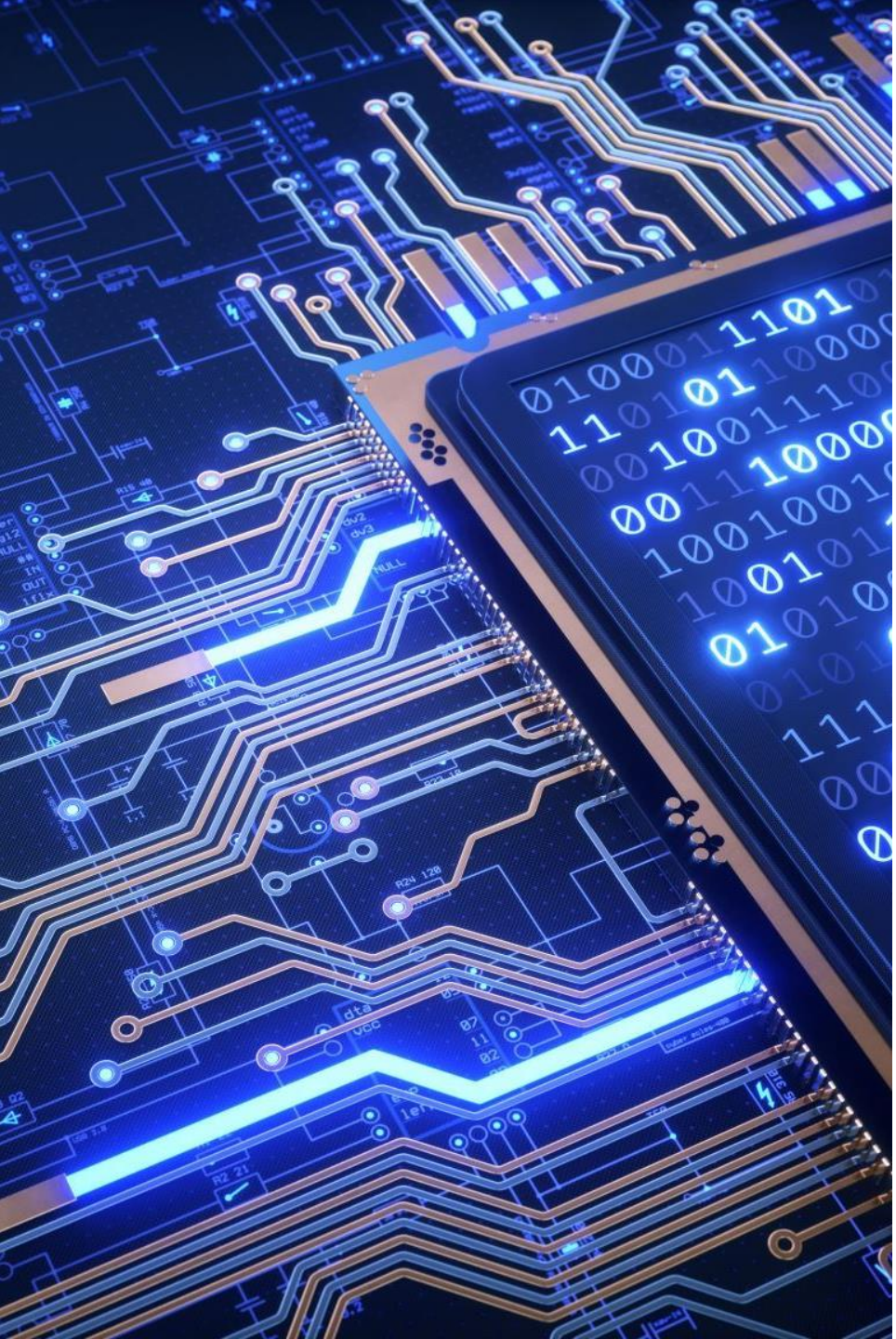


CSci363 User Interface Design

Friday, September 6, 2024

- **Today's session**:

