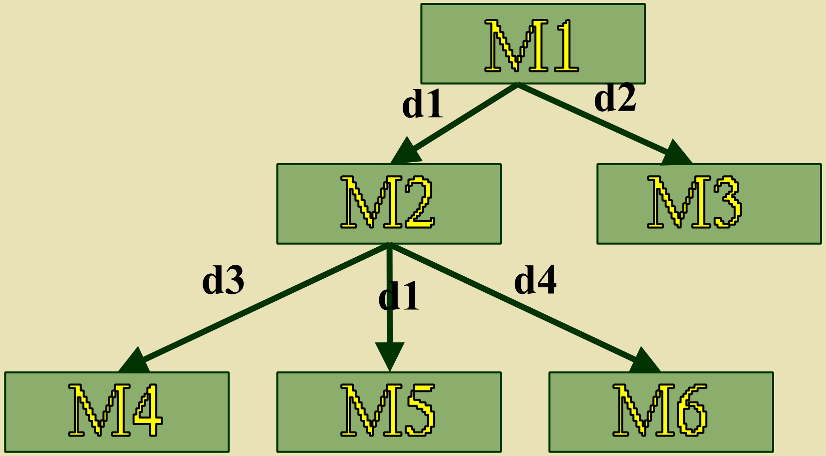
1. ***An architectural design, which shows how components/pieces are interrelated.***
2. ***Module design is translated from information flow model (DFD).***
3. ***Interface design: SW communicating within itself, with incorporated systems and with users.***
4. ***Component design: Structural elements of SW architecture into procedural description of software components (algorithm).***

Broad Characteristics (contd.)

- Design activities are usually classified into two stages:
  - preliminary (or high-level) design
  - detailed design.
- Meaning and scope of the two stages:
  - vary considerably from one methodology to another.

High-level design

- Identify:
  - **modules**
  - control relationships among modules
  - interfaces among modules.



