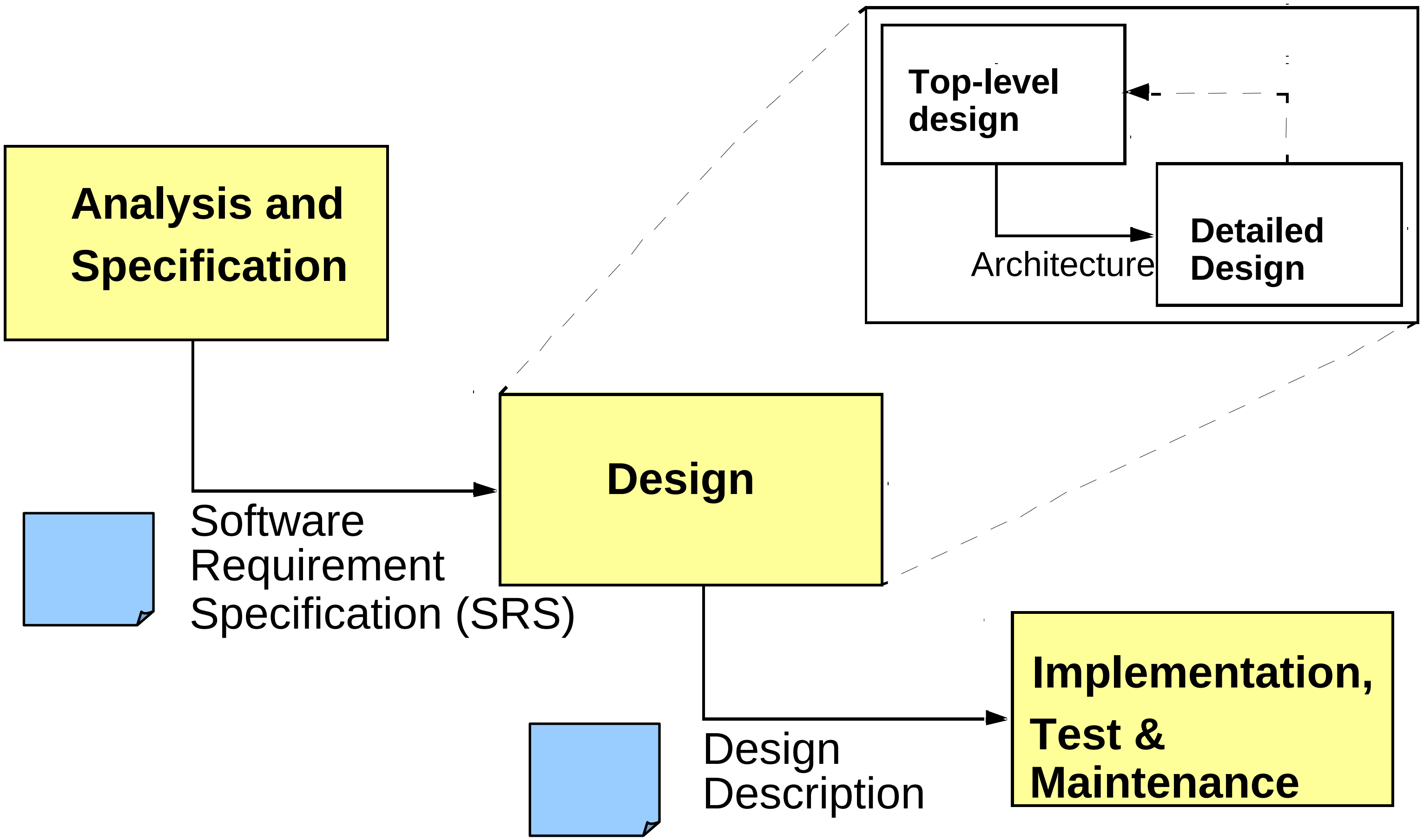
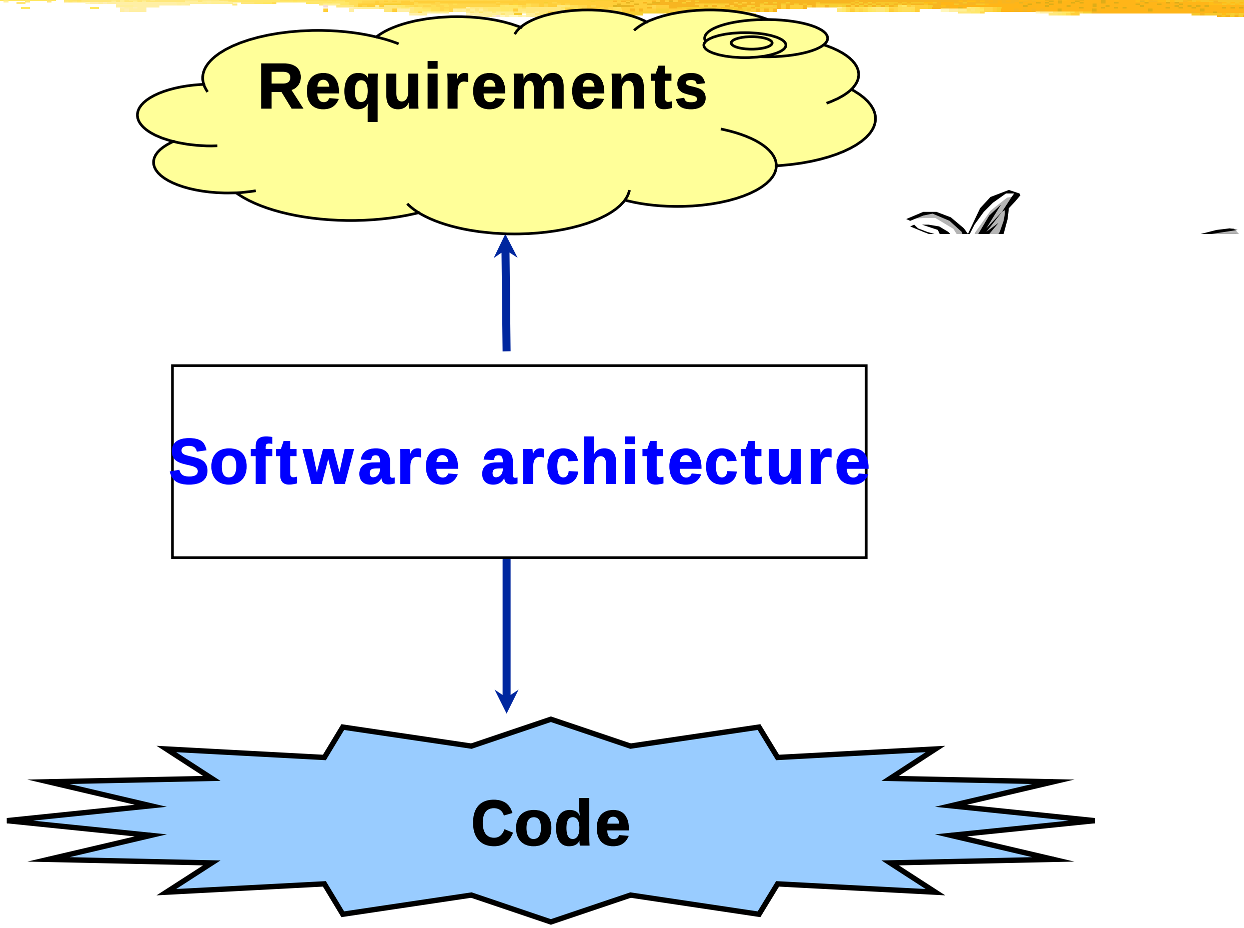
# Software Engineering

