ICSI 403
DESIGN AND ANALYSIS OF ALGORITHMS

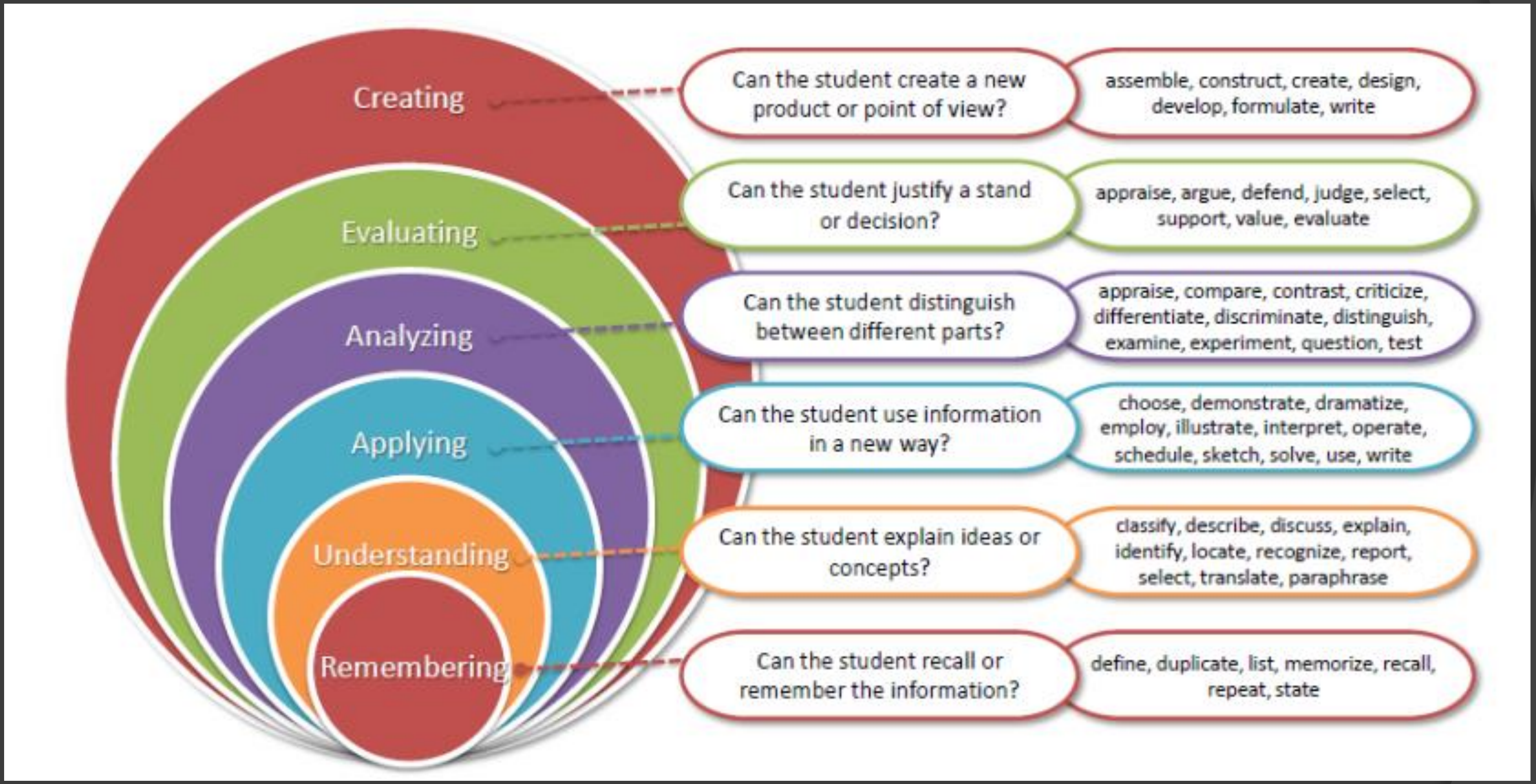
Lecture 01 – Fundamentals

J Marques de Carvalho

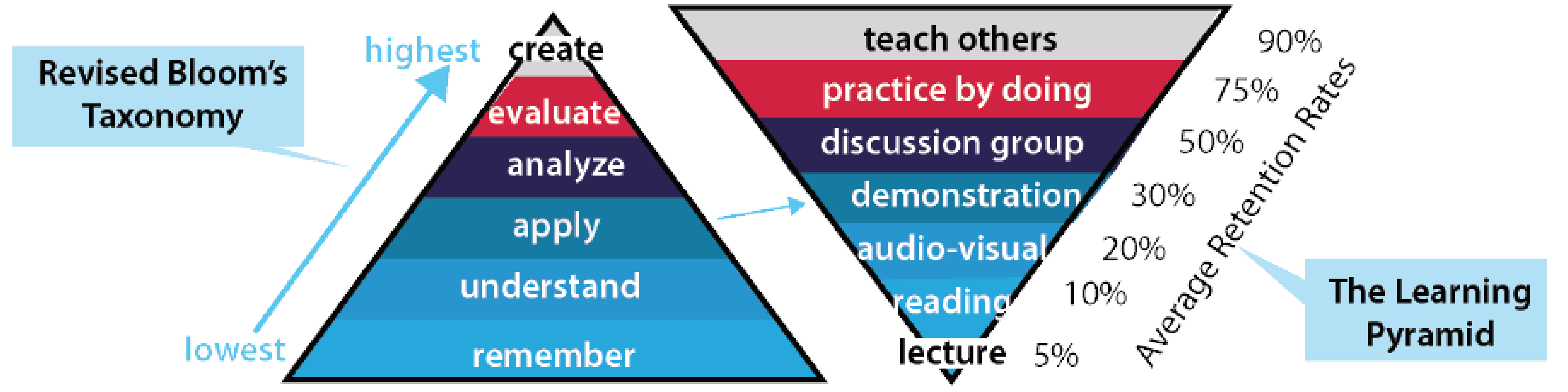
General Information

- Passive vs Active Students.
  - Never leave the class with doubts not cleared.
  - Address questions to the instructor and/or co-instructor.
- ICSI 403 does not teach how to program in C++
  - C++ as a way to make you more competitive in the job market.

Bloom's Taxonomy (Learning and Higher Order Thinking)



Learning and Thinking (cont)



(Adapted from National Training Laboratories Institute, Bethel, Maine.)

DESIGN AND SOFTWARE

What is design?

- Provides structure to any product
- Decomposes system into parts, assigns responsibilities, ensures that parts fit together to achieve a global goal
- Design refers to both an activity and the result of the activity

Two meanings of "design“ activity

- Activity that acts as a bridge between requirements and the implementation of the software
- Activity that gives a structure to the product
  - e.g., a requirements specification document must be *designed*
    - must be given a structure that makes it easy to understand and evolve

The sw design activity

- Defined as system decomposition into modules
- Produces a Software Design Document
  - describes system decomposition into modules
- Often a *software architecture* is produced prior to a software design