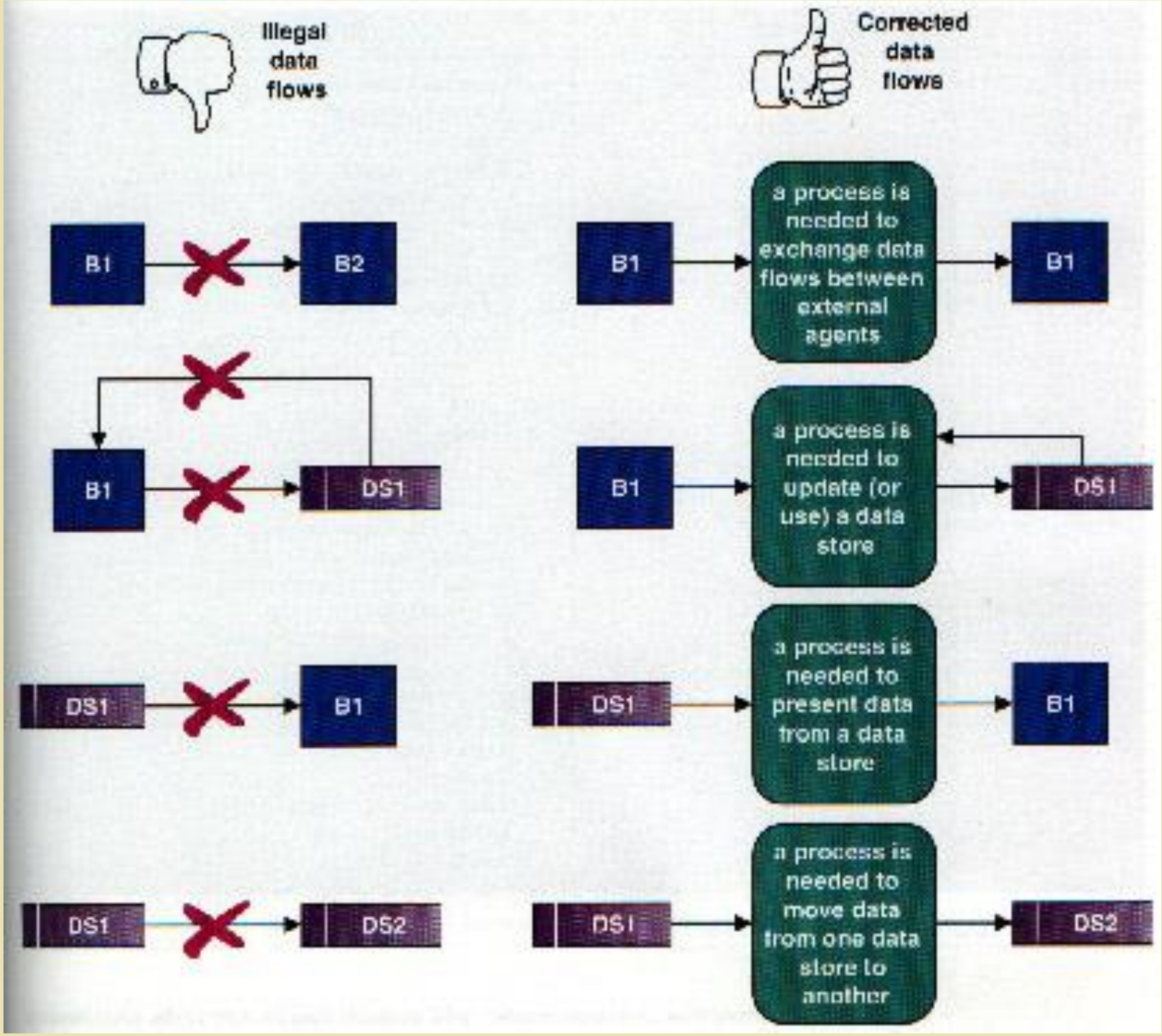
High-level design

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

Detailed design

- For each module, design:
  - Data-structure
  - algorithms
- Outcome of detailed design:
  - Module/process specification.

Illegal Data Flows in DFD



The Data Dictionary

- The data dictionary is an organized listing of all data elements pertinent to the system with precise rigorous definitions so that both user and SA will have common understanding of all data elements.
- It describes:
  - meaning and composition of data flows, data stores
  - relevant values and units of elementary data elements
  - details of relationships between stores that are highlighted in a ERD

D D…

=	is composed of
+	and
[|]	either / or
{ }	repetition of (string)
()	optional data
---	for comments

Data Dictionary Notation

Example:

customer-name = title + first-name + (middle-name) + last-name

title = [Mr. | Miss | Ms. | Mrs.| Dr. | Prof.]

first-name = {legal-character}

order = customer-name + shipping-addr + {item}

Fundamental Concepts

- Stepwise Refinement
- Abstraction
- Software Architecture
- Program Structure
- Data Structure
- Modularity
- Software Procedure
- Information Hiding

Stepwise Refinement

Stepwise refinement is a top-down approach where a system is refined as a hierarchy of increasing levels of detail.

This process may be begun during requirements analysis and conclude when the detail of the design is sufficient for conversion into code. Processing procedures and data structures are likely to be refined in parallel.

Abstraction

- Abstraction is a means of describing a program function, at an appropriate level of detail.
- Highest level of abstraction - requirements analysis.
- Lowest level of abstraction – implementation/programming.

Abstraction and Refinement

- **Abstraction:**
  - “*Abstraction permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details*..”
  - Software Engineering is a process of refining the abstractions
  - Modern programming languages allow for abstraction, e.g. abstract data types
  - Types: data, procedural and control
- **Stepwise refinement**
  - gradual top-down elaboration of detail
  - Abstraction and refinement are complementary. Abstraction suppresses low-level detail while refinement gradually reveals it.

