Software architecture and design concepts

Professor Hossein Saiedian

EECS 348: Software Engineering

Fall 2023

THE UNIVERSITY OF
KU KANSAS

- We have the software requirements; now what?

The key word in context (KWIC) problem

"The KWIC system index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order."
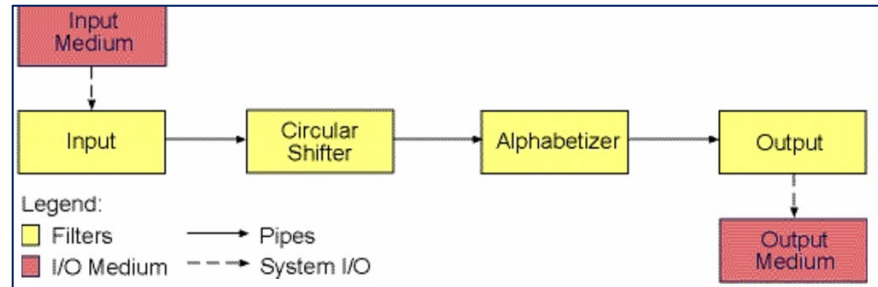
One spec, many designs
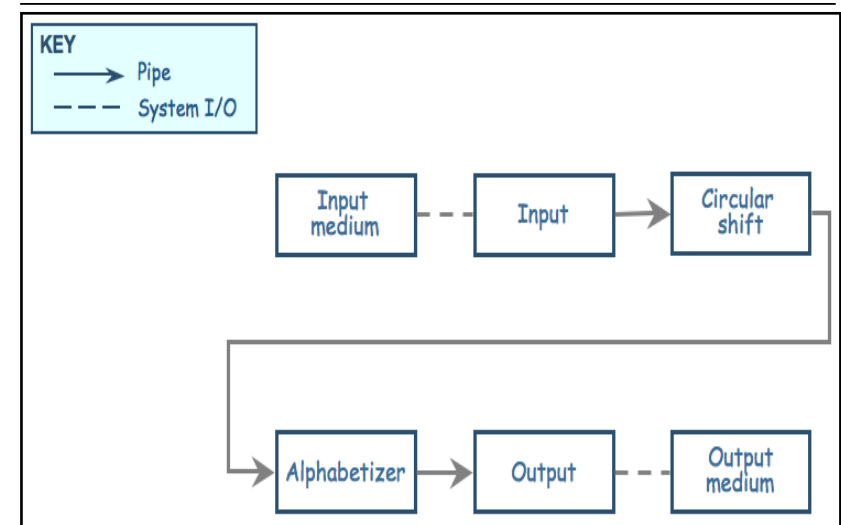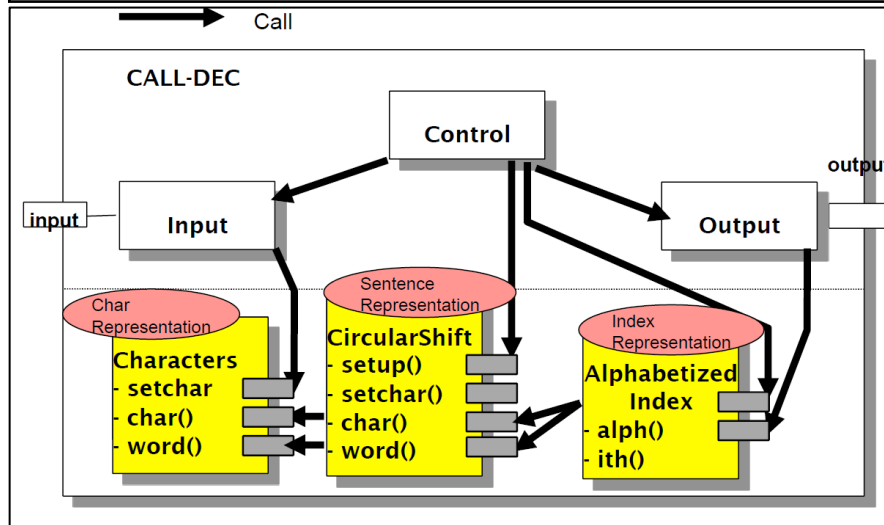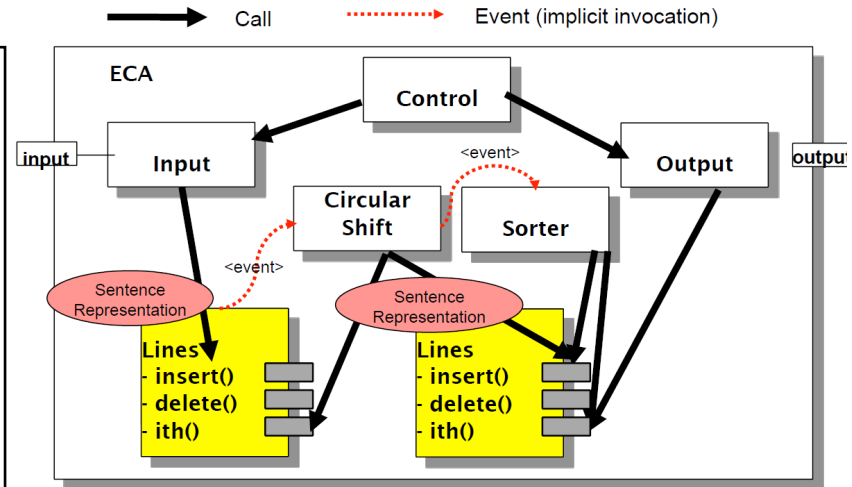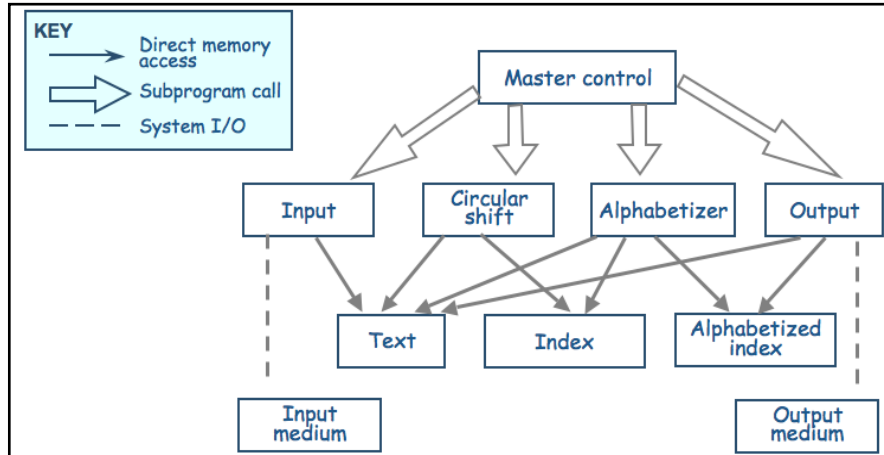
Factors to consider

Attribute	Priority	Shared data	Abstract data type	Implicit invocation	Pipe and filter
Easy to change algorithm	1	1	2	4	5
Easy to change data representation	4	1	5	2	1
Easy to change function	3	4	1	4	5
Good performance	3	5	4	2	2
Easy to reuse	5	1	4	2	5

- Other factors: modularity, testability, security, ease of use, ease of integration, extensibility

Software architecture

- To create a reliable, secure and efficient product, you need to pay attention to architectural design which includes:
  - its overall organization
  - how the software is decomposed into components
  - the server organization
  - the technologies that you use to build the software
- The architecture of a software product affects its performance, usability, security, reliability and maintainability.
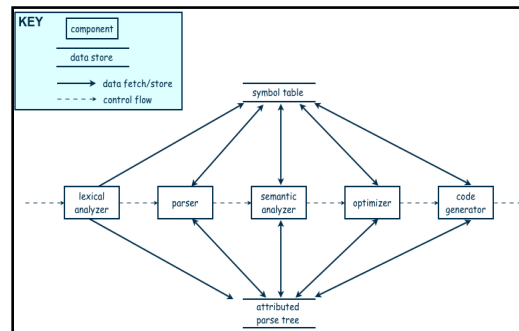
- A software system's architecture is the set of principal design decisions made about a system to be developed
  - A blueprint for the system construction and evolution

- A software architecture is represented in terms of
  - Software components
  - Software connectors
  - Configuration (topology)
  - A rationale for the decomposition, identification, or definition of the above

- A software system's architecture is the set of principal design decisions made about a system to be developed
  - A blueprint for the system construction and evolution

- A software architecture is represented in terms of
  - Software components
  - Software connectors
  - Configuration (topology)
  - A rationale for the decomposition, identification, or definition of the above
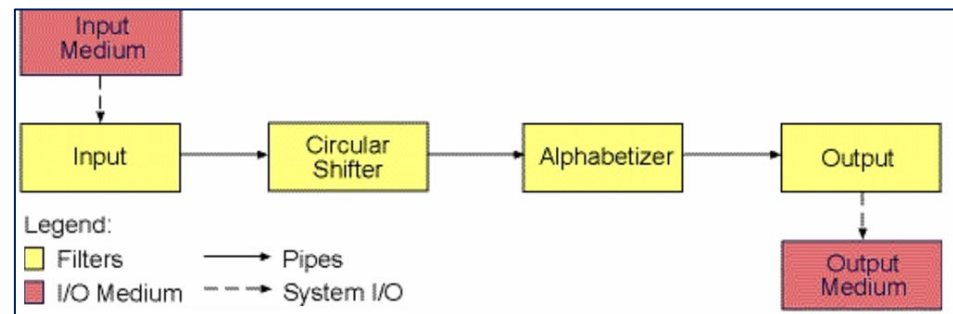
What is a software architecture

- A software system's architecture is the set of principal design decisions made about a system to be developed
  - A blueprint for the system construction and evolution
- A software architecture is represented in terms of
  - Software components
  - Software connectors
  - Configuration (topology)
  - A rationale for the decomposition, identification, or definition of the above
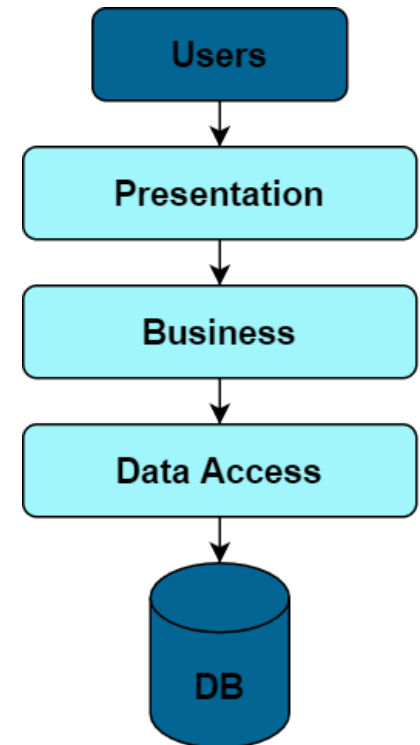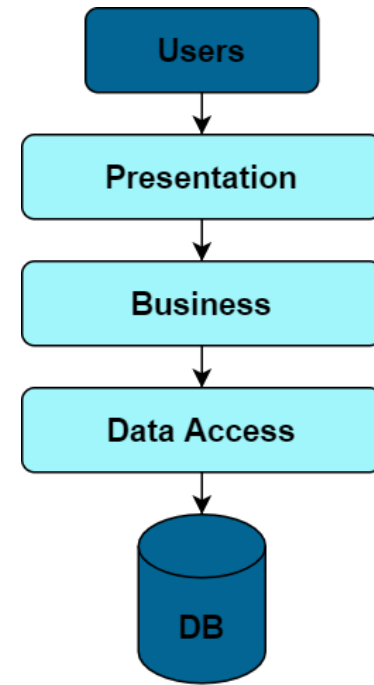
A software component

- Encapsulates a subset of the system's functionality and/or data

- Restricts access to that subset via an explicitly defined interface, and

- Embodies principles of encapsulation, modularity and abstraction
 - Imply composability, reusability, evolvability

How to choose an architecture?