## Modularity

# Design Within the Life-Cycle



# How to Create an Architecture?



# Software Design: Definitions

- Source: IEEE Standard 610.12-1990
- Design
  - The process of defining the architecture, components, interfaces, and other characteristics of a system or component
- Architectural design
  - The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system

# Software Design: Definitions

- Preliminary design
  - The process of analyzing design alternatives and defining the architecture, components, interfaces, and timing and sizing estimates for a system or component
- Design requirement
  - A requirement that specifies or constrains the design of a system or system component

# Design Activities

- Top-level design
  - Dividing the system hierarchically into sub-systems
  - Identifying components
  - Assigning components to subsystems
  - Identifying relations between components
  - Designing the interaction of the components
  - Result:
    - Description of the architecture

# Design Activities

- Detailed design
  - Specification of the interfaces of all components
  - Specification of the uses and call relations between the components
  - Result:
    - Component designs

# Architecture: Definition

- The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements**,** the externally visible properties of those elements, and the relationships among them

Bass, L., Clements, P., Kazmann, R. (2003): Software Architecture in Practice, 2nd Edition, Addison-Wesley

# Modularity

# Modularity Goals

*Required properties resulting from a "modular" design method*

- The following criteria can help evaluate design methods with respect to modularity
  - Decomposability
  - Composability
  - Understandability
  - Continuity
  - Protection

# Modularity and Quality

- Correctness
  - modular understandability
  - modular decomposability
  - modular protection
- Robustness
  - modular protection

# Modularity and Quality

- Extendibility
  - modular continuity
  - modular composability
- Reusability
  - modular composability

# Modularity and Quality

- Compatibility
  - modular composability
  - modular continuity
- Efficiency
  - modular composability