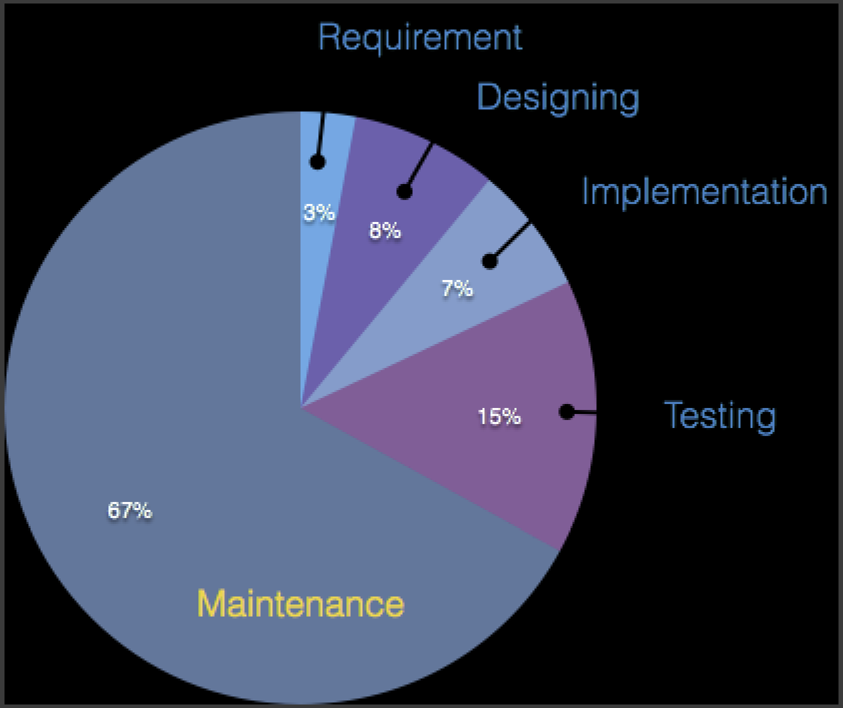
Software architecture

- Shows gross structure and organization of the system to be defined
- Its description includes description of
  - main components of a system
  - relationships among those components
  - rationale for decomposition into its components
  - constraints that must be respected by any design of the components
- Guides the development of the design

Two important goals

- Design for change (Parnas)
  - designers tend to concentrate on current needs
  - special effort needed to anticipate likely changes
- Product families (Parnas)
  - think of the current system under design as a member of a program family

Sample likely changes? (1)



Sample likely changes? (2)

- **Adaptive:** To cope with changes in the software environment (DBMS, OS).
- **Perfective:** It includes new features, new user requirements for refining the software and improve its reliability and performance.
- **Corrective:** Diagnosing and fixing errors.
- **Preventive:** It aims to attend problems, which are not significant at this moment but may cause serious issues in future.
- 75% of maintenance effort is on the first two. 21% is on corrective

Sample likely changes? (3)

- Perfective, adaptive maintenance
- **Algorithms**
  - e.g., replace inefficient sorting algorithm with a more efficient one
- Change of data representation
  - e.g., from binary tree to a threaded tree
  - ≈17% of maintenance costs attributed to data representation changes (Lientz and Swanson, 1980)

Sample likely changes? (4)

- Change of underlying abstract machine
  - new release of operating system
  - new optimizing compiler
  - new version of DBMS
  - …
- Change of peripheral devices
- Change of "social" environment
  - new tax regime
  - EURO vs national currency in EU
- Change due to development process (transform prototype into product)

Module

- A well-defined component of a software system
- A part of a system that provides a set of services to other modules
  - Services are computational elements that other modules may use

Questions

- How to define the structure of a modular system?
- What are the desirable properties of that structure?

Modules and relations

- Let S be a set of modules

  $$S = \{M_1, M_2, \ldots, M_n\}$$

- A binary relation r on S is a subset of

  $$S \times S$$

- If M_i and M_j are in S, $<M_i, M_j> \in r$ can be written as $M_i \; r \; M_j$