- Domain-specific software architectures

- Reference architectures

- Architectural styles

Domain-specific software architectures

- Encode substantial knowledge acquired thru experience
  - A reference architecture for an application domain
  - Example: airline reservation systems; insurance applications
- DSSAs represent the most valuable type of experience useful in identifying a set of alternative arrangements for a design
- DSSA analogy
  - House designs by a housing development that are instantiated many times
  - Not identical: different colors, cabinetry, banister styles, carpet
  - The same structural elements: floor plan, location of windows, fireplace(s)

A reference architecture

- The ISO OSI model is an excellent reference architecture for internet applications

Architectural styles
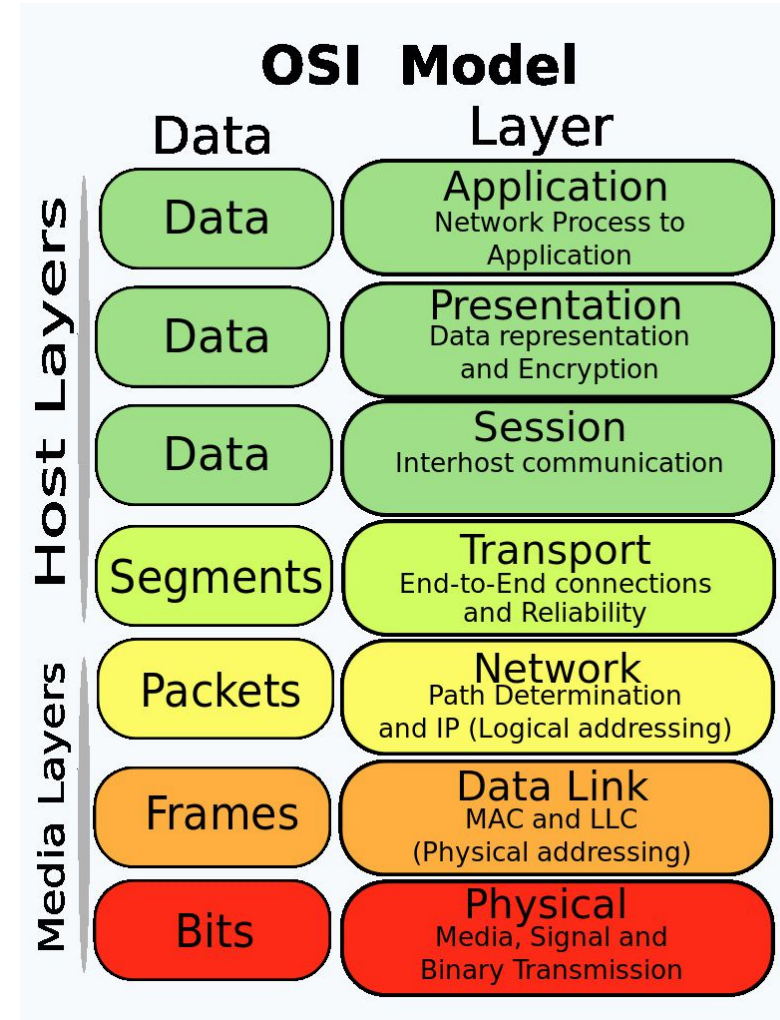
- An architectural style is a named set of architectural design decisions (that introduce useful qualities in each resulting system)
  - Traditional language-influenced styles (main program calling subroutines)
  - Object-oriented design
  - Layered (virtual machines, client server)
  - Data-flow styles (pipe and filters)
  - Shared memory
  - Implicit invocation
  - Peer-to-peer
  - …
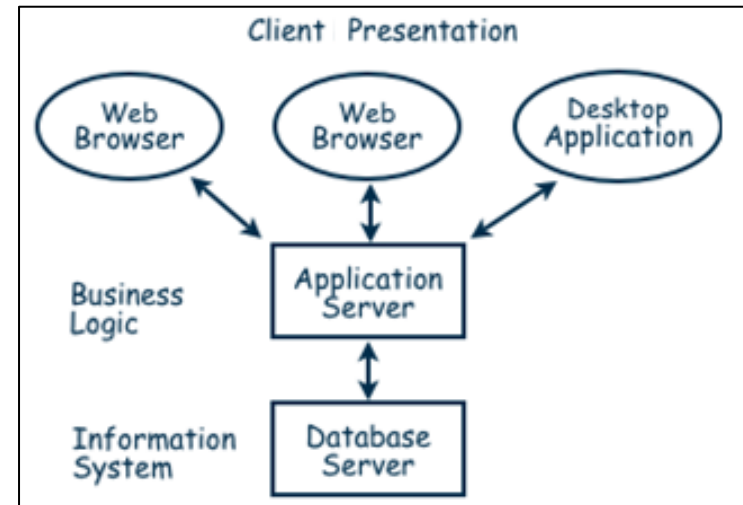
An architecture-centric approach to design
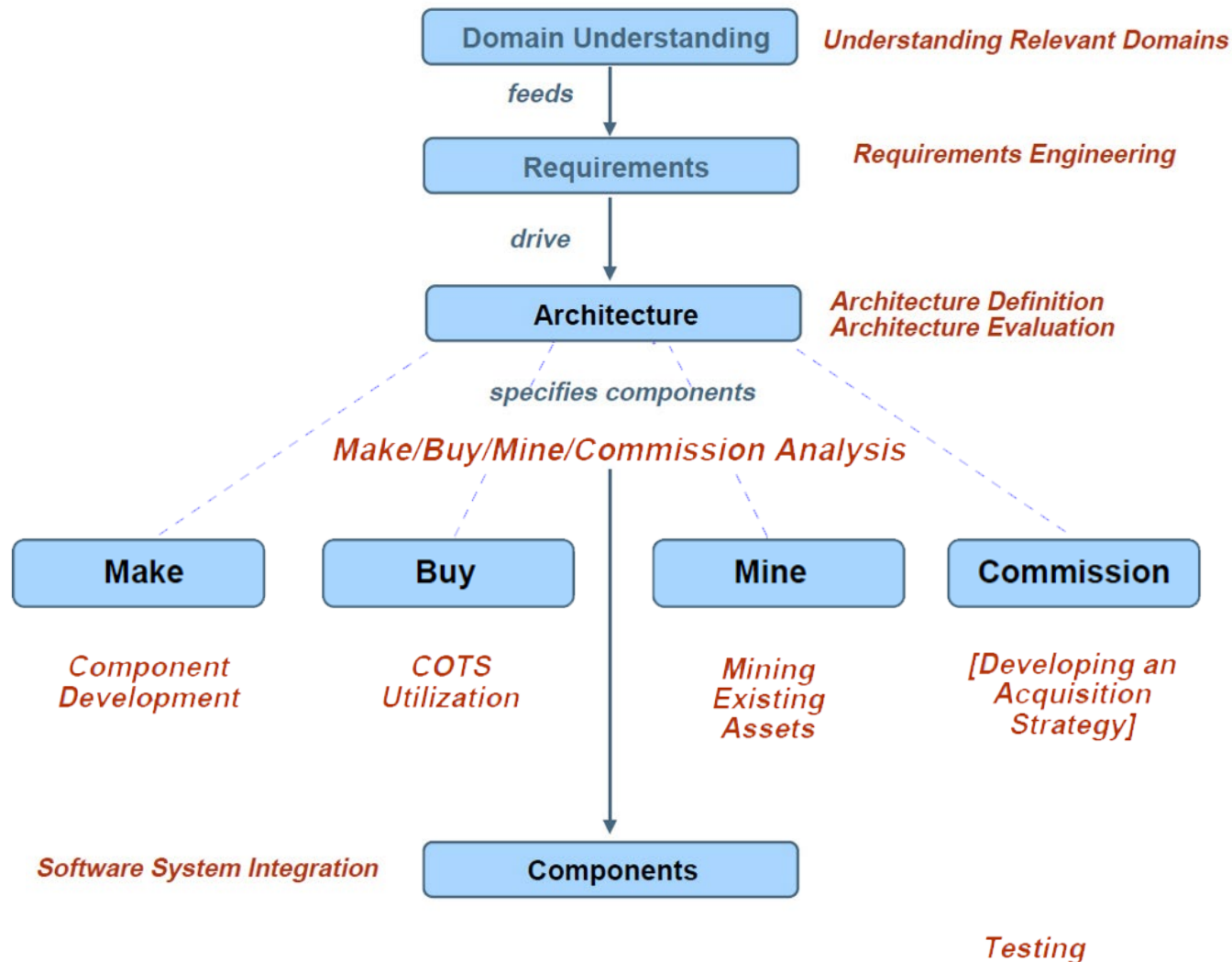
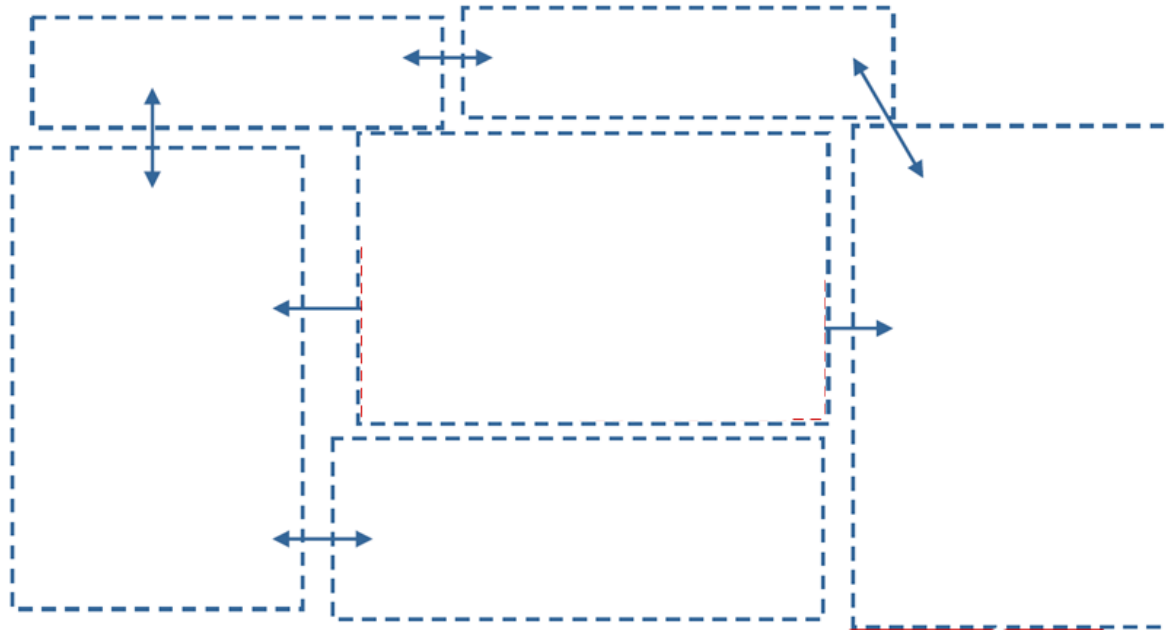- Bridge the gap between a problem and the expected solution