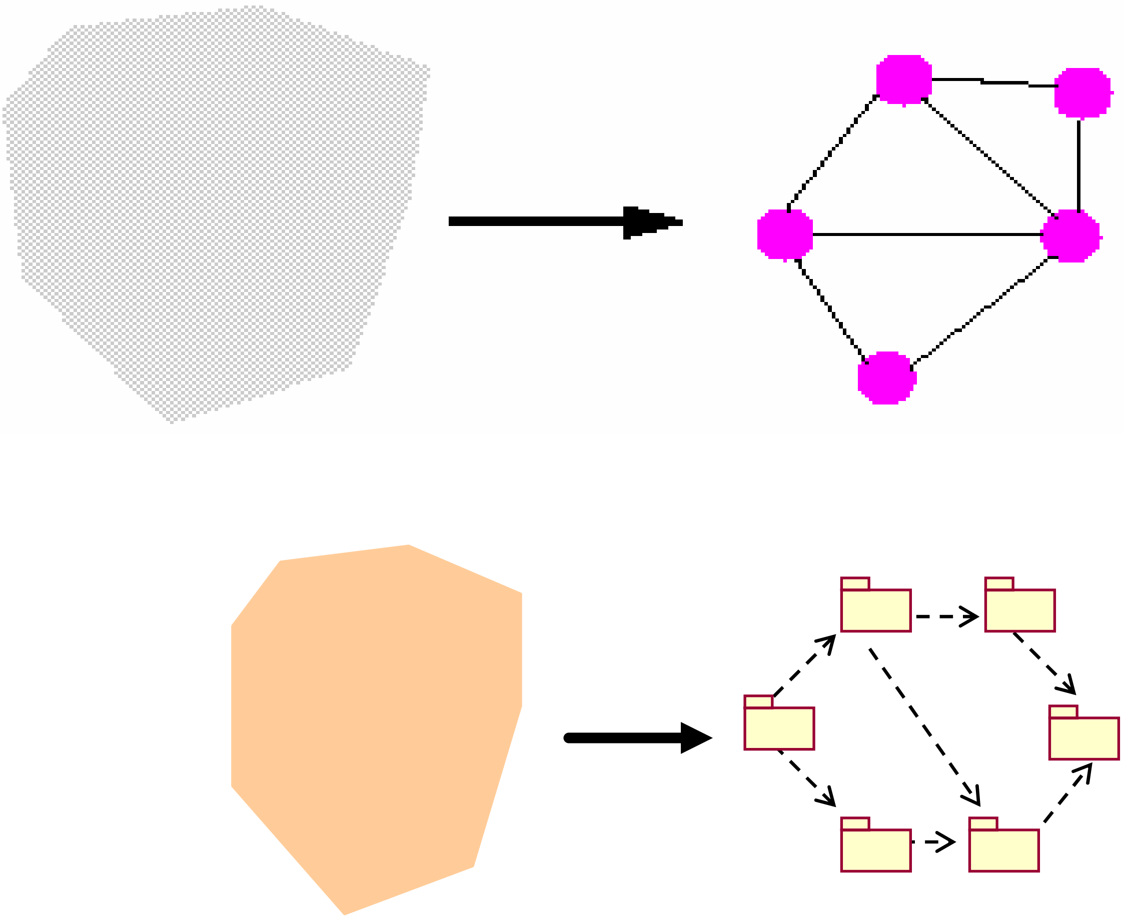
# Modularity and Quality

- Portability
  - modular composability
- Timeliness
  - modular composability
  - modular continuity
  - modular protection

# Modular Decomposability

- When a method supports decomposing a software problem into a small number of less complex sub-problems independent enough to allow further work to proceed separately on each of them
  - In general method will be repetitive
    - Sub-problems will be divided still further
  - Top-down design methods fulfil this criterion
    - Stepwise refinement is an example of such method
    - Counter-example: include in each software system a global initialization module (temporal cohesion)