Relations

- Transitive closure r^+ of r

  $M_i\ r^+\ M_j$ <u>iff</u>

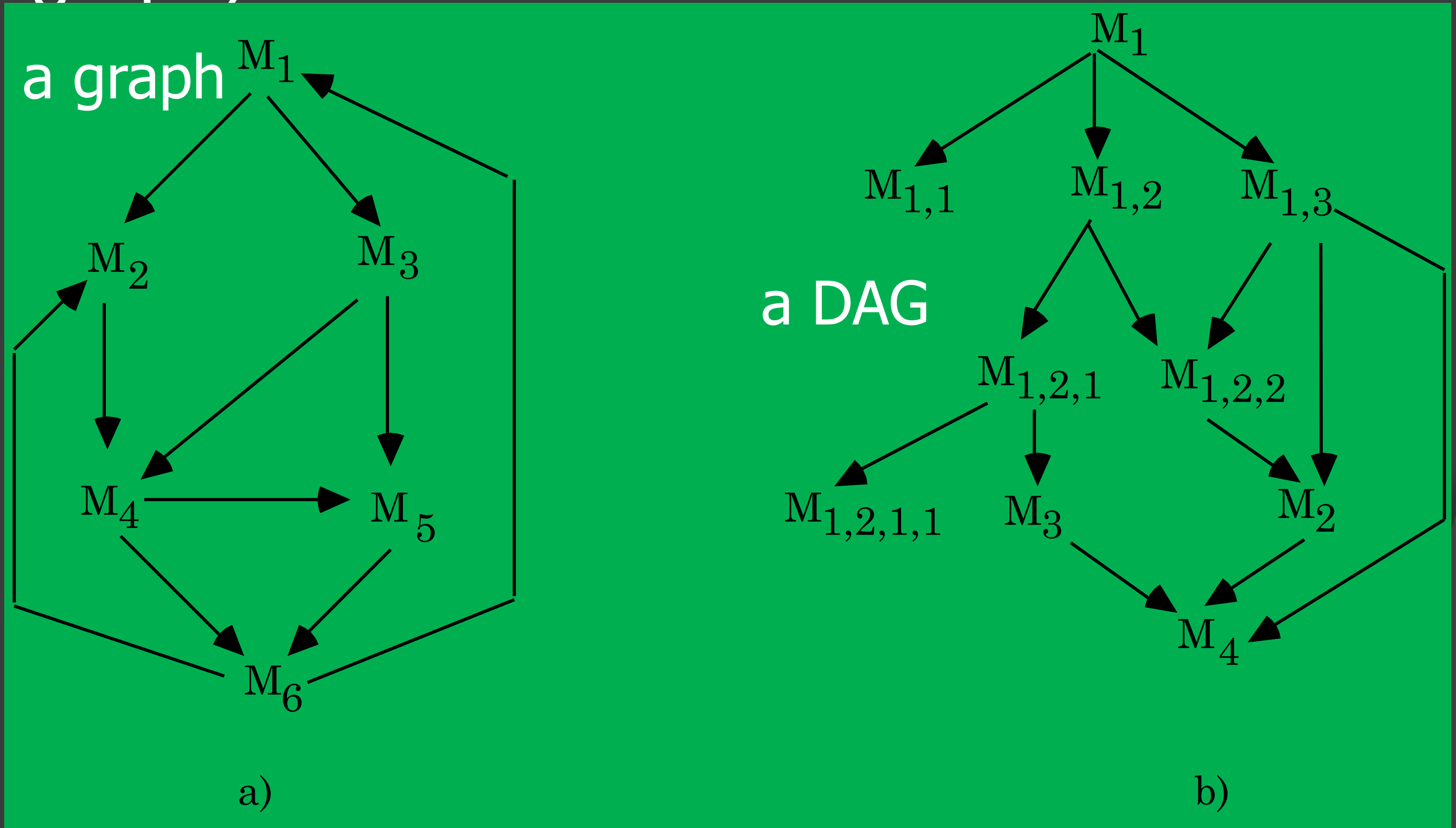
  $M_i\ r\ M_j$ or $\exists\ M_k$ in S s.t. $M_i\ r\ M_k$ and $M_k\ r^+\ M_j$

(We assume our relations to be irreflexive)

- r is a hierarchy iff there are no two elements M_i, M_j s.t. $M_i\ r^+\ M_j \wedge M_j\ r^+\ M_i$

Relations

- Relations can be represented as graphs
- A hierarchy is a DAG (directed acyclic graph)



The USES relation

- A uses B
  - A requires the correct operation of B
  - A can access the services exported by B through its interface
  - it is "statically" defined
  - A depends on B to provide its services
    - example: A calls a routine exported by B
- A is a client of B; B is a server

Desirable property

- USES should be a hierarchy
- Hierarchy makes software easier to understand
  - we can proceed from leaf nodes (who do not use others) upwards
- They make software easier to build
- They make software easier to test

Hierarchy

- Organizes the modular structure through *levels of abstraction*
- Each level defines an *abstract (virtual) machine* for the next level
  - *level* can be defined precisely
    - M_i has level 0 if no M_j exists s.t. M_i r M_j
    - For each module M_i, let k be the maximum level of all nodes M_j s.t. M_i r M_j. Then M_i has level k+1

Hierarchy: USES example

- Let M_R be a module that provides input-output of record values.
- Let M_R use another module M_B that provides I/O of a single byte at a time.
- When used to output record values, the job of M_R consists of transforming the record into a sequence of bytes and isolating a single byte at a time to be output by means of M_B.
- M_B provides a service that is used by M_R.

Module Level Concepts

- If a module M_i is composed of a set of other modules $M_{S,i}$ then the modules of set $M_{S,i}$ actually provide all of the services that M_i should provide.
- In design, once M_i is decomposed in the set $M_{S,i}$, it is replaced by them. M_i is an abstraction that is implemented in terms of simpler abstractions.