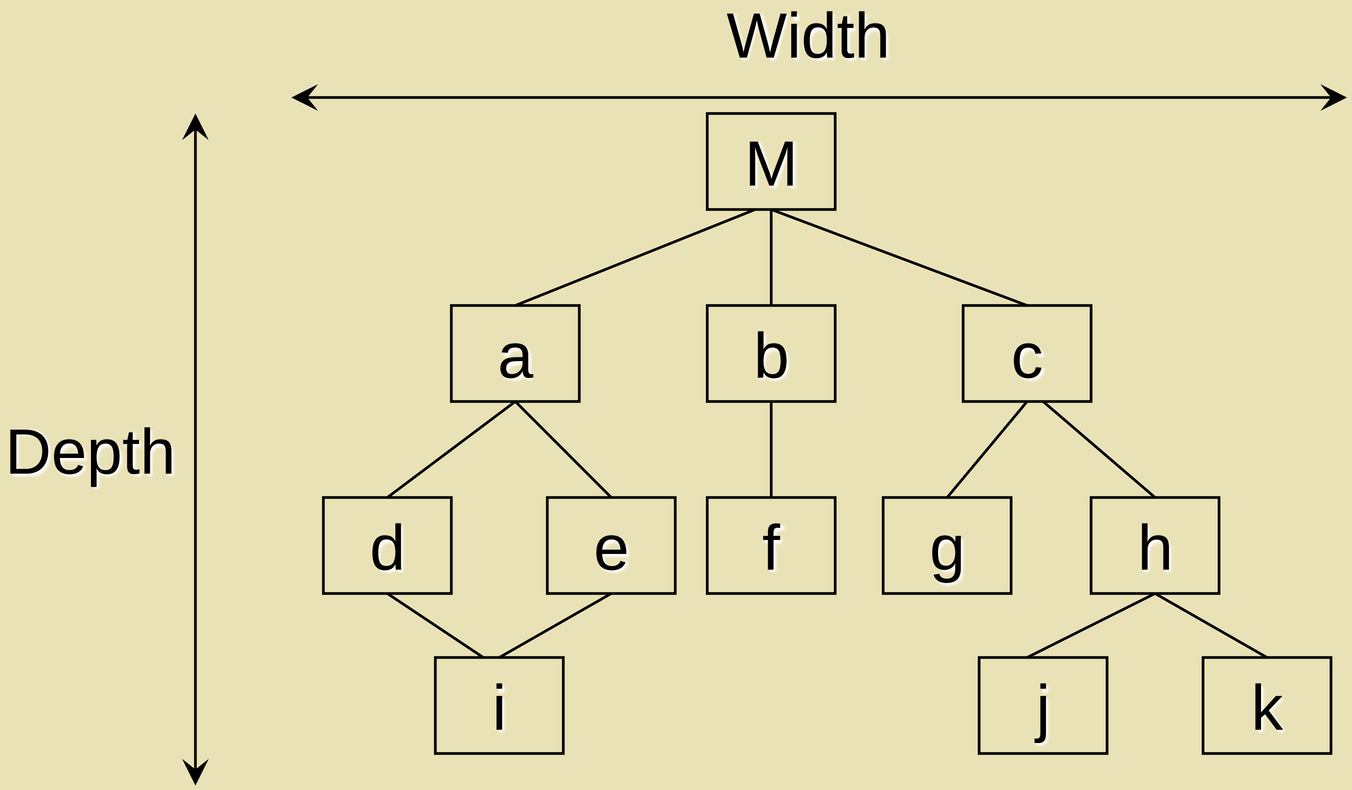
Characteristics of Abstraction

1. Which type of details should be included and which should be excluded? (Example, should the model include structural or behavioral / static or dynamic details?)
2. Independent of type, how much detail should be included? What is the purpose of the abstraction? The purpose of the abstraction will dictate the type and amount of detail to include.

Software Architecture

While refinement is about the level of detail, architecture is about structure. The architecture of the procedural and data elements of a design represents a software solution for the real-world problem defined by the requirements analysis.

Control Hierarchy



Data Structure

Data structure dictates the organization, access methods, degree of associativity, and processing alternatives for problem-related information. Classic data structures include scalar, sequential, linked-list, n-dimensional, and hierarchical. Data structure, along with program structure, makes up the software architecture.

Modularity

- Modularity derives from the architecture.
- Modularity is a logical partitioning of the software design that allows complex software to be manageable for purposes of implementation and maintenance.
- The logic of partitioning is based on related functions, implementation considerations, data links, or other criteria.
- Modularity does imply interface overhead related to information exchange between modules and execution of modules.