- Software Process

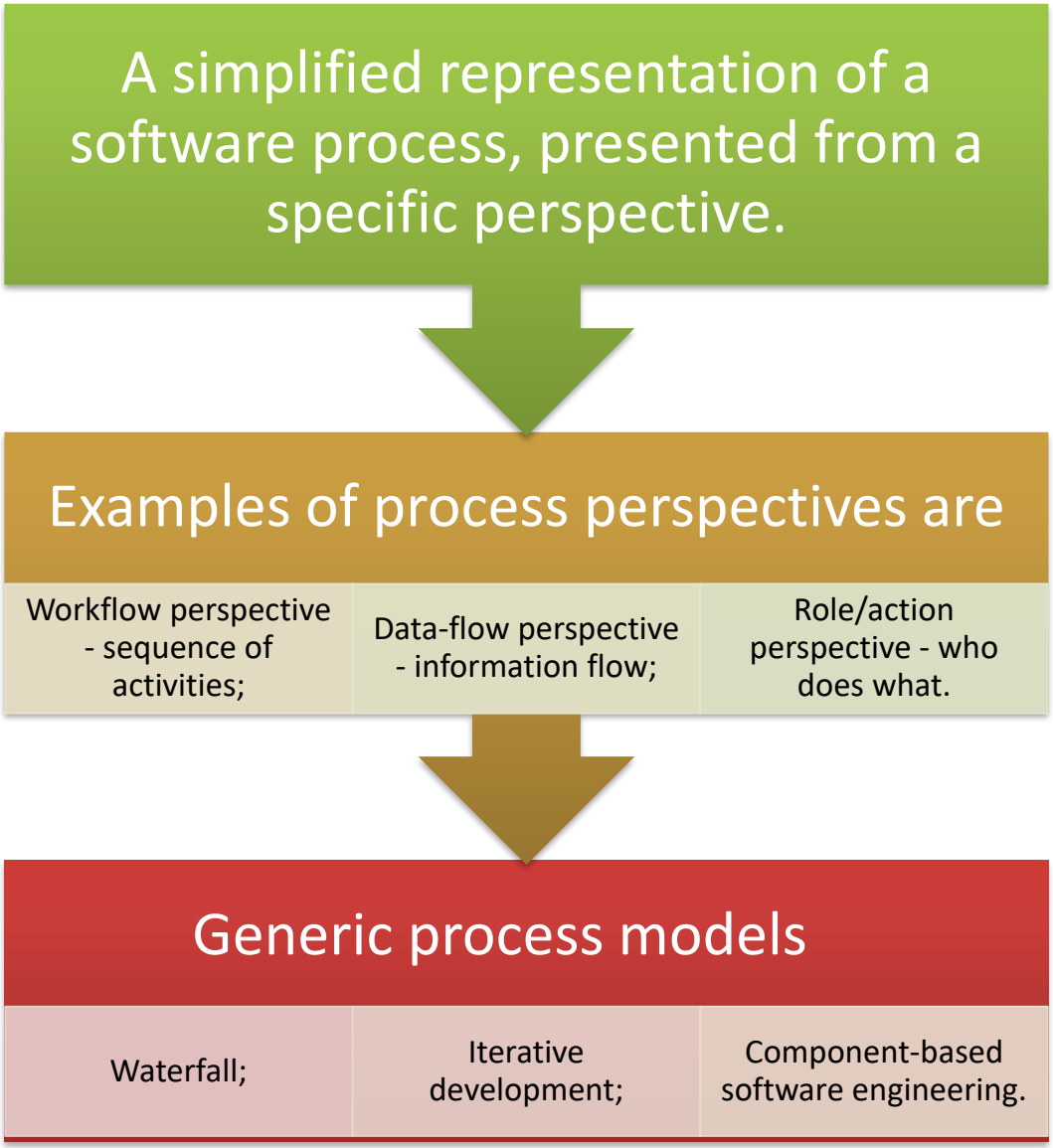
What is a software process?

A set of activities whose goal is the development or evolution of software.

Generic activities in all software processes are:

Specification - what the system should do and its development constraints	Development - production of the software system	Validation - checking that the software is what the customer wants	Evolution - changing the software in response to changing demands.

What is a software process model?



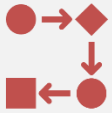
What are the costs of software engineering?

Roughly 60% of costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.

Costs vary depending on the type of system being developed and the requirements of system attributes such as performance and system reliability.

Distribution of costs depends on the development model that is used.

Process iteration



System requirements ALWAYS evolve in the course of a project so process iteration where earlier stages are reworked is always part of the process for large systems.

Iteration can be applied to any of the generic process models.