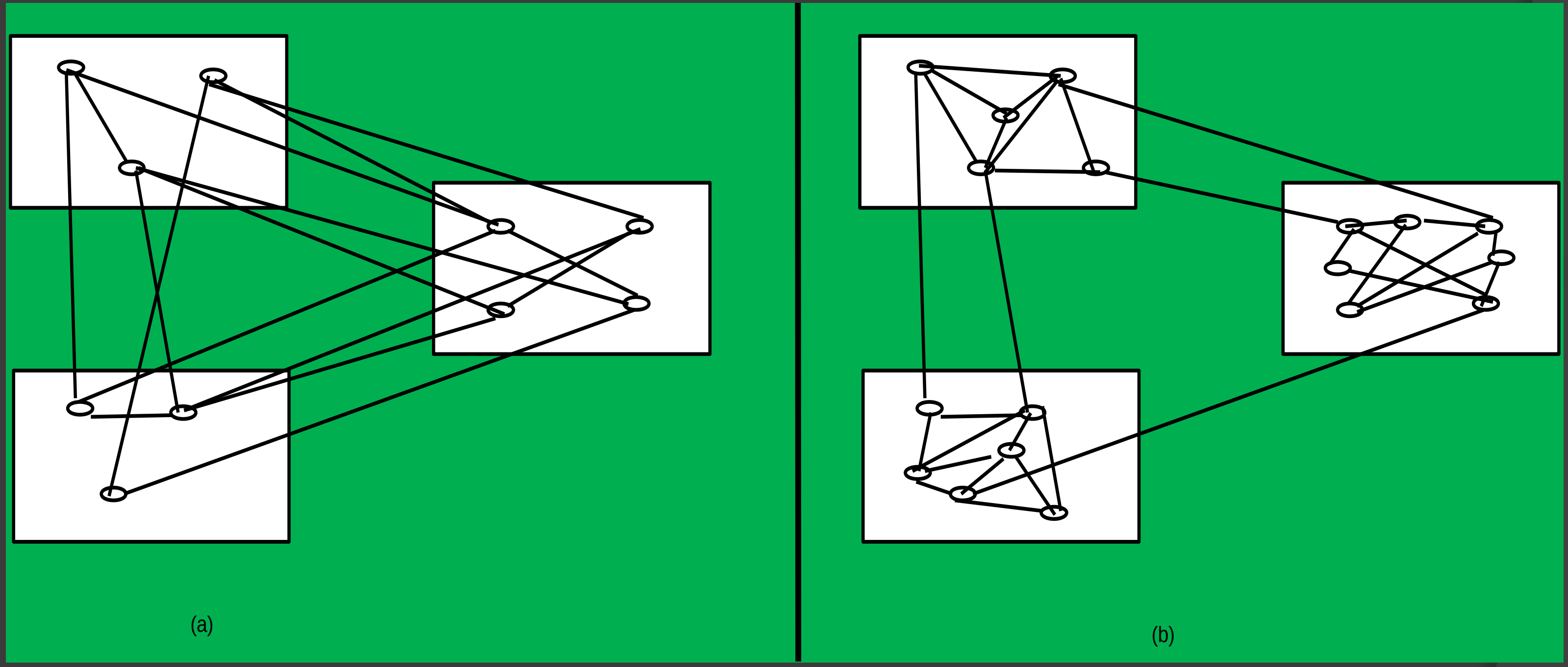
Module Level Concepts

- Ideally, we decompose up to have a minimum of interaction between modules and, conversely, a high degree of interaction within a module.
- Coupling: measure of independence
- Cohesion: logical relationship
- Cohesion and coupling help determine “quality” of the architecture.

Cohesion and coupling

- Each module should be *highly cohesive*
  - module understandable as a meaningful unit
  - Components of a module are closely related
- Modules should exhibit *low coupling*
  - modules have low interactions with others
  - understandable separately

A visual representation



high coupling

low coupling

Module Level Concepts

- The USES relation provides a way to reason about the coupling in a precise manner.
- With reference to a USES graph, we can distinguish the number of incoming edges (fan-in) and the number of outgoing edges (fan-out).

Module Level Concepts(cont)

- A good design structure should keep the fan-out low and the fan-in high.

Module Level Concepts

- A high fan-in is an indication of good design because a module with high fan-in represents a meaningful i.e. general abstraction that is used heavily by other modules.
- A high fan-out is an indication that a module is doing too much which in turn may imply that a module has poor cohesion.
- The evaluation of the quality of design should not merely depend on the USES relation.