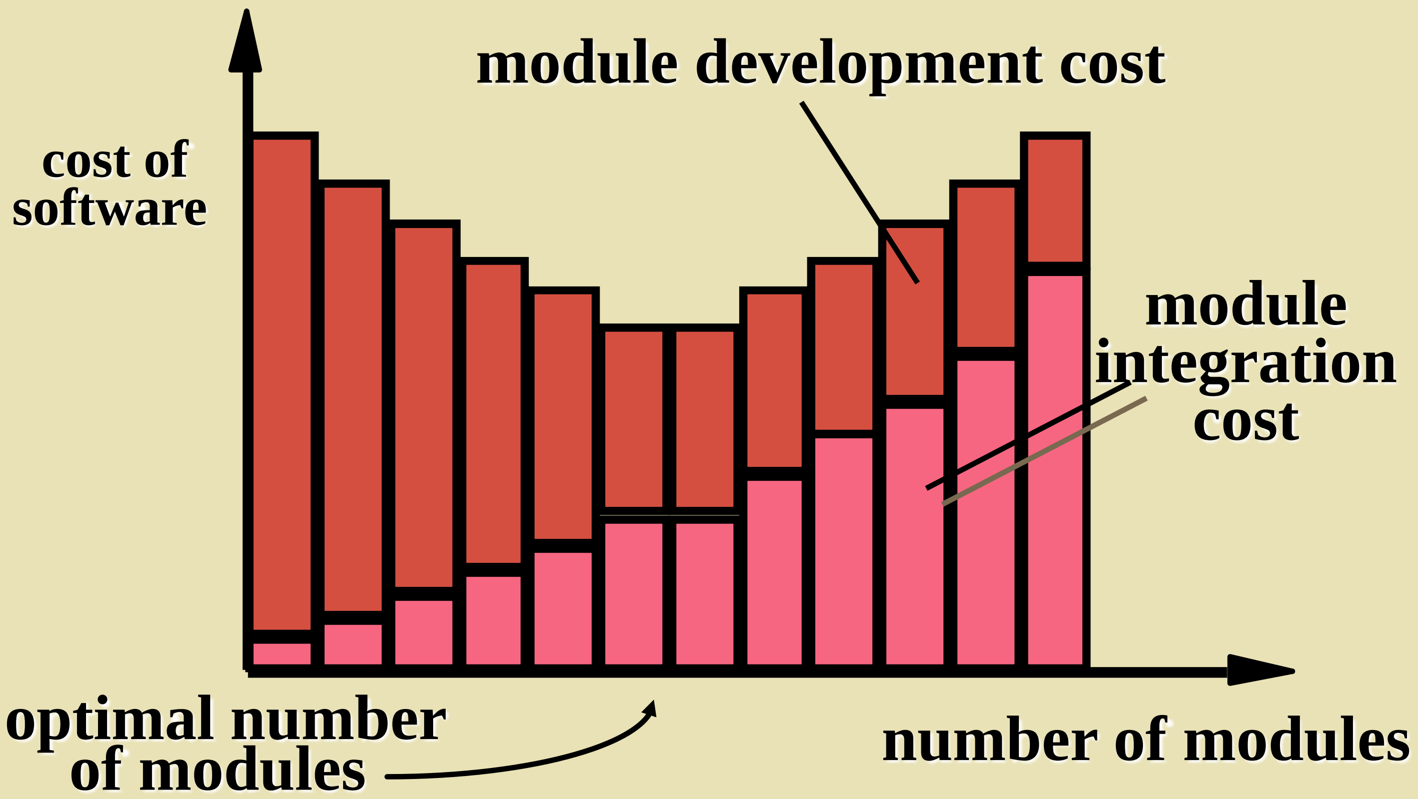
Benefits of Modularity

- Easier to build; easier to change and easier to fix the errors.
- “Modularity is the single attribute of software that allows a program to be intellectually manageable”.
- Don’t overdo it. Too many modules makes integration complicated.
- Sometimes the code may be monolithic (e.g. real-time and embedded software) but the design still shouldn’t be.
- Effective modular design is achieved by developing “single minded” (highly cohesive) modules with a “dislike” to excessive interaction (low coupling).

- Management: Partition the overall development effort – Divide and conquer (**easy to manage**)
- • Evolution: Decouple parts of a system so that changes to one part are isolated from changes to other parts (**other modules can be added easily**).
- – Principle of directness (clear allocation of requirements to modules, ideally one requirement (or more) maps to one module)
- – Principle of continuity/locality (small change in requirements triggers a change to **one module** only)
- • Understanding: Permit system to be understood easily.

Modularity: Trade-offs

What is the "right" number of modules for a specific software design ?



Modularity

p1,p2…pN be the processes

C(p1) > C(p2) ➔ E(p1) > E(p2)

C(p1+p2) > C(p1)+C(p2) ➔ E(p1+p2) > E(p1)+E(p2)

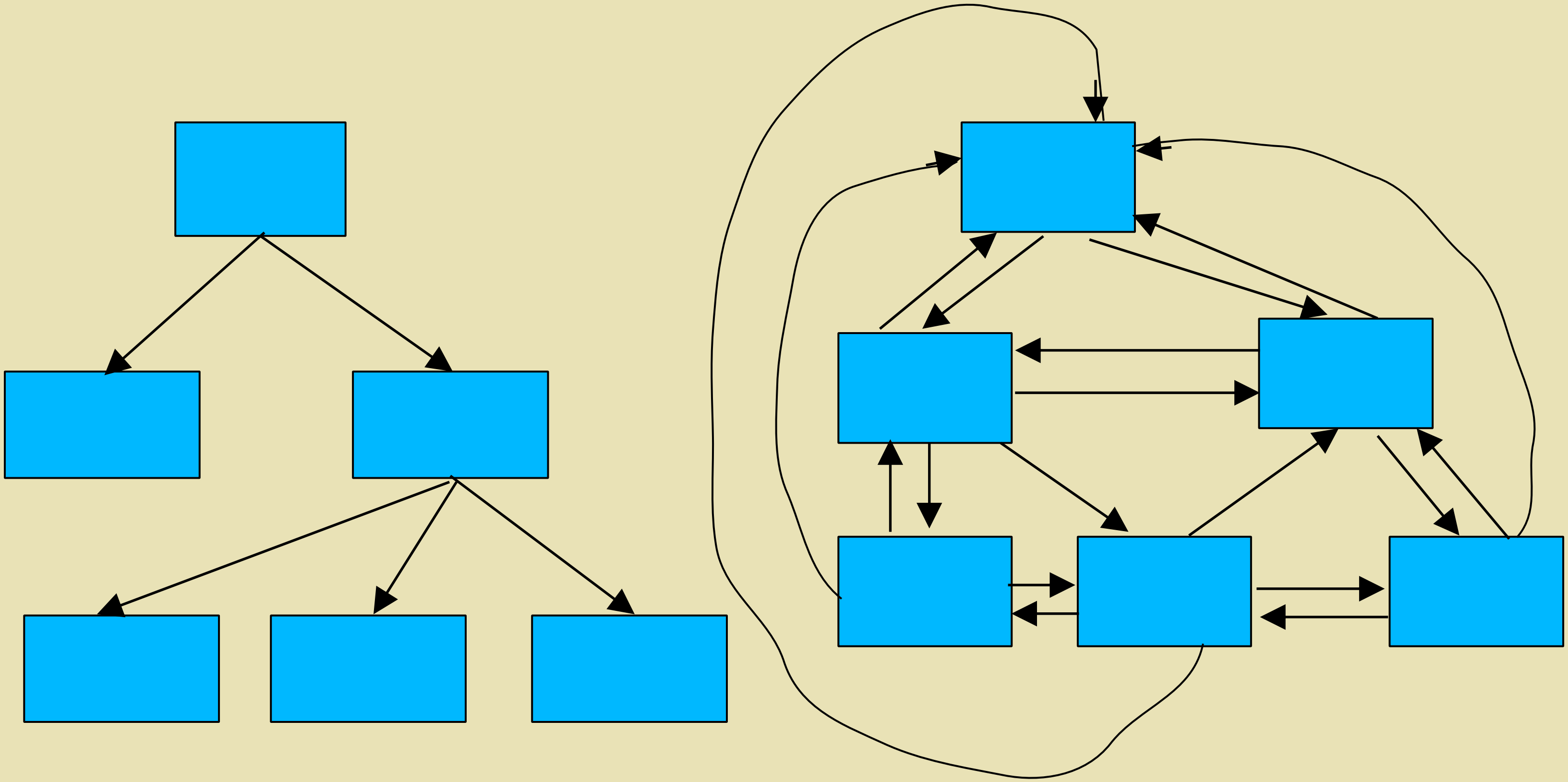
(C-complexity, E-efforts)