Two (related) approaches

Incremental delivery;

Spiral development.

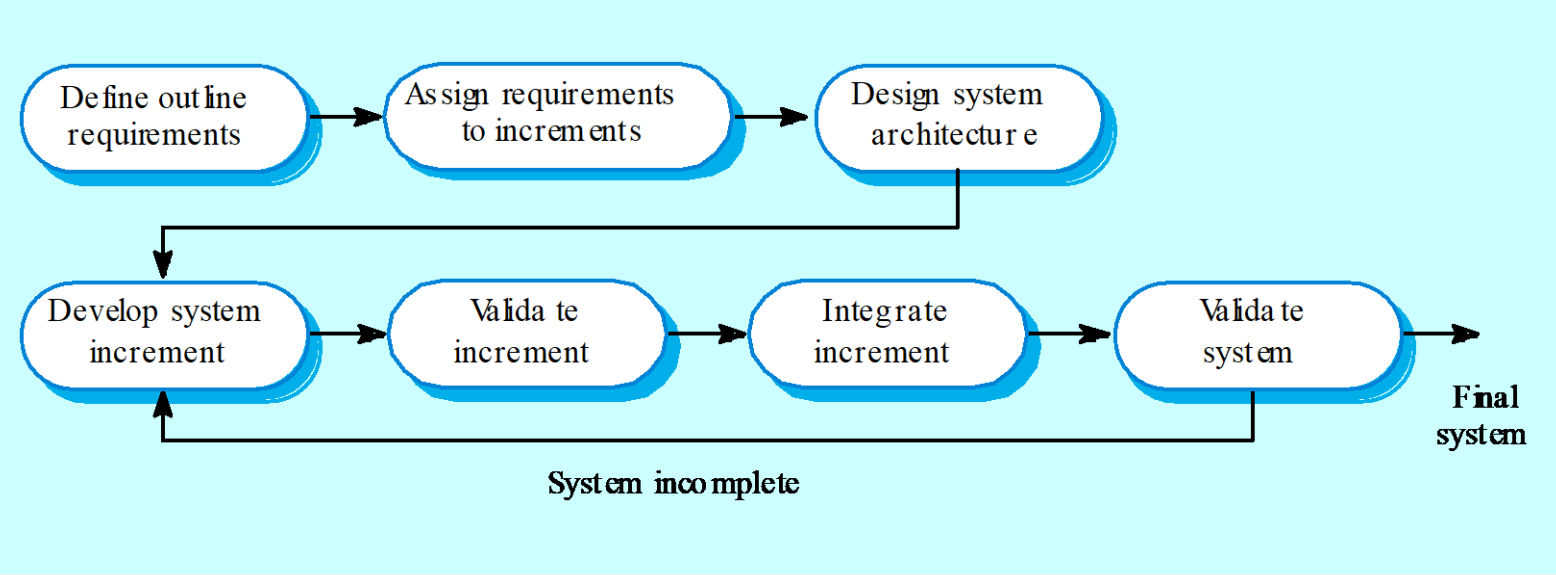
Incremental Delivery

Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.

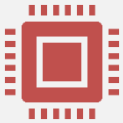
User requirements are prioritised and the highest priority requirements are included in early increments.

Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.

Incremental Development



Incremental Development Advantages



Customer value can be delivered with each increment, so system functionality is available earlier.

Early increments act as a prototype to help elicit requirements for later increments.

Lower risk of overall project failure.

The highest priority system services tend to receive the most testing.

Software Engineering Principles

Outline

Principles form the basis of methods, techniques, methodologies and tools

Seven important principles that may be used in all phases of software development