- Divide and conquer: model new system as a set of subsystems

- Choose architecture style (e.g., a layered architecture)

- Identify components (subsystems)

- Identify connectors (how the components will communication)

Subsystems (components) of a compiler

Lexer

Service:
- Scan input file and provide stream of tokens
- Initialize symbol table
- Report lexical errors

Features :
- next_token (File, ST)

Parser

Service:
- Parse token stream and build abstract syntax tree
- Enter symbol table information
- Report syntax errors

Features:
- AST(File, ST)

Static Analyzer

Service:
- Perform semantic analysis
- Fill symbol table
- Report type errors

Features :
- perform_analysis (AST, ST)

Code Generator

Service:
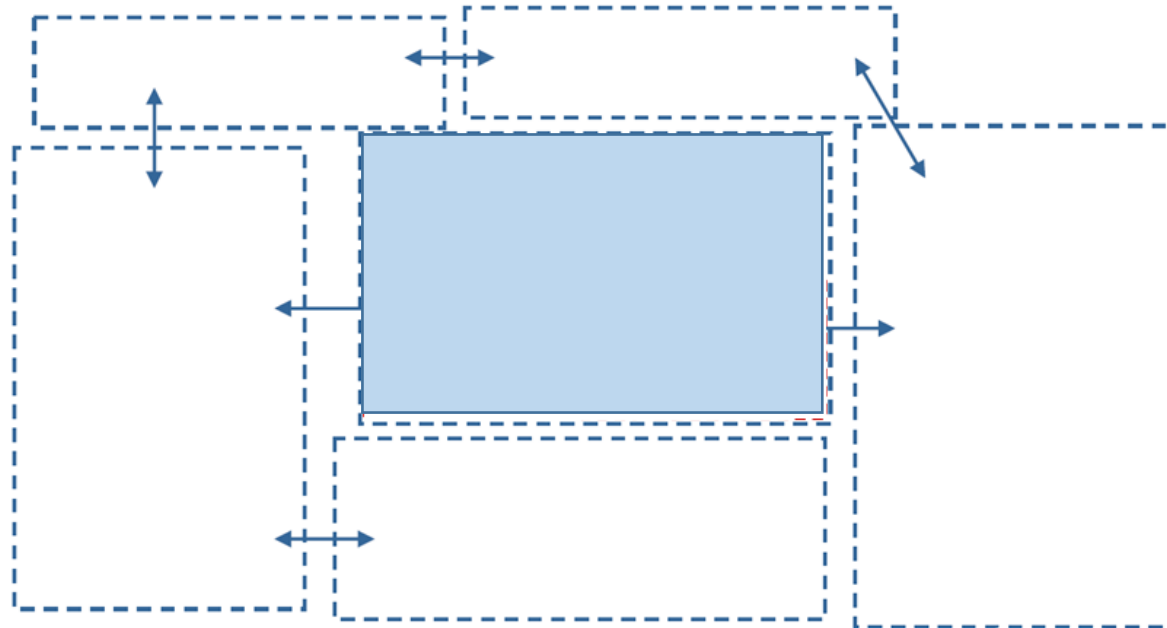- Generate target code from analyzed syntax tree

Features :
- generate_code(AST, ST)

- Choose architecture style (e.g., a layered architecture)

- Identify components (subsystems)

- Identify connectors (how the components will communication)

- Choose architecture style (e.g., a layered architecture)
- Identify components (subsystems)
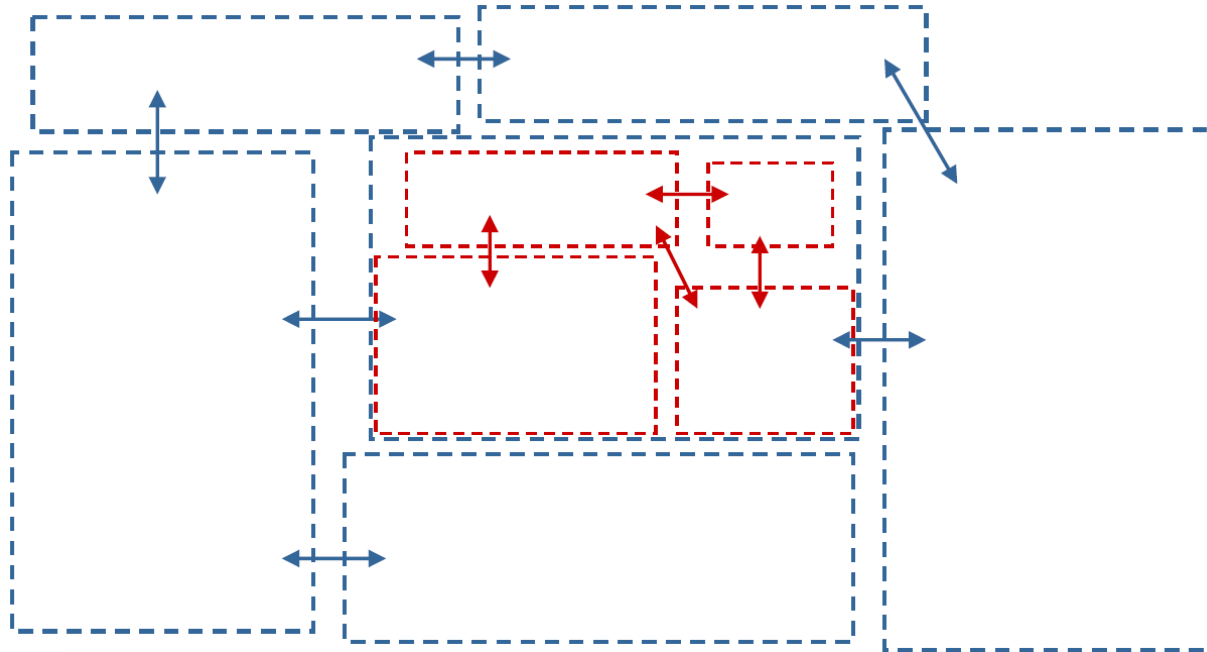- Identify connectors (how the components will communication)

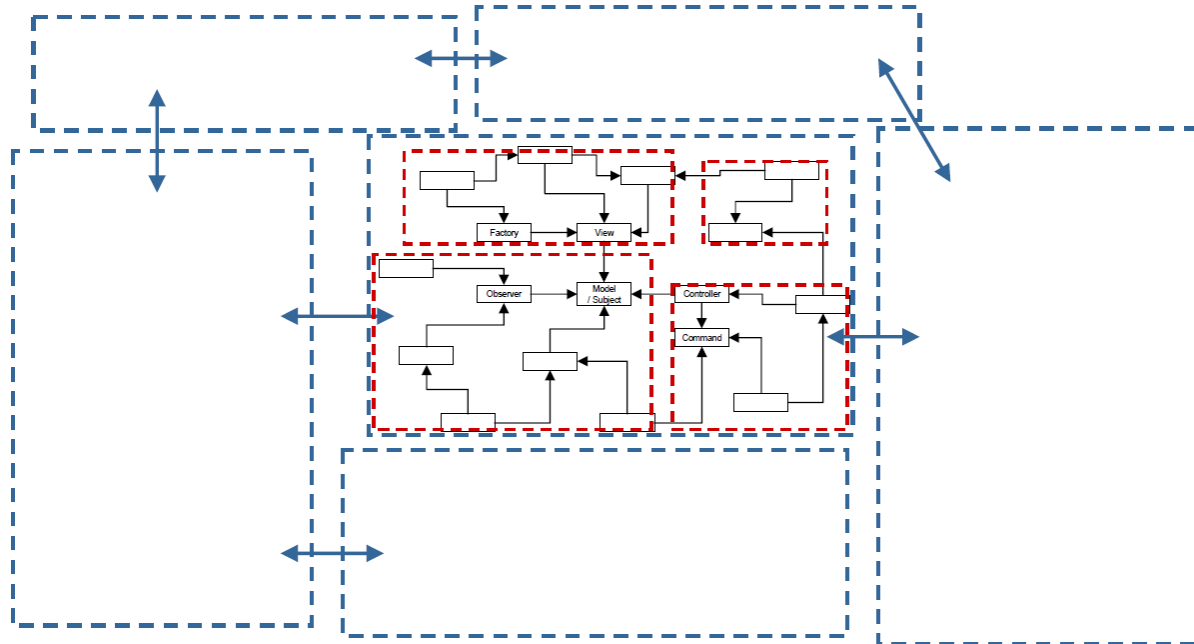# Develop software architecture

- Choose architecture style (e.g., a layered architecture)

- Identify components (subsystems)

- Identify connectors (how the components will communication)
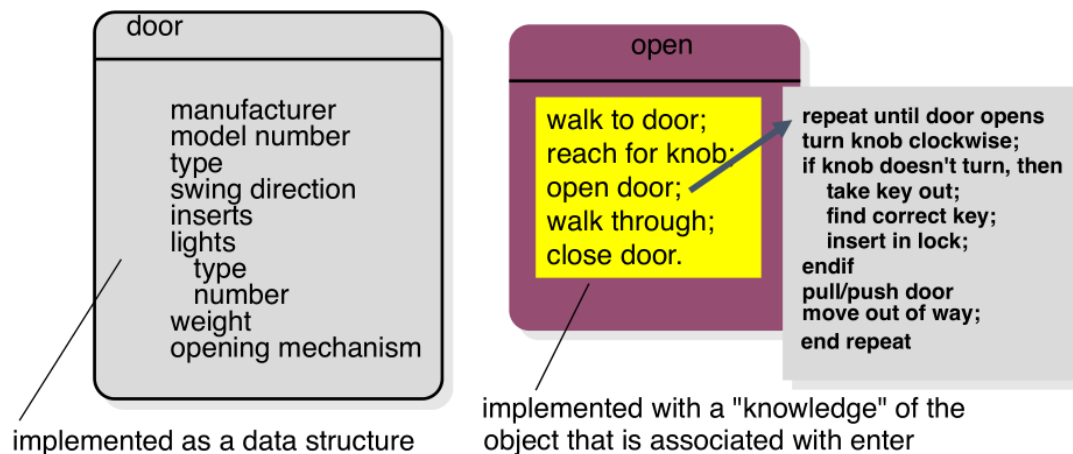
Why decompose