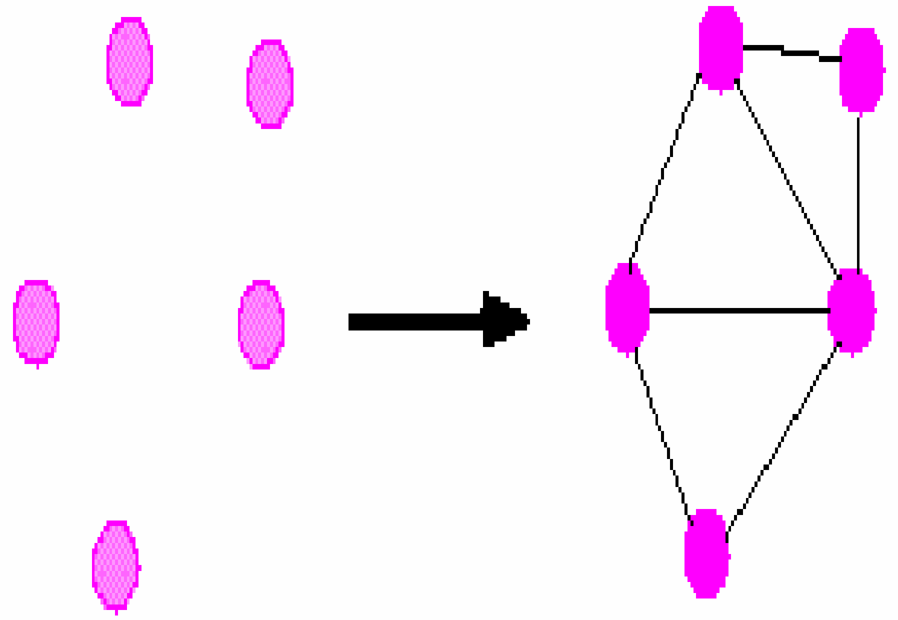
# Modular Decomposability



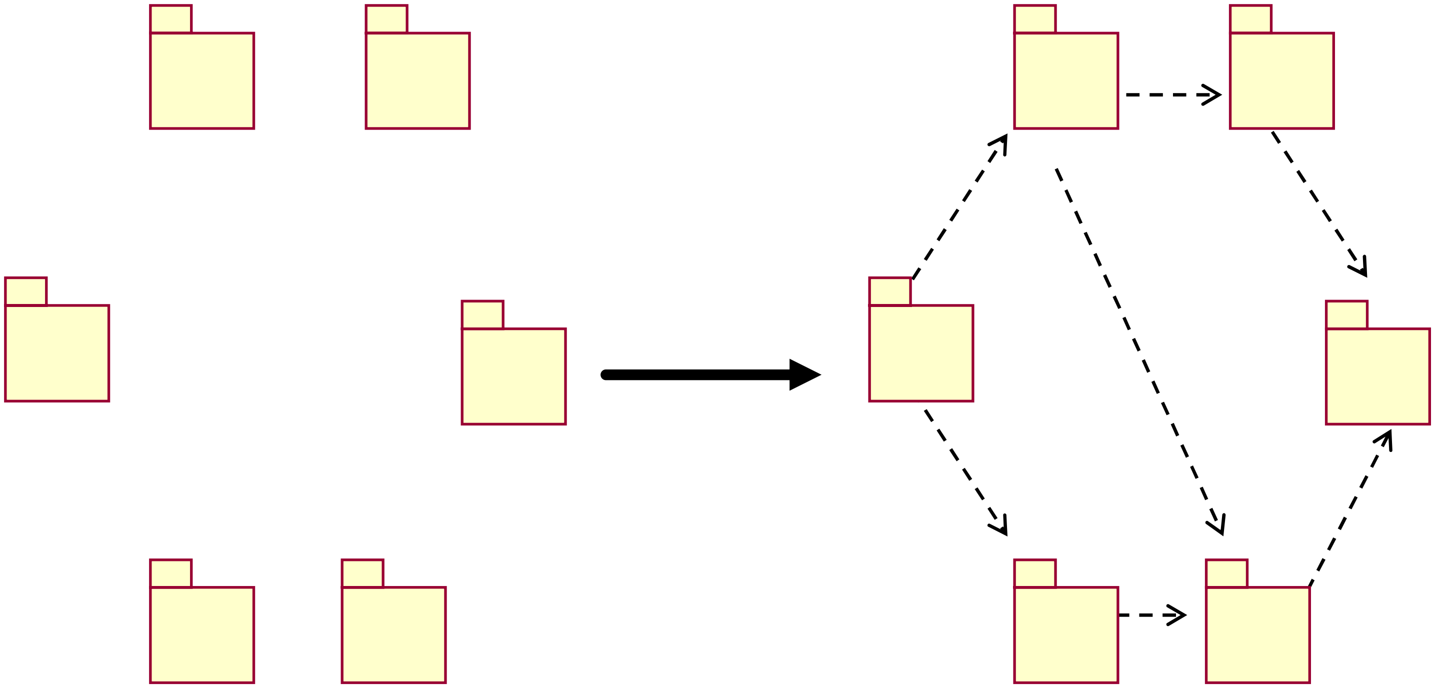
# Modular Composability

- When a method favors the production of software elements which may then be freely combined with each other to produce new system
  - Composability is directly related to the issue of reusability
    - Example 1: subprogram libraries
    - Example 2: Unix shell conventions
      - A basic Unix command operates on an input viewed as a sequential character stream
        - The Unix shell provides the pipe facility
    - Counter-example: preprocessors