- Modular Decomposability
- Modular Composability
- Modular understandability
- Modular continuity
- Modular protection
- Modular testability

Modularity Support

- Design method supports effective modularity if it evidences:
  - Decomposability - a systematic mechanism for decomposing the problem
  - Composability - able to reuse modules in a new system
  - Understandability - the module can be understood as a standalone unit
  - Continuity - minimizes change-induced side effects
  - Protection - minimizes error-induced side effects
  - Testability – how easily can we test the components and the whole system

Example of Cleanly and Non-cleanly Decomposed Modules



Modularity

- In technical terms, modules should display:
  - high cohesion
  - low coupling.
- We must understand
  - cohesion and coupling.

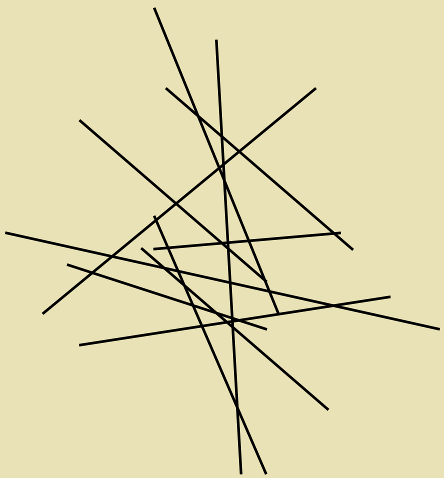
Cohesion and Coupling

- 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.

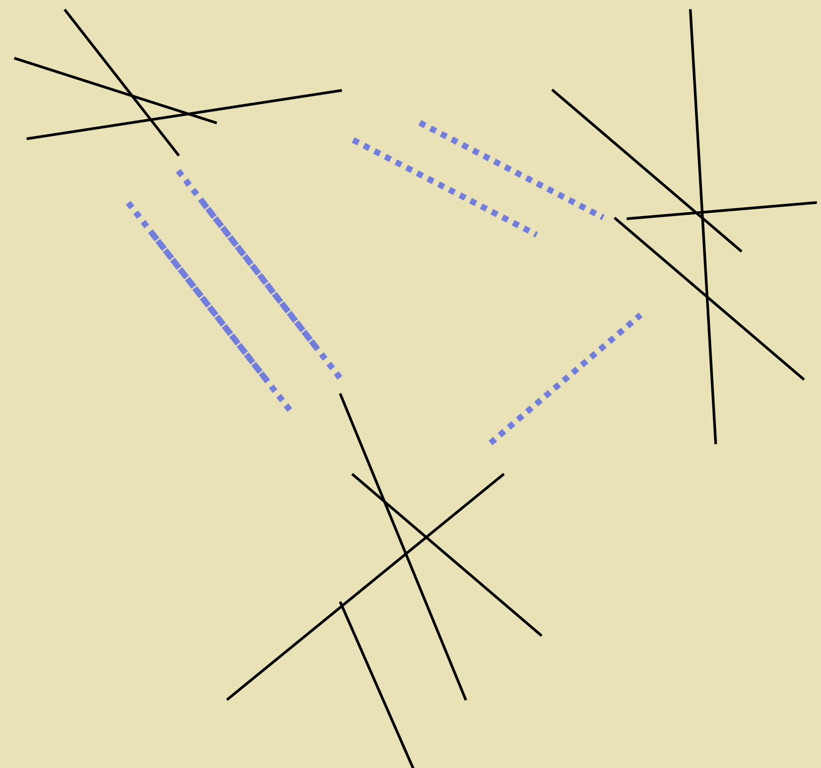
Cohesion and Coupling

- A module having high cohesion and low coupling:
  - functionally independent of other modules:
    - A functionally independent module has minimal interaction with other modules.