- Management
  - Partition effort
  - Clear assignment of requirements to modules

- Modification
  - Decouple parts so that changes to one don't affect others

- Understanding
  - Allow understanding system one chunk at a time

- Abstraction: present a problem and solution at appropriate levels of abstraction
  - Procedural abstraction: a sequence of instructions of a specific and limited function, with details of the function suppressed
  - Data abstraction: a collection of data that describes a data object

implemented as a data structure

implemented with a "knowledge" of the object that is associated with enter

- Separation of concerns
  - "Concern": a feature specified by the requirement model
  - Lead to software modularity, functional independence, refinement, and aspects

- Functional independence
  - Each module addresses a specific subset of requirements
  - Each module has a simple interface
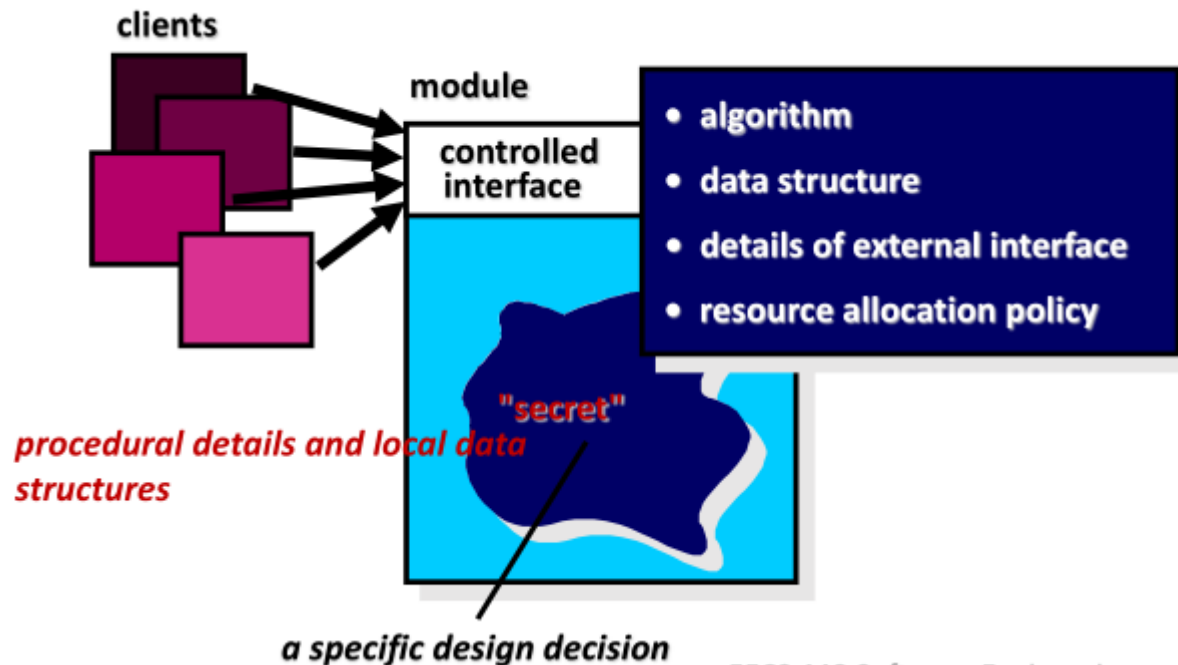
- Achieving functional independence
  - Cohesion: interdependence of elements of one module
  - Coupling: interdependence between different modules
  - Goal: high cohesion and low coupling
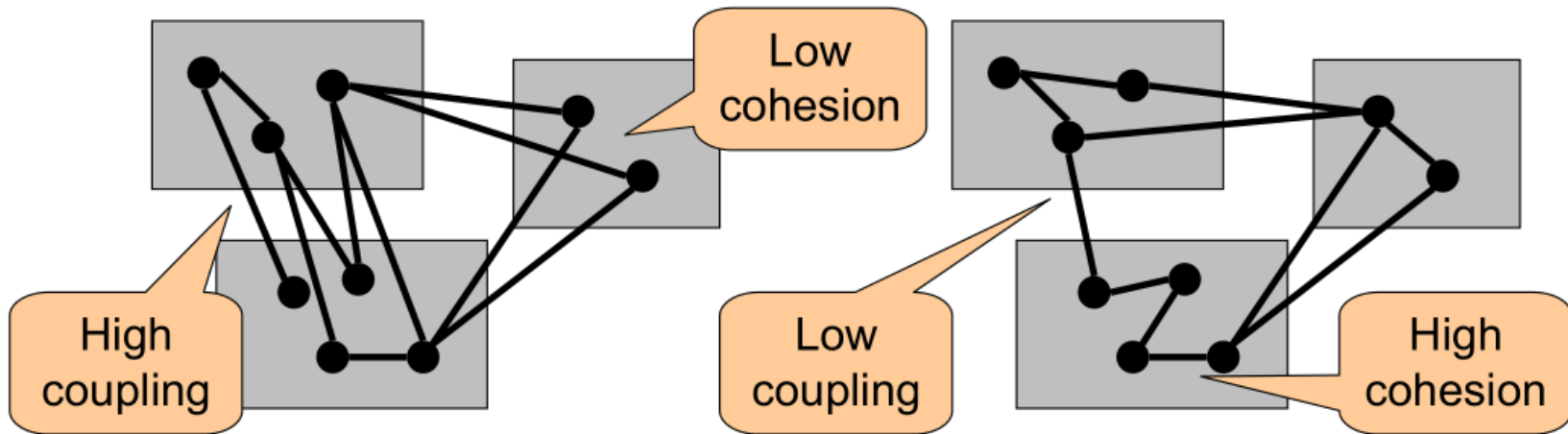
- Coupling is an indication of the relative interdependence among modules

- Two modules are tightly coupled when they depend a great deal on each other
  - Loosely coupled modules have some dependence, but their interconnections are weak
  - Uncoupled modules have no interconnections at all; they are completely unrelated

Great deal of dependence ⟶ **Independent**

☹ **Highly coupled**   **Loosely coupled**   **Uncoupled** ☺

Design concepts: coupling
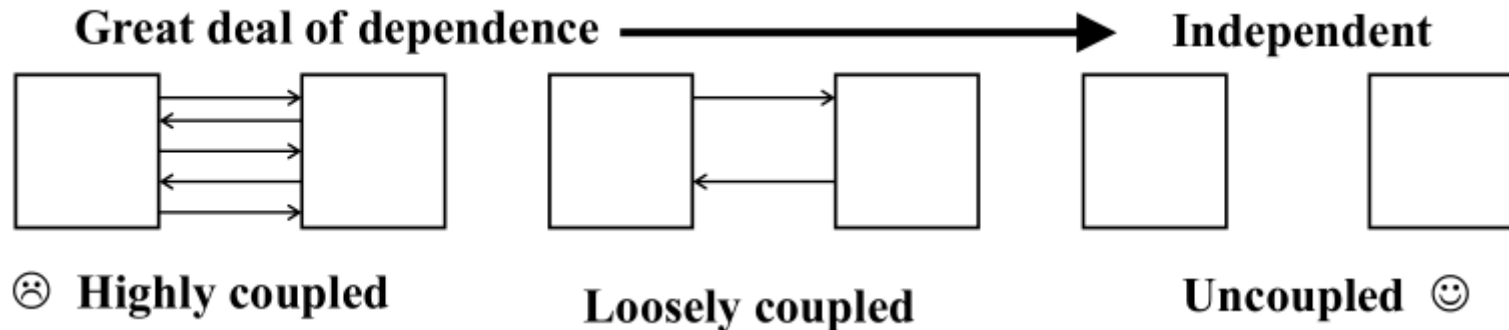
- Coupling: There are many ways that modules can be dependent on each other:
  - The references made from one module to another
  - The amount of data passed from one module to another
  - The amount of control that one module has over the other
- Coupling can be measured along a spectrum of dependence

- No direct coupling

- Data coupling (better)

- Stamp coupling

- Control coupling

- Common coupling

- Content coupling (worse)

Types of Modules Coupling
There are various types of module Coupling are as follows:

No Direct Coupling
Data Coupling
Stamp Coupling
Control Coupling
External Coupling
Common Coupling
Content Coupling

Best
Worst

Design concepts: Cohesion
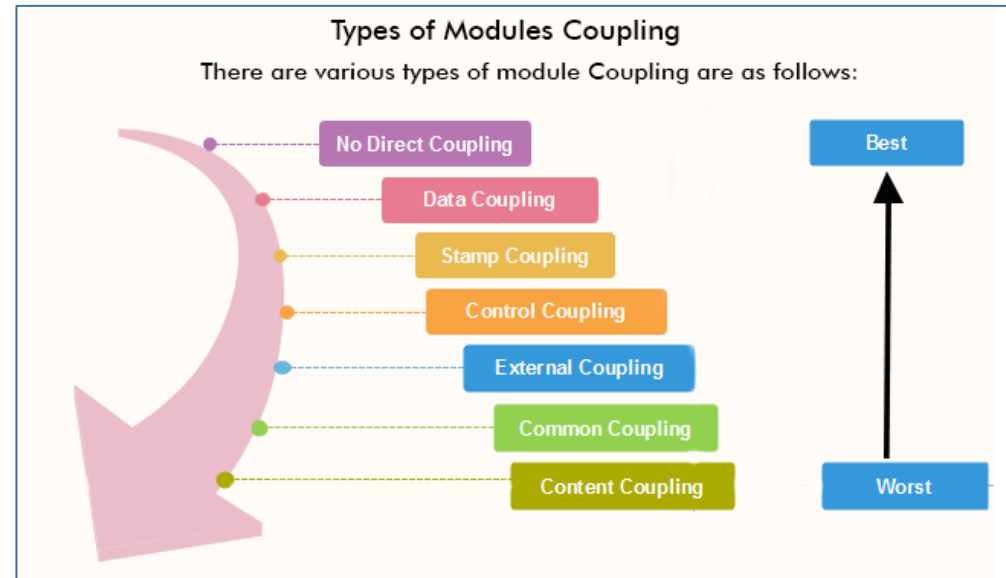
- Cohesion refers to the dependence within and among a module's internal elements (e.g., data, functions, internal modules)
  - Functional cohesion
  - Sequential cohesion
  - Communication cohesion
  - Procedural cohesion
  - Temporal cohesion
  - Logical cohesion
  - Coincidental cohesion