# Modular Composability



# Modular Composability

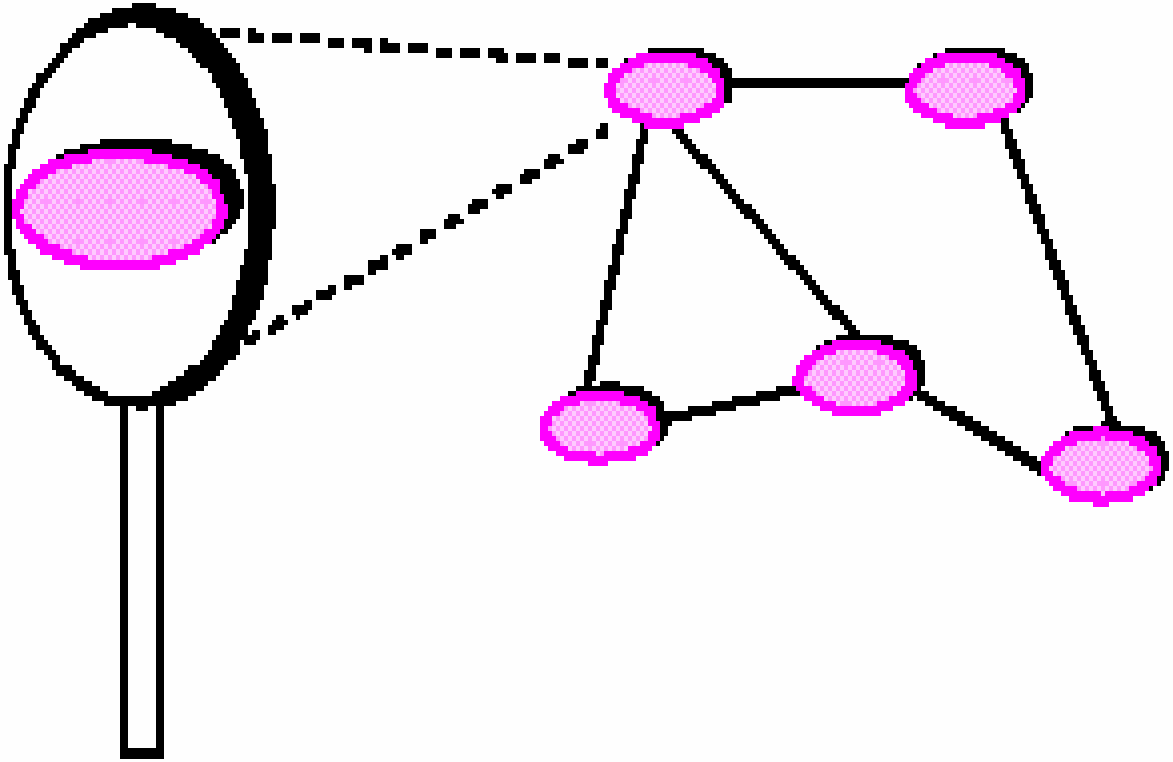


# Modular Understandability

- When a method helps produce software in which a human reader can understand each module without having to know the others
  - Counter-example 1: sequential dependencies

    modules which have to be activated in a certain prescribed order: A | B | C
  - Counter-example 2: a thousand lines program, containing no procedures

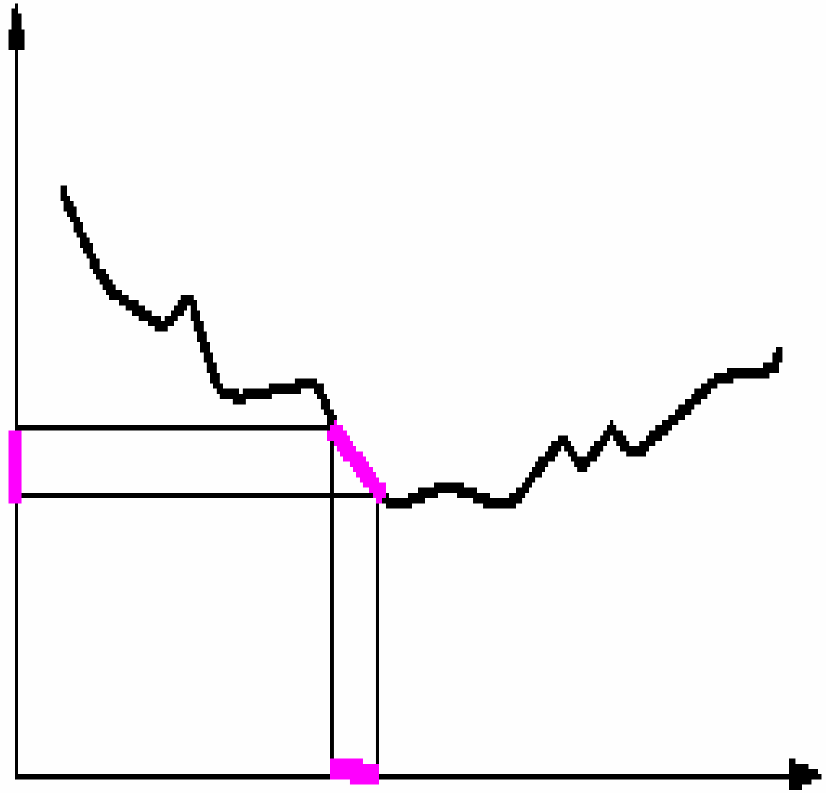
# Modular Understandability



# Modular Continuity

- When a method satisfies that small change in a problem specification will trigger a change of just one or small number of modules
  - Example 1: symbolic constants
  - Example 2: the Uniform Access principle
  - Counter-example 1: using physical representations
  - Counter-example 2: static arrays