Modularity is the cornerstone principle supporting software design

Application of Principles

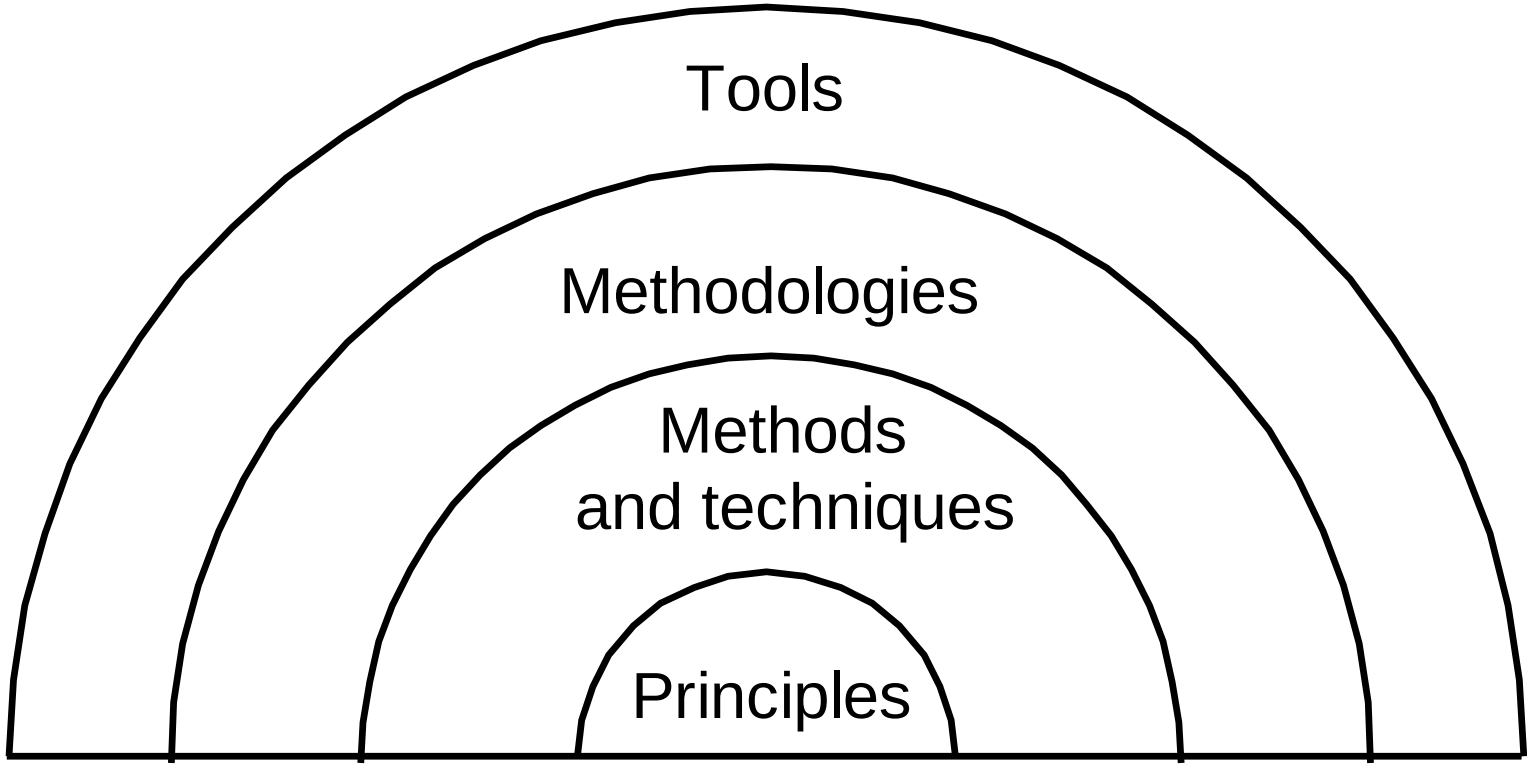
Principles apply to process and product

Principles become practice through methods and techniques

often methods and techniques are packaged in a *methodology*

methodologies can be enforced by *tools*

A Visual Representation



Key Principles

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality
- Reuse (my addition)

Rigor and formality

Software engineering is a creative design activity, BUT

It must be practiced systematically

Rigor is a necessary complement to creativity that increases our confidence in our developments