Types of Modules Cohesion

Functional Cohesion 01
02 Sequential Cohesion
Communication Cohesion 03
04 Procedural Cohesion
Temporal Cohesion 05
06 Logical Cohesion
Coin Cidental Cohesion 07

Best

Worst

Design concepts: Cohesion
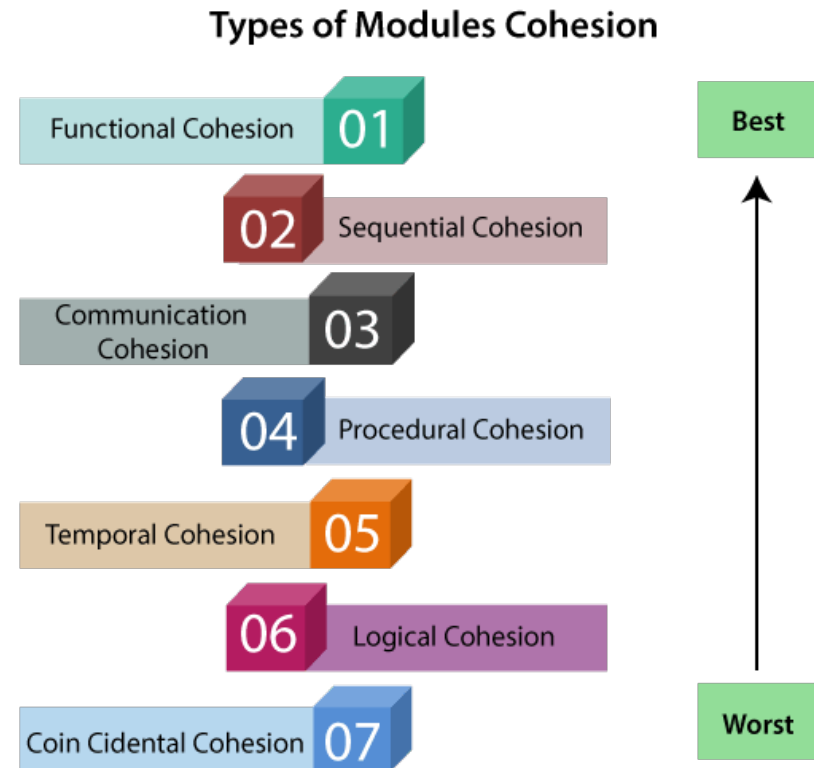
- Coincidental: Parts are unrelated to one another

- Temporal: Module's data and functions related because they are used at the same time in an execution

- Functional (ideal degree)
  - All elements essential to a single function are contained in one module, and all of the elements are essential to the performance of the function
  - An adaption of functional cohesion to data abstraction and object-based design
  - Data, actions, or objects are placed together

- Modularity
  - Helps development, increments and changes, testing and debugging, long-term maintenance
  - Increase cohesion, decrease coupling
  - Favored by architectural techniques tending to ensure decentralization of modules
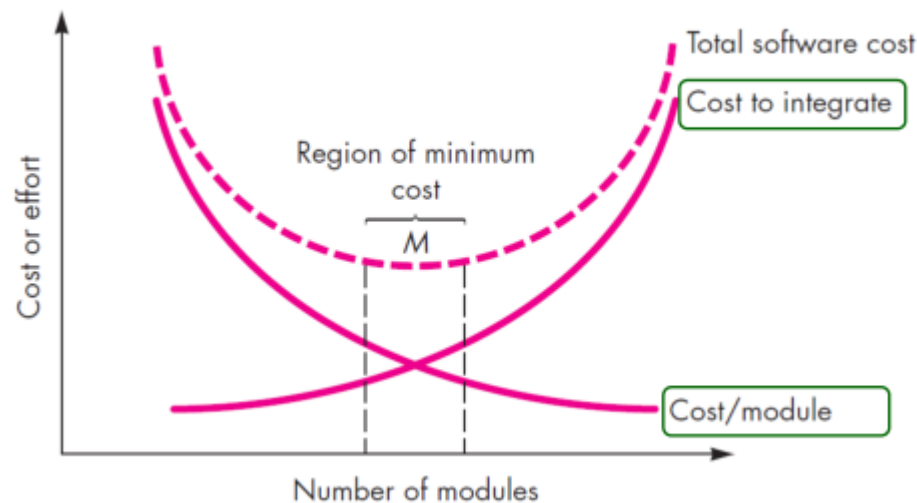
- Decomposability: Decompose complex systems into subsystems
  - Division of labor

Design concepts: Composability
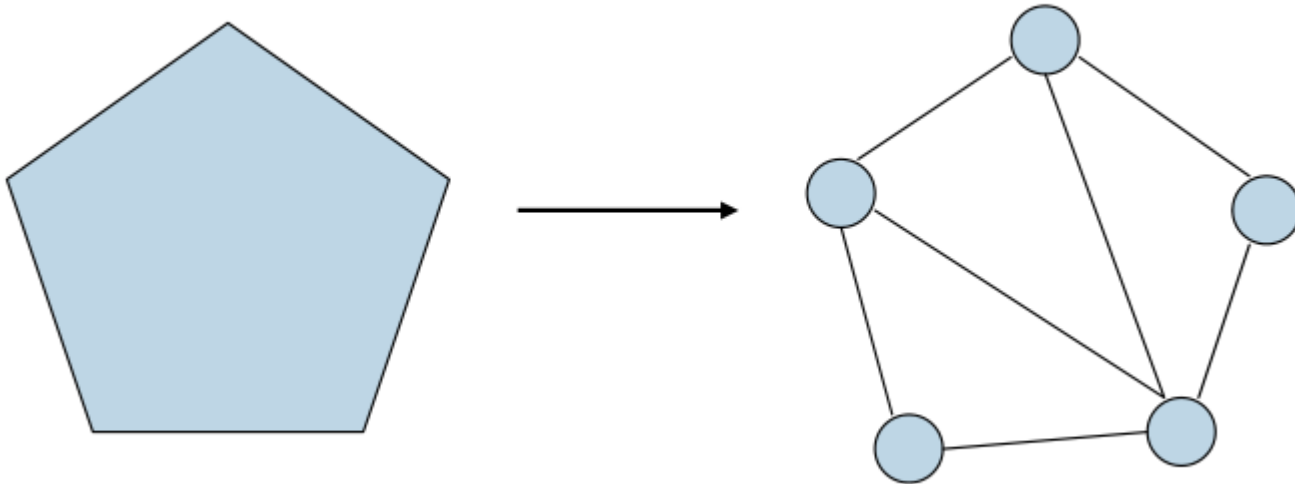
- Composability: Build software elements so that they may be freely combined with others to produce new software
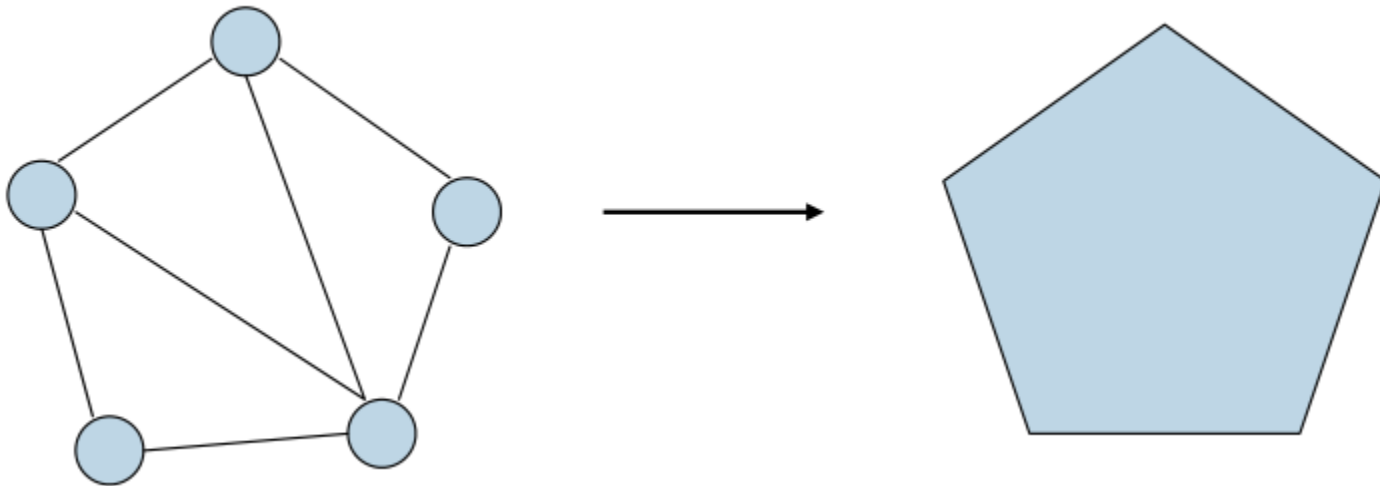
- Small interfaces principle: If two modules communicate, they exchange as little information as possible

- Explicit interfaces principle: Whenever two modules A and B communicate, this is obvious from the text of A or B or both
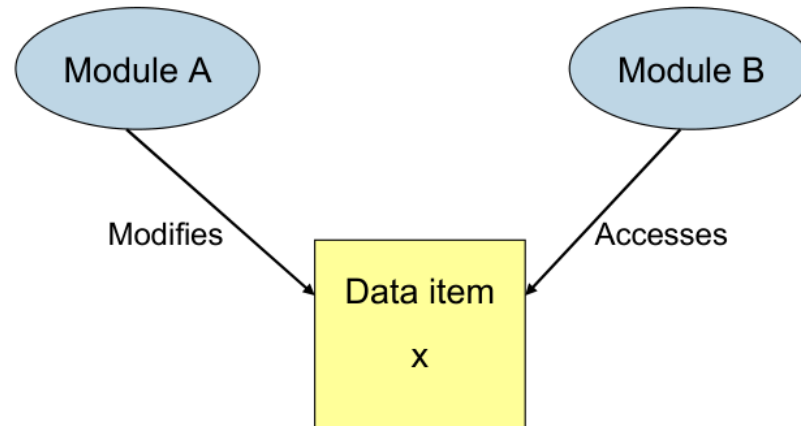
- Uniform access principle: A module's facilities are accessible to its clients in the same way whether implemented by computation or storage

- Not uniform: **a.balance, balance(a), a.balance()**

- Uniform: **a.balance()**

- Information hiding principle: The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules

- Information hiding: Every module should be known to the outside world through an official, "public" interface

- The rest of the module's properties comprises its "secrets"

- It should be impossible to access the secrets from the outside

Design concepts: Reusability

- Reusability: design for reusability to use parts of the design and implementation in other applications

- Type of reuse
  - Object code (or equivalent)
    * Example: sharing dll's between word processor and spreadsheet
  - Classes in source code form
    * Example: Customer class used by several applications
  - Assemblies of related classes (or pattern of classes)