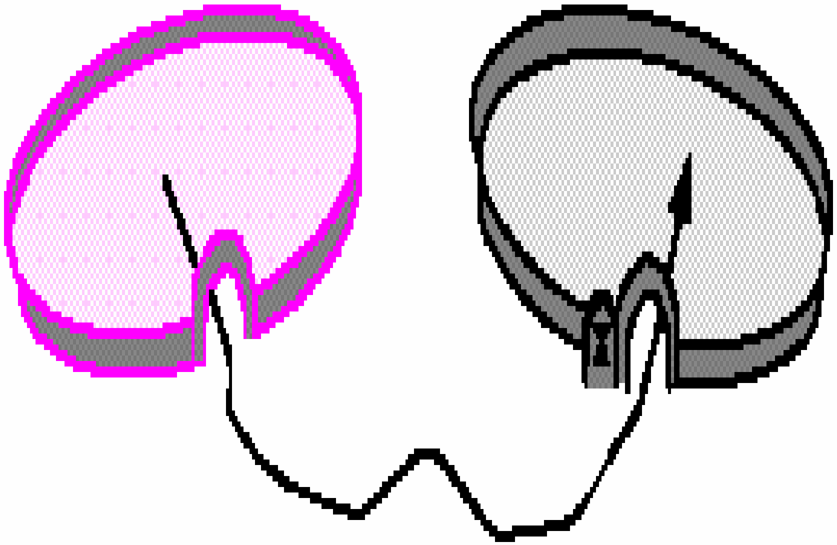
# Modular Continuity



# Modular Protection

- When a method confines the effect of an abnormal condition occurring in a module to that module, or at worst only a few neighboring modules
  - Example: validating input at the source
  - Counter-example: undisciplined exceptions

# Modular Protection



# Modularity

*Language has **Enabling Factors***

*Help to achieve*
***Modularity Goals***

*Designer applies **Guidelines***

# Enabling Factors

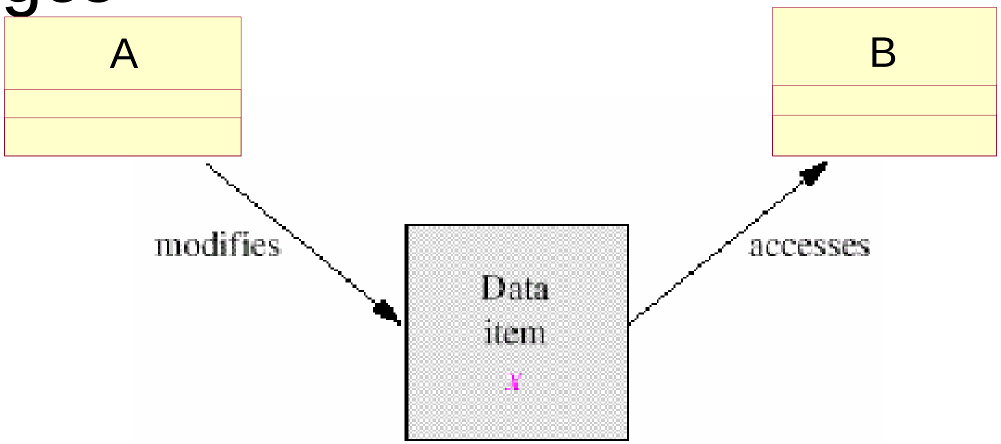
- Principles that must be followed to ensure proper modularity
  - Linguistic modular units
  - Explicit interfaces
  - Information hiding
  - Few interfaces (coupling)
  - Small interfaces (coupling)

# Linguistic (Syntactic) Modular Units

- Modules must correspond to syntactic units in the language used
  - The language may be a programming language, a program design language, a specification language, etc.
  - This principle follows from several modularity criteria:
    - Decomposability: if a system is divided into separate tasks, then each one must result in a clearly delimited syntactic unit which is separately compilable
    - Composability: only closed units can be combined
    - Protection: the scope of errors can only be controlled if modules are syntactically delimited

# Explicit Interfaces

- Whenever two modules communicate this must be obvious from their text
  - If we change a module, we need to see what other modules may be affected by these changes