Coupling…

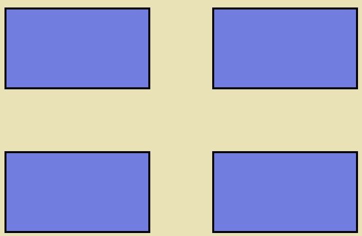


High coupling

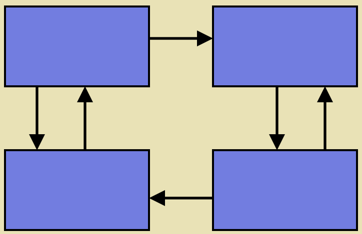


Low coupling

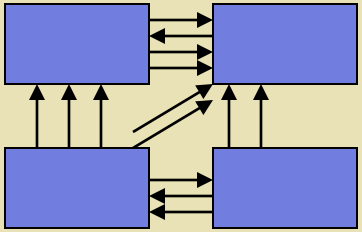
Coupling: Degree of inter-dependence among components



No dependencies



Loosely coupled-some dependencies



Highly coupled-many dependencies

High coupling makes modification of parts of the system very difficult, e.g., modifying a component affects all the components to which the component is connected.

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.

Advantages of Functional Independence…

- A functionally independent module:
  - can be easily taken out and reused in a different program.
    - each module does some well-defined and precise function
    - the interfaces of a module with other modules are simple and minimal.

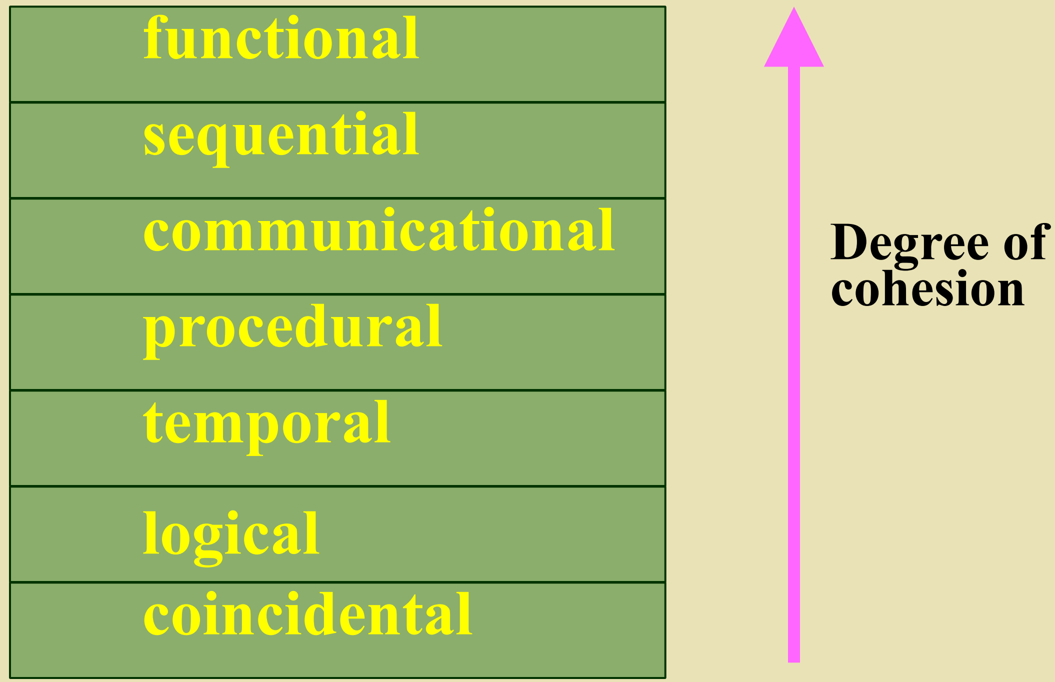
Continue…

- Unfortunately, there are no ways:
  - to quantitatively measure the degree of cohesion and coupling:
  - classification of different kinds of cohesion and coupling will give us some idea (based on research conducted in the area of software engineering) regarding the degree of cohesiveness of a module.

Classification of Cohesiveness

- Classification is often subjective:
  - yet gives us some idea about cohesiveness of a module.
- By examining the type of cohesion exhibited by a module:
  - we can **roughly** tell whether it displays high cohesion or low cohesion.

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.
  - module performs multiple, unrelated actions

  – This amounts to arbitrary modularization

- The result of randomly breaking the project into modules to gain the benefits of having multiple smaller modules to work on-

- Inflexible enforcement of rules such as: "**every function/sub-module shall be between 40 and 80 lines in length**" can result in coincidental cohesion.
- Usually worse than no modularization and may often confuses the reader

Logical cohesion

- All elements of the module perform similar or related actions/operations:
  - e.g. error handling, data input, data output, etc.
- An example of logical cohesion:
  - a set of print functions to generate an output report arranged into a single module.

Temporal cohesion

- The module contains tasks that are related by the fact:
  - all the tasks must be executed in the same time span.
- Example:
  - The set of functions responsible for
    - all system initialization actions
    - start-up, shut-down of some process, etc.

Procedural cohesion

- The set of functions of the module:
  - all are part of a procedure (algorithm)
  - certain sequence of steps have to be carried out in a certain order for achieving an objective, module performs a set of weakly-connected actions corresponding to the sequence of steps in some operation.
  - E.g.-

    - all of the operations involved in an ATM transaction,

    - the algorithm for decoding a message.