Design concepts

- Refactoring
  - A reorganization technique
    * Simplifies the code of a component without changing its function
    * "Improves the internal structure of a design (or source code) without changing its external functionality or behavior"
  - Examine an existing design for
    * Redundancy
    * Unused design elements
    * Inefficient or unnecessary algorithms
    * Poorly constructed or inappropriate data structures
    * Any other design failure that can be corrected to yield a better design

- **Fan-in** refers to the number of units that use a particular software unit

- **Fan-out** refers to the number of units used by particular software unit

Design concepts: Interfaces
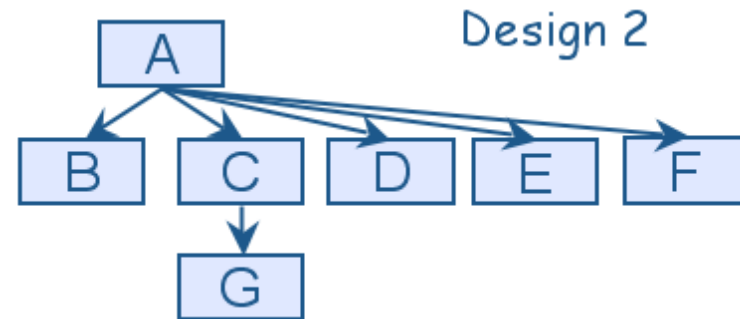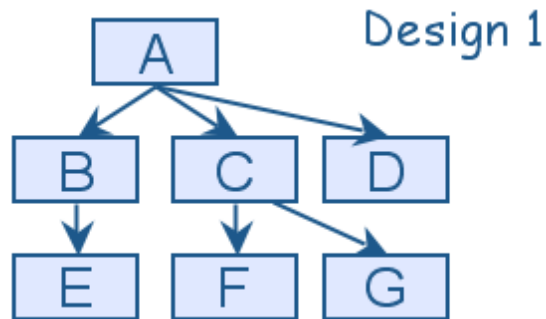
- Interfaces: The specification of a software unit's interface describes the externally visible properties of the software unit

- An interface specification should communicate to other system developers everything that they need to know to use our software unit correctly
  - Purpose
  - Preconditions (assumptions)
  - Protocols
  - Postconditions (visible effects)
  - Quality attributes

Design concepts: Generality

- Generality: a design principle that makes a software unit as universally applicable as possible, to increase the chance that it will be useful in some future system

- We make a unit more general by increasing the number of contexts in which it can be used
  - Parameterizing context-specific information

Design concepts

- Generality: The following four procedure interfaces are listed in order of increasing generality

```
PROCEDURE SUM: INTEGER;
POSTCONDITION: returns sum of 3 global variables


PROCEDURE SUM (a, b, c: INTEGER): INTEGER;
POSTCONDITION: returns sum of parameters


PROCEDURE SUM (a[]: INTEGER; len: INTEGER): INTEGER
PRECONDITION: 0 <= len <= size of array a
POSTCONDITION: returns sum of elements 1..len in array a


PROCEDURE SUM (a[]: INTEGER): INTEGER
POSTCONDITION: returns sum of elements in array a
```

OO design concepts

- Object-oriented methodologies are the most popular and sophisticated design methodologies

- A design is object-oriented if it decomposes a system into a collection of runtime components called objects
  - Objects are uniquely identifiable runtime entities that can be designated as the target of a message or request
  - Objects can be composed, in that an object's attributes may themselves be objects, thereby encapsulating the implementations of the object's internal variables
  - The implementation of an object can be reused and extended via inheritance, to define the implementation of other objects
  - OO code can be polymorphic: written in generic code that works with objects of different but related types

OO design terminology

- A class is a software module that partially or totally implements an abstract data type

- If a class is missing implementations for some of its methods, we say that it is an *abstract* class

- The class definition includes constructor methods that spawn new object instances

- Instance variables are program variables whose values are references to objects

OO design terminology

- Building new classes by combining component classes, much as children build structures from building blocks is done by object composition

- Alternatively, we can build new classes by extending or modifying definitions of existing classes
  - This kind of construction, called inheritance, defines a new class by directly reusing (and adding to) the definitions of an existing class

- Examples of inheritance

OO design terminology
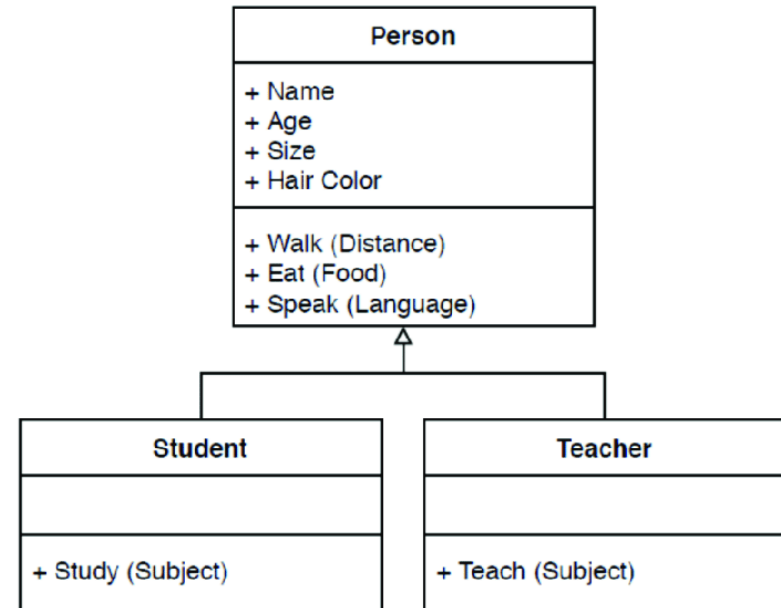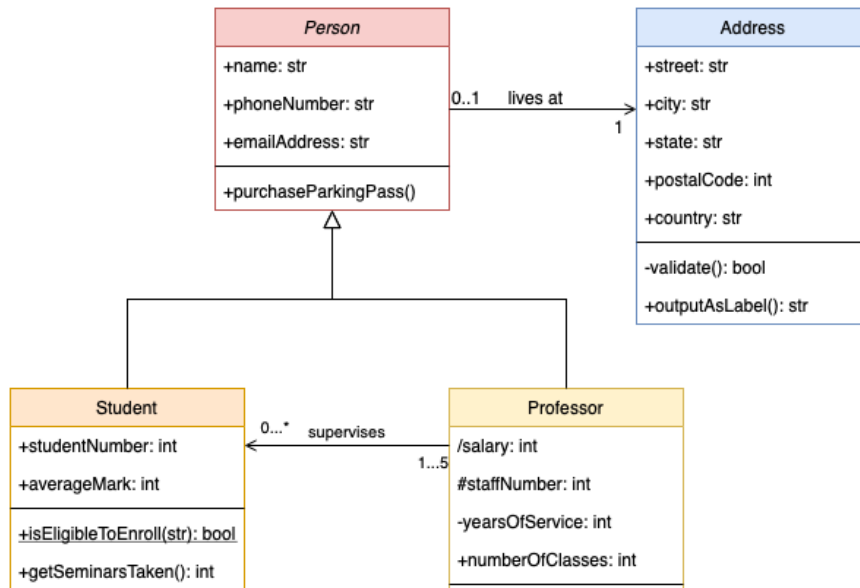
- Polymorphism occurs when code is written in terms of interactions with an interface, but code behavior depends on the object associated with the interface at runtime and on the implementations of that object's method

- Inheritance, object composition, and polymorphism are important features of an OO design that make the resulting system more useful in many ways

Types of polymorphism (C++)

- Operator and function overloading
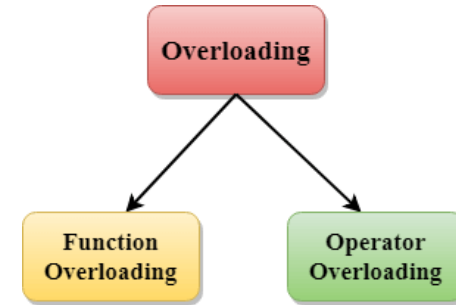
- Template classes
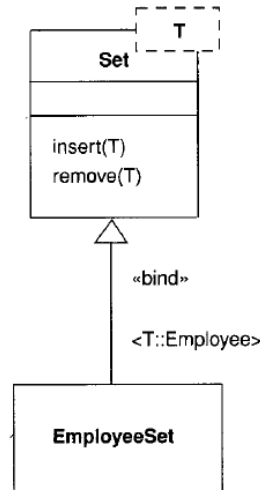
- More advanced pure polymorphism (runtime binding)

- Operator overloading
  - Example: binary operator +
  - Used for adding two numbers
  - Can be overloaded to add two complex numbers, two matrices, two vectors, concatenating two strings, etc.

    results = x + y;

  - Type compatibility is essential

Operator overloading

```cpp
1   // C++ program to overload the binary operator +
2   // This program adds two complex numbers
3
4   #include <iostream>
5   using namespace std;
6
7   class Complex {
8      private:
9       float real;
10      float imag;
11
12     public:
13      // Constructor to initialize real and imag to 0
14      Complex() : real(0), imag(0) {}
15
16      void input() {
17          cout << "Enter real and imaginary parts respectively: ";
18          cin >> real;
19          cin >> imag;
20      }
21
22      // Overload the + operator
23      Complex operator + (const Complex& obj) {
24          Complex temp;
25          temp.real = real + obj.real;
26          temp.imag = imag + obj.imag;
27          return temp;
28      }
29
30      void output() {
31          if (imag < 0)
32              cout << "Output Complex number: " << real << imag << "i";
33          else
34              cout << "Output Complex number: " << real << "+" << imag << "i";
35      }
36  };
```

Operator overloading

```cpp
int main() {
    Complex complex1, complex2, result;

    cout << "Enter first complex number:\n";
    complex1.input();

    cout << "Enter second complex number:\n";
    complex2.input();

    // complex1 calls the operator function
    // complex2 is passed as an argument to the function
    result = complex1 + complex2;
    result.output();

    return 0;
}
```
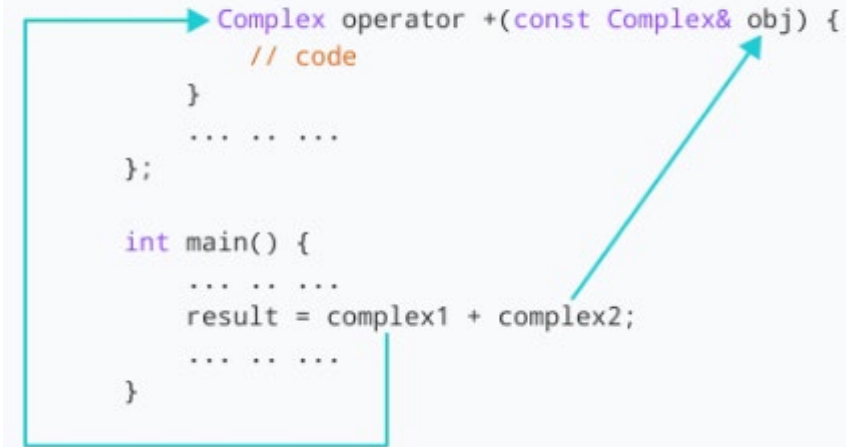
When you use a member function to overload a binary operator, the left-hand operand is the object that calls the overloaded operator

```cpp
class Complex {
    ... .. ...
    public:
        ... .. ...
        Complex operator +(const Complex& obj) {
            // code
        }
        ... .. ...
};

int main() {
    ... .. ...
    result = complex1 + complex2;
    ... .. ...
}
```

function call from complex1

Function overloading

- Function overloading (different from function overriding in specialization)
  - The number and type of the parameters determine which function is called

```cpp
void SumNum(int A, int B)
{
    cout<< endl << "SUMNUM is : "<< A+B;
}


void SumNum(int A, int B, int C)
{
    cout<< endl << "SUMNUM is : "<< A+B+C;
}


void SumNum(int A, int B, int C, int D)
{
    cout<< endl << "SUMNUM is : "<< A+B+C+D;
}
```