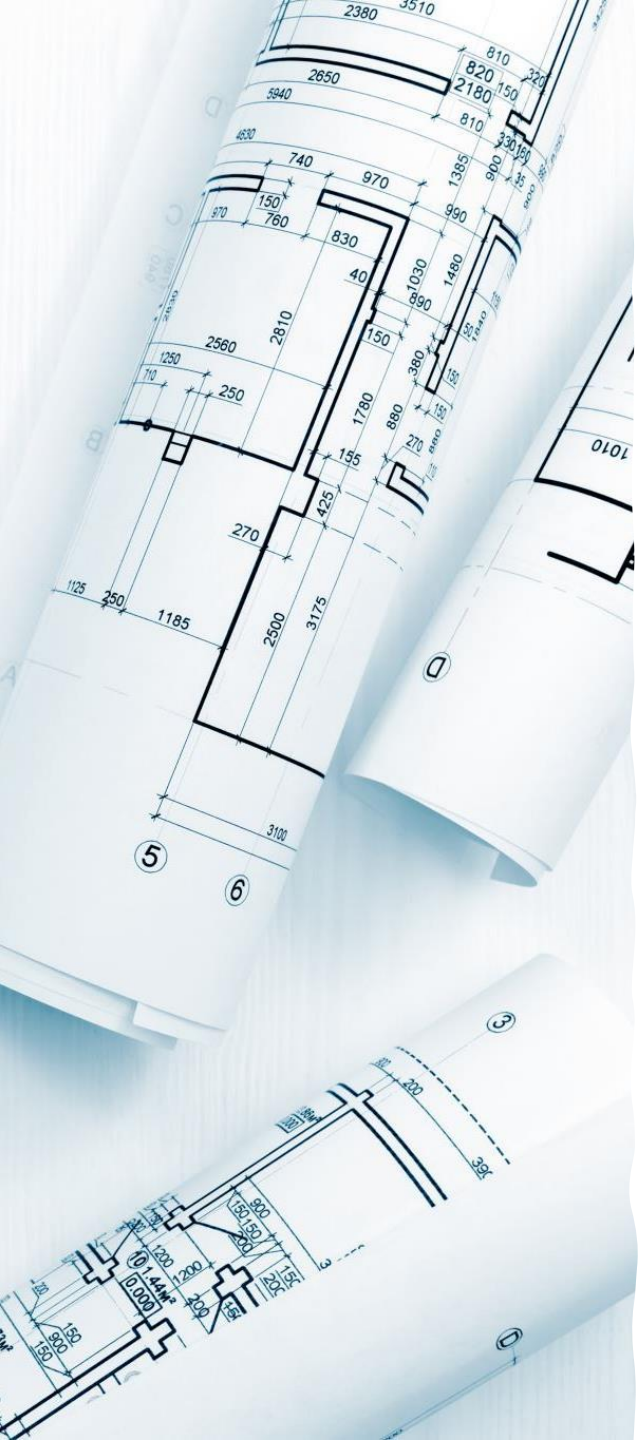
Formality is rigor at the highest degree

software process driven and evaluated by mathematical laws

Examples: Product

Mathematical (formal) analysis of program correctness

Systematic (rigorous) test data derivation



Example: Process

Rigorous documentation of development steps helps project management and assessment of timeliness

Separation of concerns

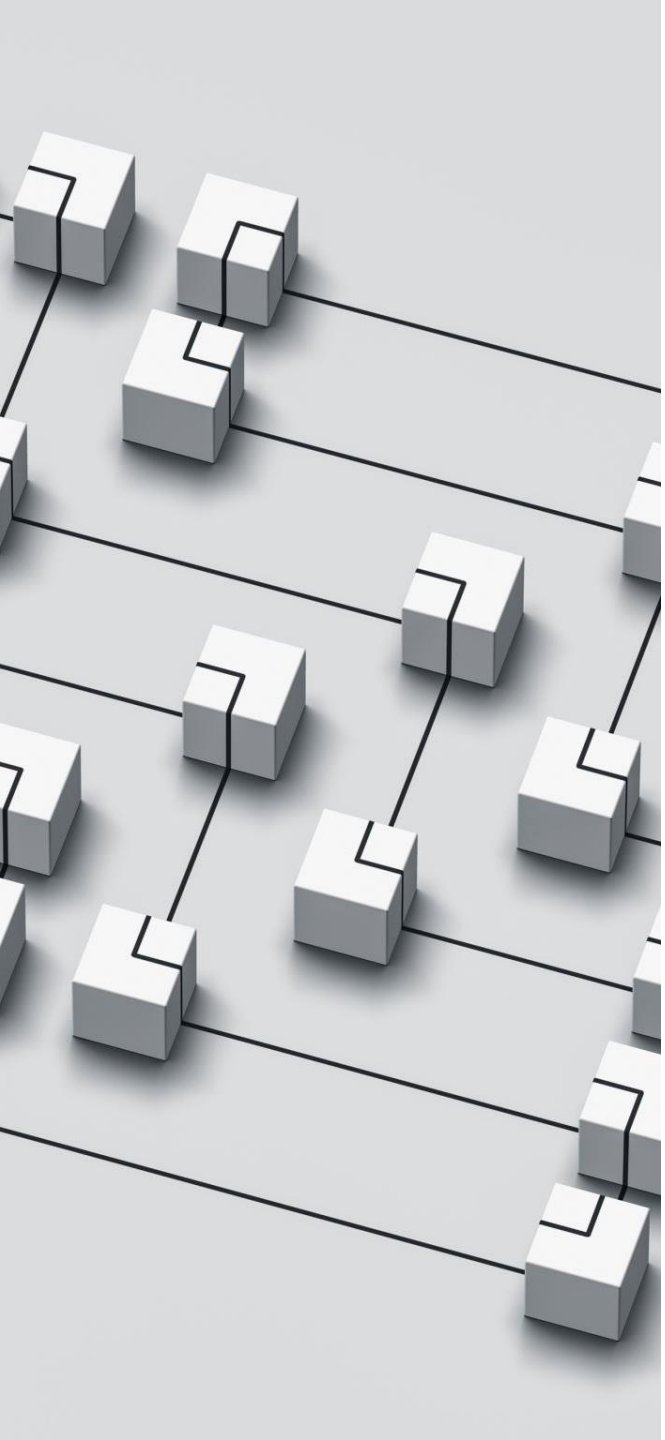
- To dominate complexity, separate the issues to concentrate on one at a time
- "Divide & conquer" (*divide et impera*)
- Supports parallelization of efforts and separation of responsibilities