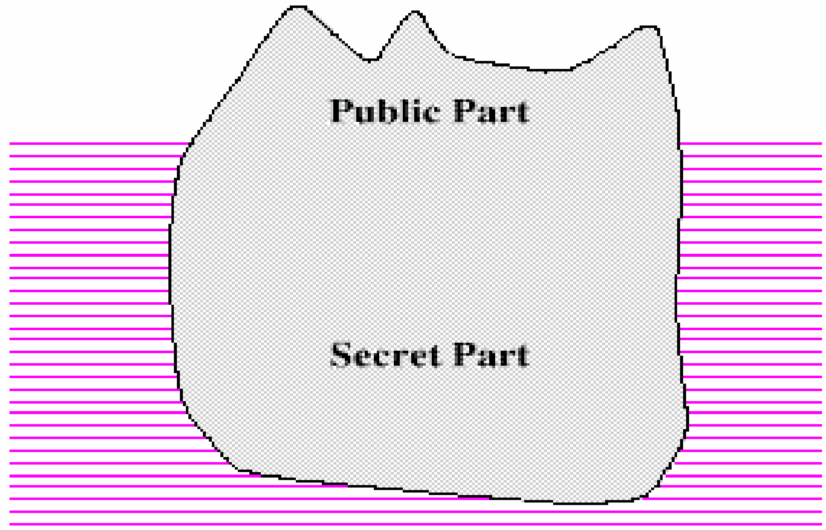


# Explicit Interfaces

- Criteria:
  - Decomposability and composability: if a module is to be decomposed into or composed with others, any outside connection should be clearly marked
  - Continuity: what other element might be impacted by a change should be obvious.
  - Understandability: difficult to understand A if its behavior is influenced by B in some tricky way

# Information Hiding

- Every module can make a subset of its properties available to clients



# Information Hiding

- All information about a module should be private to the module unless it is specifically declared public
- The assumption is made that every module is known to the rest of the world through some official description or interface
- The fundamental reason behind this principle is the continuity criterion.
  - If a module changes, but only in a way that affects its private elements, not the interface, then other modules who use it will not be affected
- Information hiding emphasizes the need to separate function from implementation

# Few Interfaces

- Every module should communicate with as few others as possible
- The few interfaces principle restricts the overall number of communication channels between modules in a software architecture
- This principle follows from the criteria of continuity and protection
  - If there are too many relations between modules, then the effect of a change or of an error may propagate to a large number of modules
- It is also connected to composability, understandability and decomposability
  - To be reusable in another environment, a module should not depend on too many others

# Small Interfaces

- If any two modules communicate at all, they should exchange as little information as possible
- It stems from the criteria of continuity (propagation of changes) and protection (propagation of errors)

# Enabling Factors

- **Uniform access**
  - Access to services of a module are expressed in a uniform notation independent of the implementation
    - Facilities are accessible to its clients in the same way whether implemented by computation (routine) or by storage (attribute)

# Enabling Factors

- **Open-Closed principle**
  - A satisfactory modular decomposition technique should yield modules that are both open and closed
    - Open Module: is one still available for extension
      - This is necessary because the requirements and specifications are rarely completely understood from the system's inception
    - Closed Module: is available for use by other modules, usually given a well-defined, stable description and packaged in a library
      - This is necessary because otherwise code sharing becomes unmanageable because reopening a module may trigger changes in many clients

# Modularity Guidelines

- **Coupling**
  - The degree of independence of modules from each other
    - The relationship between different modules
- **Cohesion**
  - The degree of the inseparability of the structural and behavioral features of a single module
    - The relationship of the elements within the module

***Achieve Weak Coupling***

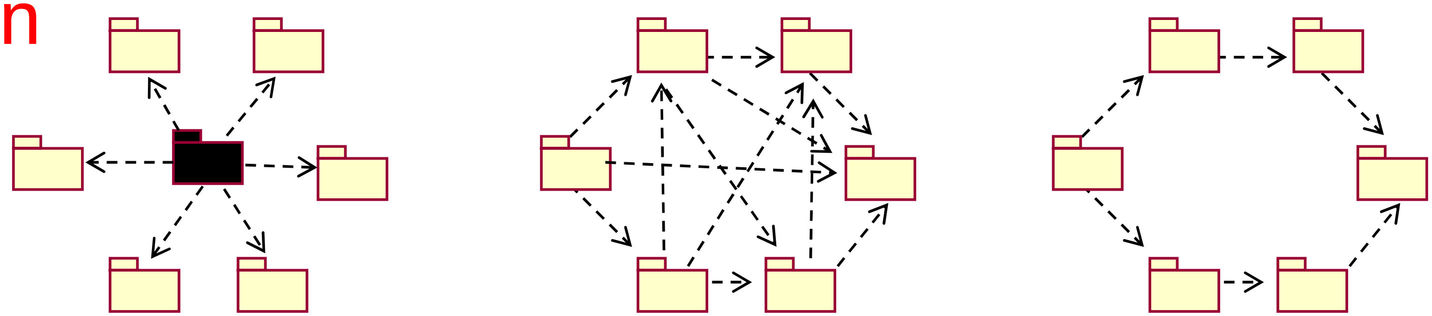
***Achieve Strong Cohesion***

# Modularity Guidelines

- **Coupling: Direct Mapping**
  - The modular structure of the software should remain compatible with the modular structure obtained by modeling the problem domain
  - Helps to achieve: continuity, decomposability