Template (generic) classes in C++

- A template class in C++ is a class that allows the programmer to operate with generic data types

- Allows the class to be used on many different data types as per the requirements without the need of being re-written for each type

- Consider a **stack** class
  - Develop one for integers?
  - Develop one for a **Book** type?
  - Develop one for **Task** type?
  - No; develop one class but make the type to be generic

Template classes in UML

- Template or parameterized (or generic) classes in UML
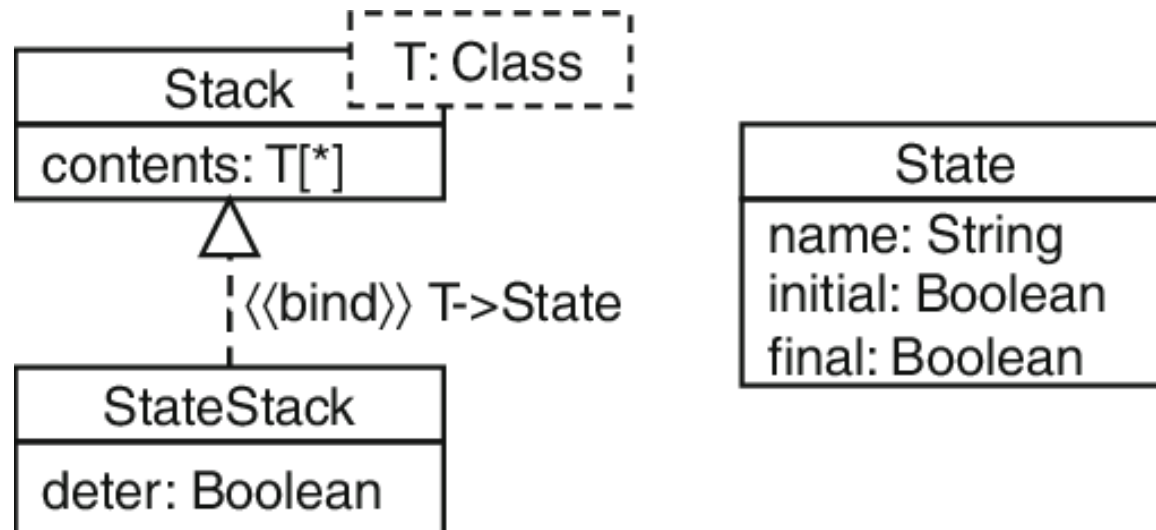
Template classes in ++

```cpp
1   // C++ Program to Implement stack using Class Templates
2   #include <iostream>
3   #include <string>
4   #define SIZE 5
5
6   // Class to represent stack using template by class
7   // taking class in template
8   template <class T> class Stack {
9       // Public access modifier
10  public:
11      // Empty constructor
12      Stack();
13
14      // To add element to stack which can be any type
15      // using stack push() operation
16      void push(T k);
17
18      // To remove top element from stack
19      // using pop() operation
20      T pop();
21
22      // To print top element in stack
23      // using peek() method
24      T topElement();
25
26      // To check whether stack is full
27      // using isFull() method
28      bool isFull();
29
30      // To check whether stack is empty
31      // using isEmpty() method
32      bool isEmpty();
33
34  private:
35      // Taking data member top
36      int top;
37
38      // Initialising stack(array) of given size
39      T st[SIZE];
40  };
```

```cpp
61  // To add element element k to stack
62  template <class T> void Stack<T>::push(T k)
63  {
64
65      // Checking whether stack is completely filled
66      if (isFull()) {
67          // Display message when no elements can be pushed
68          // into it
69          cout << "Stack is full\n";
70      }
71
72      // Inserted element
73      cout << "Inserted element " << k << endl;
74
75      // Incrementing the top by unity as element
76      // is to be inserted
77      top = top + 1;
78
79      // Now, adding element to stack
80      st[top] = k;
81  }
82
```

```cpp
138  // Create object of Stack class in main()
139  // Declare objects of type Integer & String
140  Stack<int> integer_stack;
141  Stack<string> string_stack;
142
143  // Adding elements to integer stack object
144  // Custom integer entries
145  integer_stack.push(2);
146  integer_stack.push(54);
147  integer_stack.push(255);
```

Pure polymorphism

- The previous polymorphic examples (function overloading, operator overloading, and template classes) were handled at compile time

- Pure polymorphism is binding at compile time
  - Dynamic binding
  - Subtyping relationship
  - If S is a subtype of T, the subtyping relation means that any object of type S can safely be used in any context where an object of type T is expected

Pure polymorphism C++ example

```cpp
1   class Shape
2   {
3       //Pure virtual function declaration
4       public:
5       virtual void display() = 0;
6   };
7
8
9   class Rectangle: public Shape
10  {
11      public:
12      void display()
13      {
14          cout << "I'm a Rectangle" << endl;
15      }
16  };
17
18  class Circle: public Shape
19  {
20      public:
21      void display()
22      {
23          cout << "I'm a Circle" << endl;
24      }
25  };
```

```cpp
32  int main()
33  {
34      Shape *aptr; //Base class pointer
35
36      Rectangle r;  //derived class object creation.
37      Circle c;  //derived class object creation.
38
39      aptr = &r;
40      aptr->display();
41
42      aptr = &c;
43      aptr->display();
44
45      return 0;
46  }
```

- Domain-specific software architectures

- Reference architectures

- Architectural styles
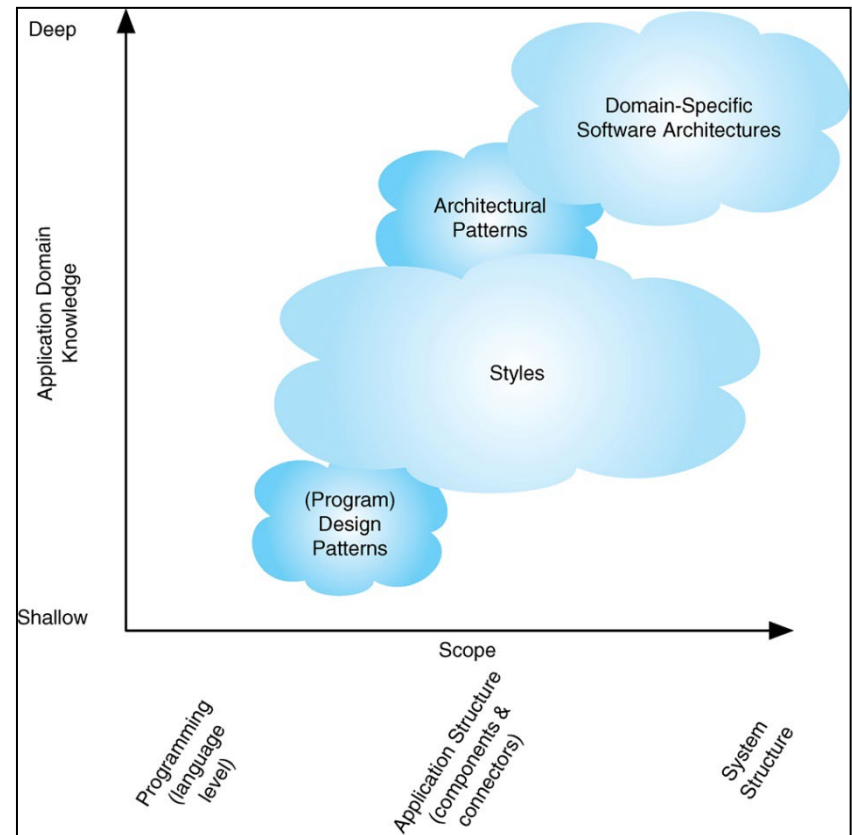
- Design patterns