Example: Process

- Go through phases one after the other (as in waterfall)
  - Does separation of concerns by separating activities with respect to time

Example: Product

- Keep product requirements separate
  - functionality
  - performance
  - user interface and usability



Modularity

- A complex system may be divided into simpler pieces called *modules*
- A system that is composed of modules is called *modular*
- Supports application of separation of concerns
  - when dealing with a module we can ignore details of other modules

Cohesion and Coupling

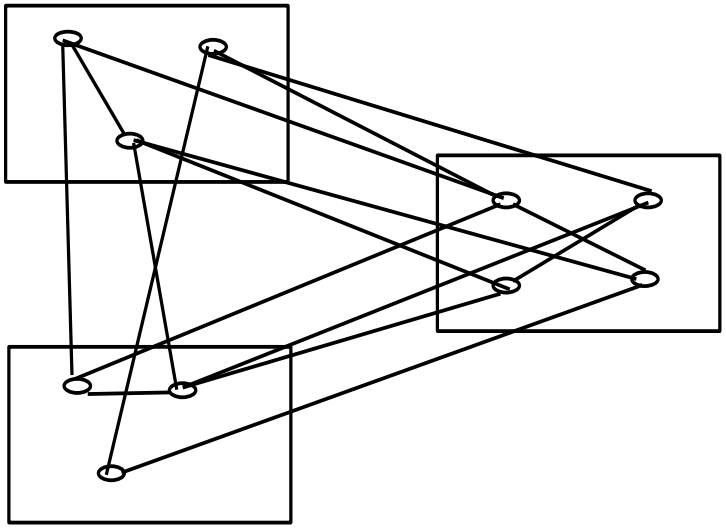
Each module should be *highly cohesive*

- module understandable as a meaningful unit
- Components of a module are closely related to one another

Modules should exhibit *low coupling*

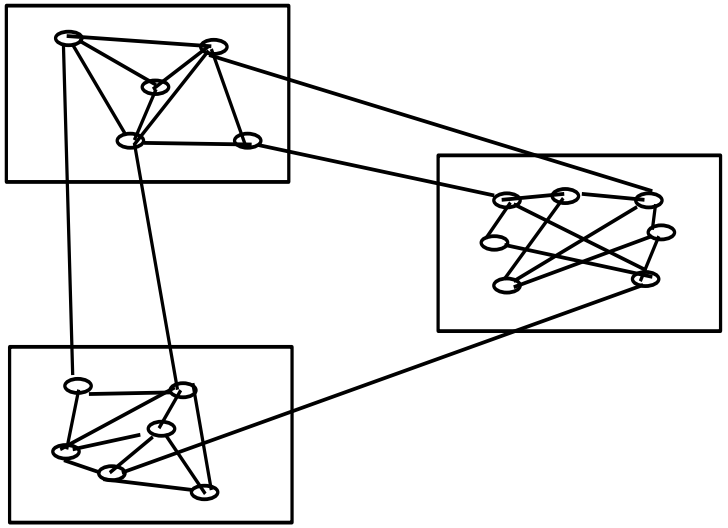
- modules have low interactions with others
- understandable separately