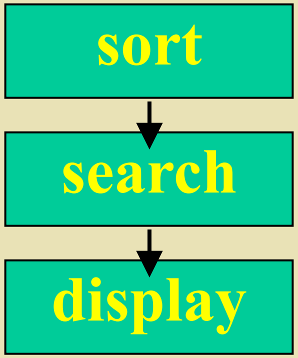
Communicational cohesion

- All functions of the module:
  - reference or update the same data structure, or which operate on the same data.

  Example:
  - the set of functions defined on an array or a stack.
  - <u>Update a database</u>, record update to audit trail, print the update etc.

Sequential cohesion

- Elements of a module form different parts of a sequence,
  - output from one element of the sequence is input to the next.
  - Example:



Functional cohesion

- Different elements of a module cooperate:
  - to achieve a single function.
  - e.g. managing an employee's pay-roll.
    - calculate sales commission.
- When a module displays functional cohesion,
  - we can describe the function using a single sentence.

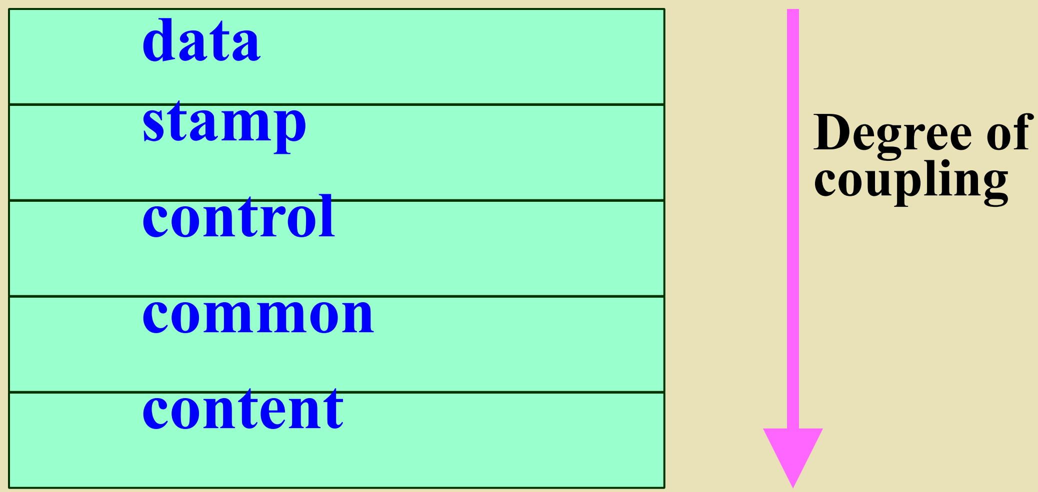
Coupling

- Coupling indicates:
  - how closely two modules interact or how interdependent they are.
  - The degree of coupling between two modules depends on their interface complexity.

Coupling

- There are no ways to precisely determine coupling between two modules:
  - classification of different types of coupling will help us to approximately estimate the degree of coupling between two modules.
- Five types of coupling can exist between any two modules.

Classes of coupling



Data coupling

- Two modules are data coupled,
  - if they communicate via a parameter:
    - an elementary data item is passed as a parameter,
    - e.g an integer, a float, a character etc.
  - The data item should be problem related:
    - not used for control purpose.

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

- 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 <u>common coupled</u>,
  - if they share some global data.

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.