Design patterns

- A design pattern is a general repeatable solution to a commonly occurring problem in software design
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
- Objectives
  - High readability and maintainability
  - High testability
  - High reusability
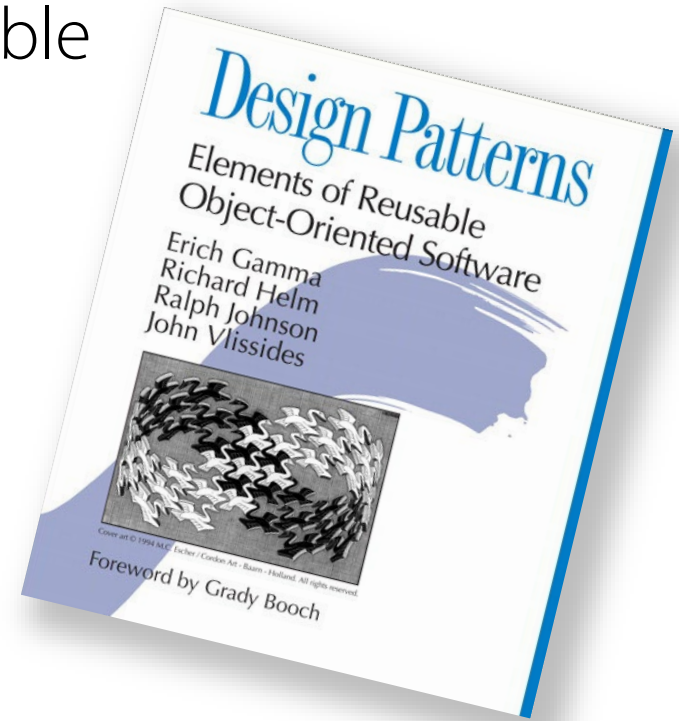
- Name
  - Describes the pattern
  - Adds to common terminology for facilitating
- Problem
  - Describes when to apply the pattern
- Solution
  - Describes elements, relationships, responsibilities, and collaborations which make up the design
- Consequences
  - Results of applying the pattern
  - Benefits and costs
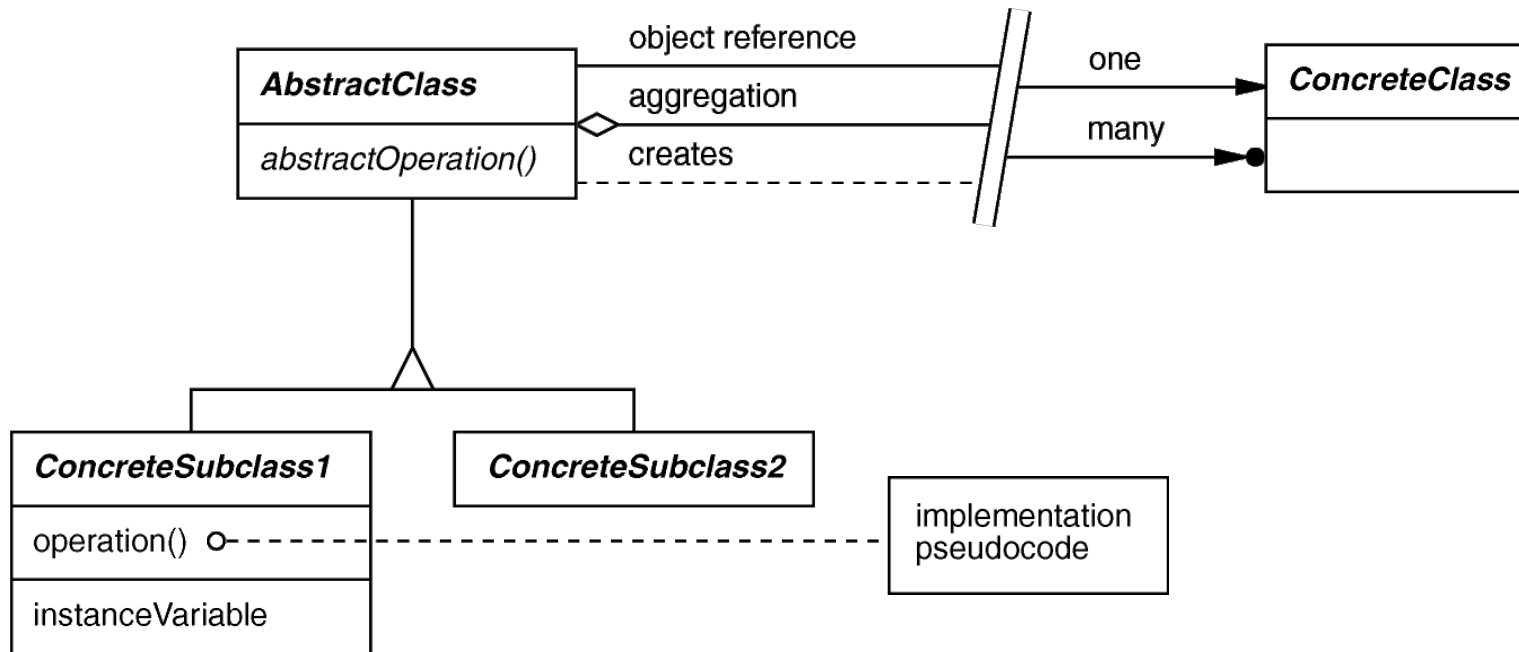
Design patterns classification

GoF: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

|		**Purpose**		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Scope: domain over which a pattern applies

Purpose: reflects what a pattern does

Adapter design pattern

- From the Wikipedia: "the adapter pattern is a software design pattern that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code."
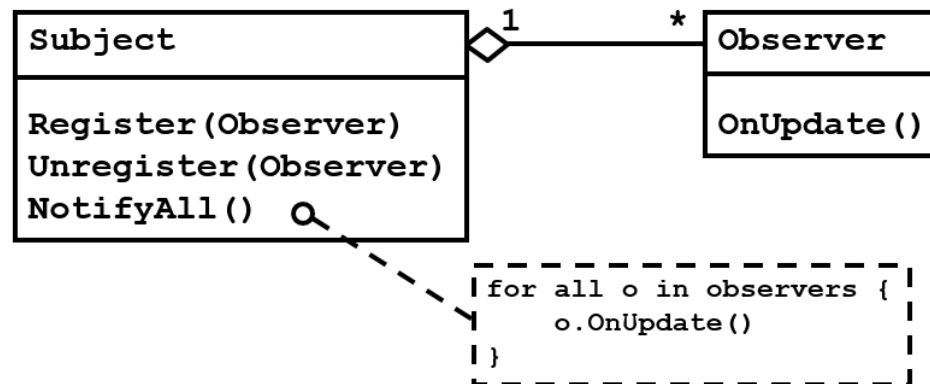
Adapter design pattern (structural)
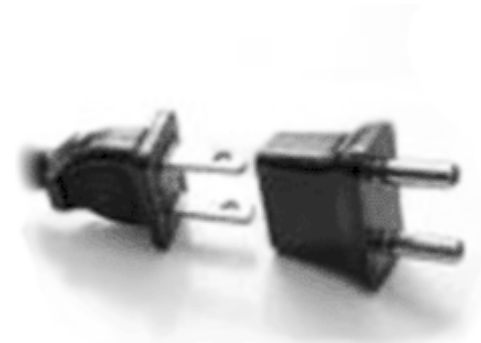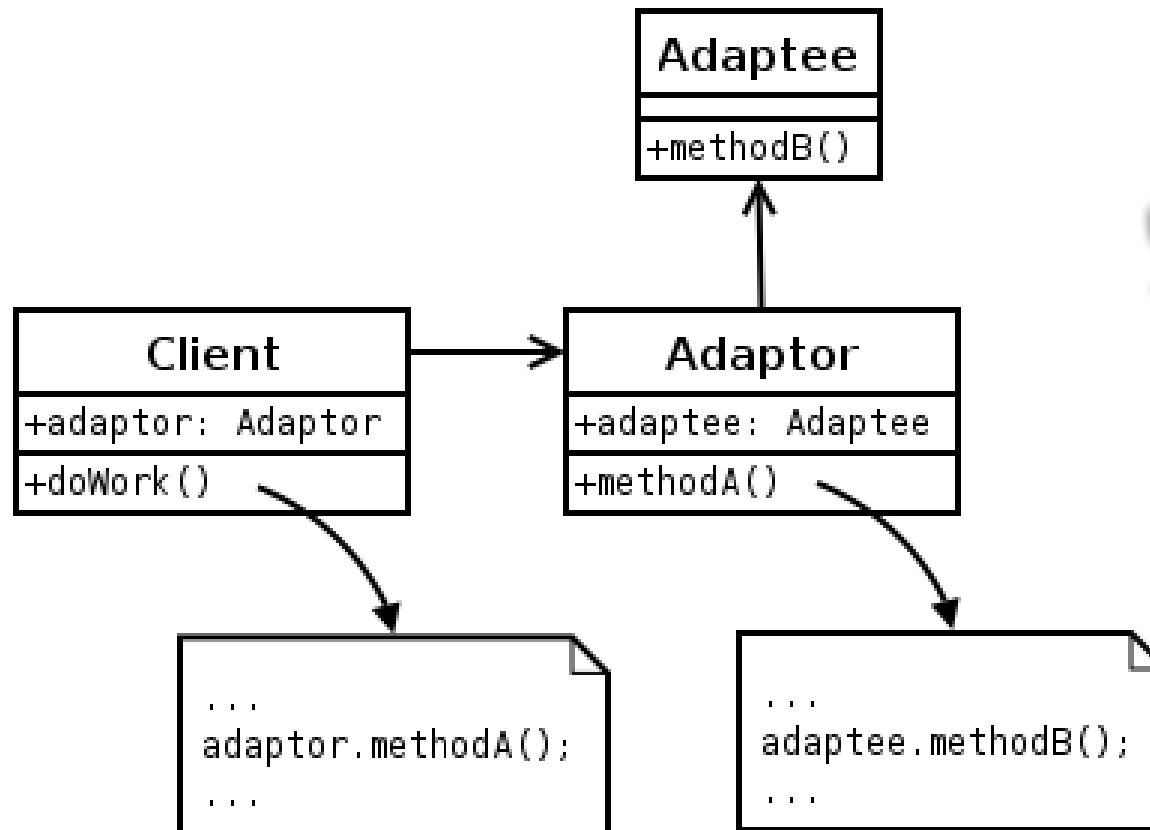
Observer design pattern
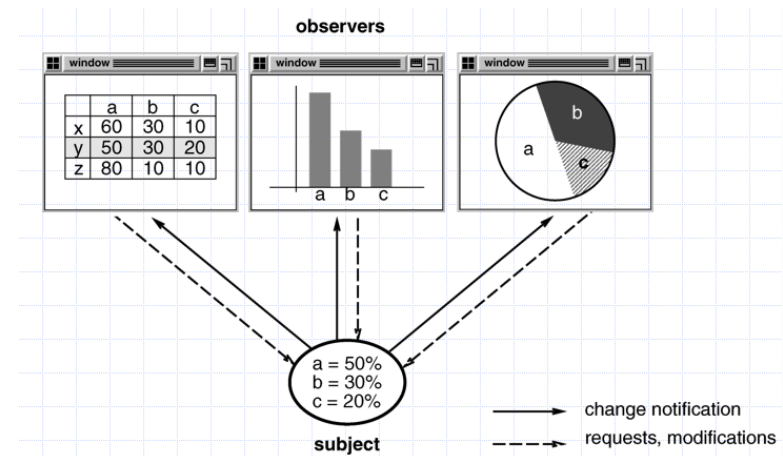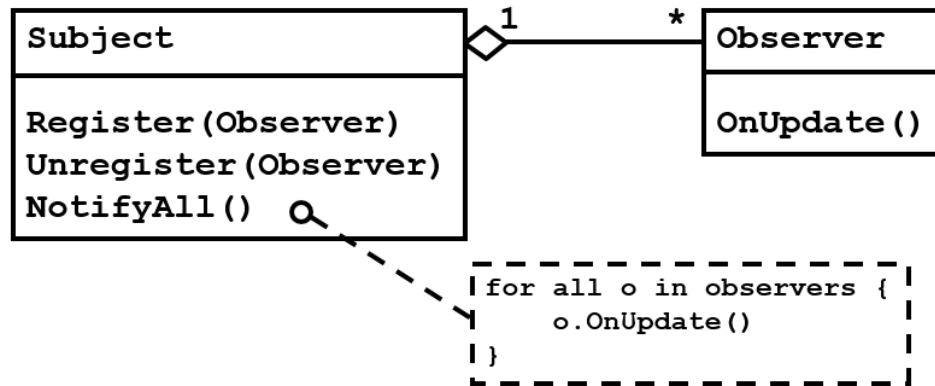
- Modularity: subject & observers may vary independently

- Extensibility: can define & add any number of observers

- Customizability: different observers offer different views of subject

- Unexpected updates: observers don't know about each other

- Some of the fundamental concepts in module design
  - A module
    * A function
    * A method
    * A class
- Design patterns