A Visual Representation



(a)

high coupling

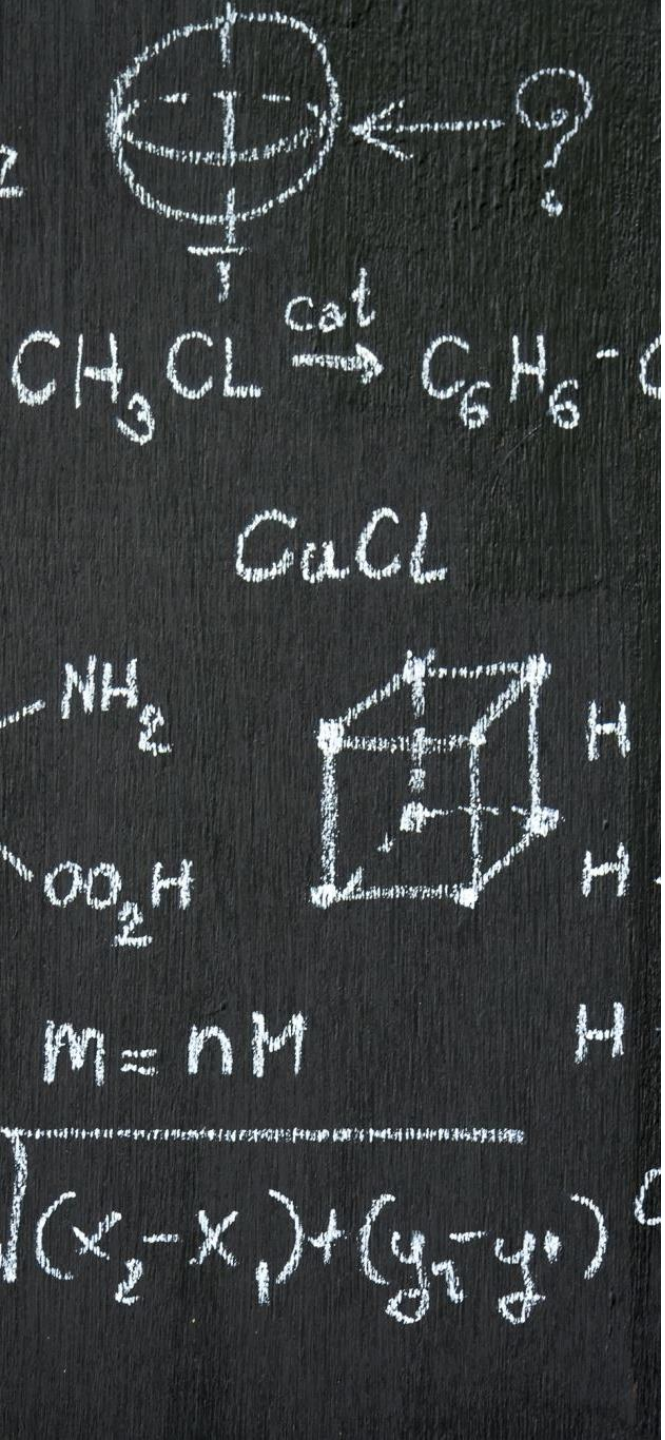


(b)

low coupling

Abstraction

- Identify the important aspects of a phenomenon and ignore its details
- Special case of separation of concerns
- The type of abstraction to apply depends on purpose
- Example : the user interface of a watch (its buttons) abstracts from the watch's internals for the purpose of setting time; other abstractions needed to support repair



Abstraction Ignores Details

- Example: equations describing complex circuit (e.g., amplifier) allows designer to reason about signal amplification
- Equations may approximate description, ignoring details that yield negligible effects (e.g., connectors assumed to be ideal)

Abstraction Yields Models

For example, when requirements are analyzed we produce a model of the proposed application

The model can be a formal or semiformal description

It is then possible to reason about the system by reasoning about the model