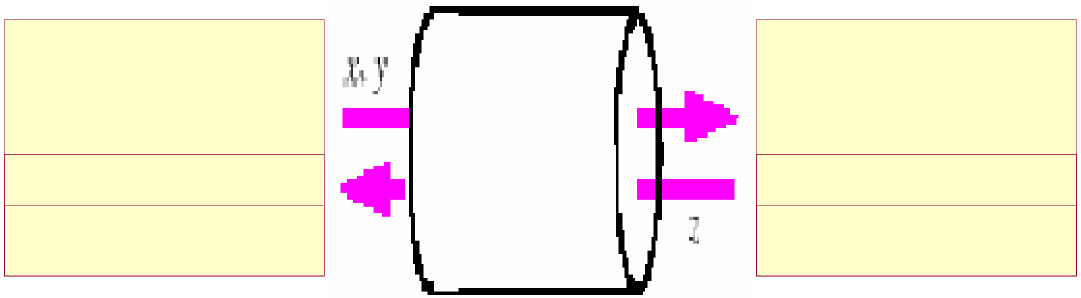
# Modularity Guidelines

- **Coupling: Few Interfaces**
  - Every module should depend on (communicate with) as few others as possible
    - The overall number of communication channels between modules should be as small as possible
  - Helps to achieve: understandability, composability, protection



# Modularity Guidelines

- **Coupling: Small Interfaces**
  - When two modules communicate, they should exchange as little information as possible
  - Helps to achieve: understandability, composability, continuity



# Modularity Guidelines

- **Coupling: Hierarchical, Non-Cyclic Dependencies**
  - there should be many clusters of modules that are independent, so they can be understood and reused separately
  - helps to achieve: understandability, decomposability, composability

# Modularity Guidelines

- **Coupling: Single Choice principle**
  - whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list

```
case p of
    book:        ....     //may access the field p. publisher
    journal      ....     //may access fields p.volume, p.issue
    proceedings:          //may access fields p.editor, p.place
end
```

# Guidelines

- **Cohesion: Separation of concerns**
  - Modules should address completely separate issues
  - Helps to achieve: understandability, weak dependencies so few interfaces and small interfaces

# Guidelines

- **Cohesion: Inseparability of addressed concerns**
  - Issues dealt within the same module should inseparably belong together
  - Helps to achieve: composability

# Guidelines

- **Cohesion: Completeness**
  - A module should implement all aspects of the concern (abstraction) it addresses
  - Helps achieve: composability

# Cohesion Levels

- Coincidental cohesion (weak)
  - Parts of a component are simply bundled together
  - There is no meaningful relationship between the elements of a module
    - Example: Arbitrary decomposition every 100 lines
- Logical association (weak)
  - Components which perform similar functions are grouped
  - The elements all relate to an apparent external function; e.g. we might provide, as one module, all input routines
    - Example: Libraries, routines for error handling

# Cohesion Levels

- Temporal cohesion (weak)
  - Components which are activated at the same time are grouped
  - The elements relate in time – e.g. all initialization activities could be grouped in one module
    - Example: Initializing, finalizing

# Cohesion Levels

- Communicational cohesion (medium)
  - All the elements of a component operate on the same input or produce the same output
  - The module works on the same data
    - Example: Procedures for Date in a calendar package
- Sequential cohesion (medium)
  - The output for one part of a component is the input to another part

# Cohesion Levels

- Functional cohesion (strong)
  - Each part of a component is necessary for the execution of a single function
  - All elements in the module relate to the performance of one function in the system
    - Iterator operations of a collection
- Object cohesion (strong)
  - Each operation provides functionality which allows object attributes to be modified or inspected
  - The module provides all the services connected with one data structure
    - Abstract Data Types, Objects

# Coupling Levels

- Content Coupling (strong)
  - One module directly references elements in another module without privacy or access control
    - Example: One component changes the code of another one (especially in assembler)
- Common Coupling (strong)
  - Two or more modules have free access to elements in common space
    - Example: Global variables, COMMON-spaces in FORTRAN

# Coupling Levels

- Control Coupling (medium)
  - One module controls the other one through steering parameters
  - Example: all sorts of (globally) accessible steering parameters
- Stamp Coupling (medium)
  - Each module lists the names of elements to which it will grant external access and names modules to which it requires access
    - Example: modules in Modula-2, packages in Java, ...

# Coupling Levels

- Data Coupling (weak)
  - All inter-module calls are phrased as procedure calls, with transmission of single element parameters only
  - Example: procedures with parameters (including calls by-reference)
- Function Coupling (weak)
  - As in data coupling, but without side effects
    - Example: functions (only call by-value)
- No Coupling (weak)

# Modularity and Quality

*Language has* ***Enabling Factors***

*Designer applies* ***Guide Lines***

*Help to achieve*
***Modularity Goals***

*Help to achieve*
***Quality***