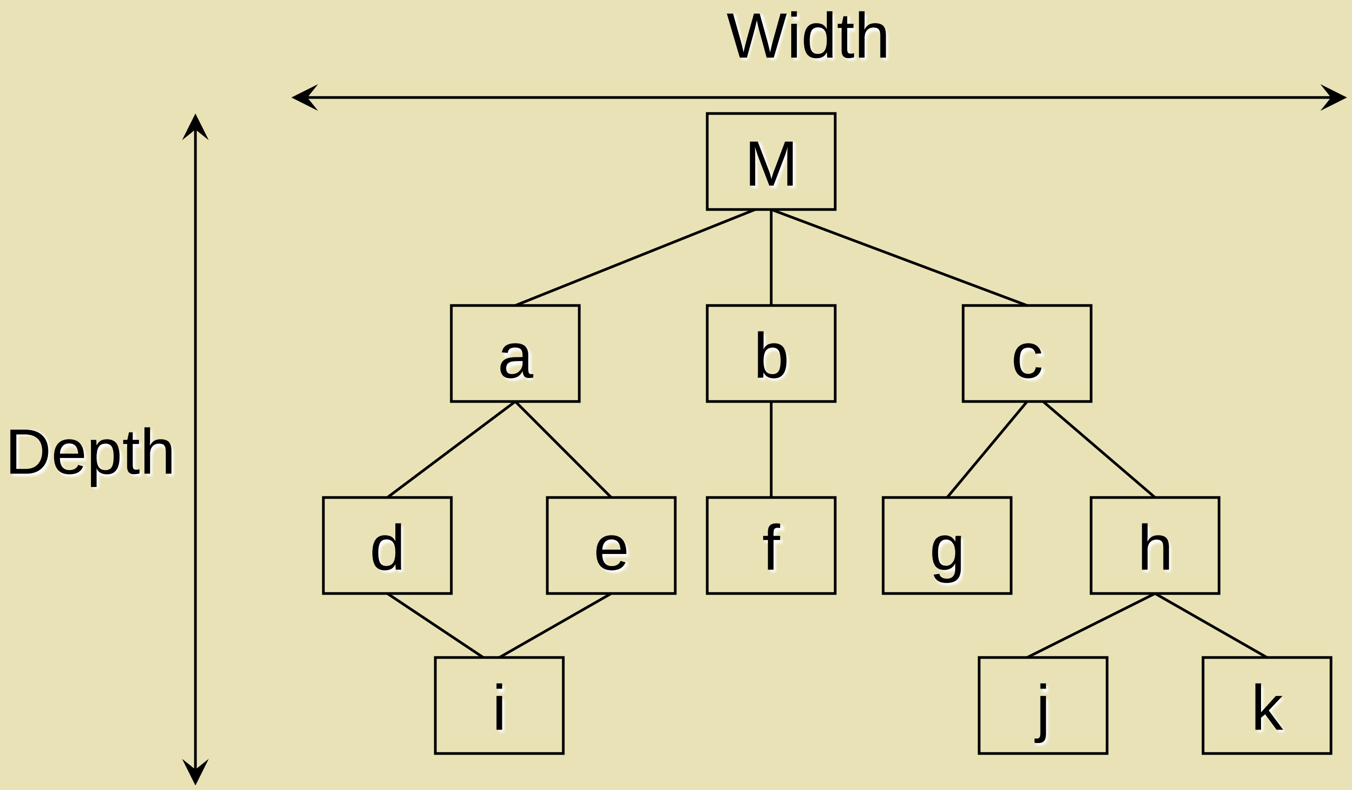
Neat Hierarchy

- Control hierarchy represents:
  - organization of modules.
- Most common notation:
  - a tree-like diagram called structure chart.

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.

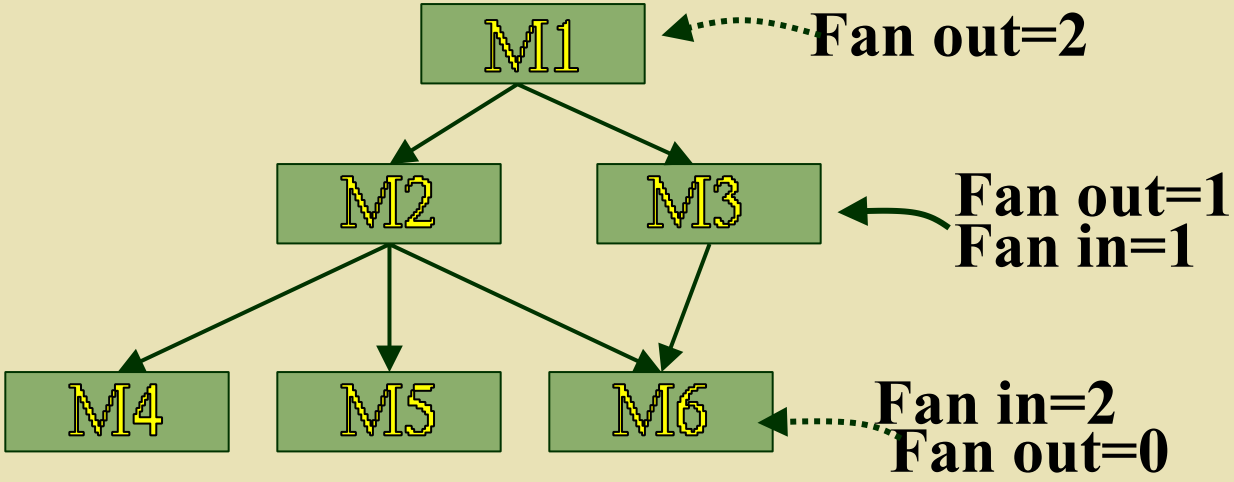
Control Hierarchy



Characteristics of Module Structure

- Fan-in:
  - indicates how many modules directly invoke a given module.

Module Structure



Information Hiding

Information hiding is an adjunct of modularity. It permits modules to be designed and coded without concern for the internals of other modules. Only the access protocols of a module need to be shared with the implementers of other modules. Information hiding simplifies testing and modification by localizing these activities to individual modules.

Benefits of Information Hiding

- A more accurate term would be “details hiding”
- E.g. Supported by public and private specifiers in C++, Java
- Benefits of Information Hiding:
  - Leads to low coupling
  - Reduces the likelihood of “side effects”
  - Limits the global impact of local design decisions
  - Emphasizes communication through controlled interfaces
  - Discourages the use of global data
  - Results in higher quality software

Summary of Design Concepts

- Abstraction: **Enables a problem to be addressed at a high-level of generalization without worrying about low-level details.**
- Refinement: **Process of gradual top-down elaboration of detail.**
- Abstraction and refinement are complementary.
- Modularity: **Software subdivided into separately named and addressable components.**
- Coupling: **The degree to which a module is connected to other modules in the system.**
- Cohesion: **The degree to which a module performs one and only one function.**
- Information Hiding: **Portions of the internal structure of a component are hidden and inaccessible from outside.**