# Software Design and Architecture

Dr. Vu Thi Huong Giang

# What is software design ?

- Design as an activity: acts as a bridge between requirements and the implementation of the system
  - Decompose system into modules
  - Assign responsibilities to these modules
  - Ensure that these modules fit together to achieve a global goal
- Design as a result of an activity: gives a structure that makes the system easy to understand and evolve
  - Produce a Software Design Document
  - Often a software architecture is produced prior to a software design

# Covered topics

- Introduction
- Divide and conquer strategy
- Modularization techniques
- Architecture

# Objectives

- After this lesson, students will be able to:
  - Explain the divide and conquer strategy
  - Sketch the decomposition tree for a given software
  - Conduct modularization techniques with a given software
  - Distinguish the concepts of program structure and software architecture

# I. DIVIDE AND CONQUER STRATEGY

1. Introduction
2. Decomposition
3. Composition
4. Step refinement

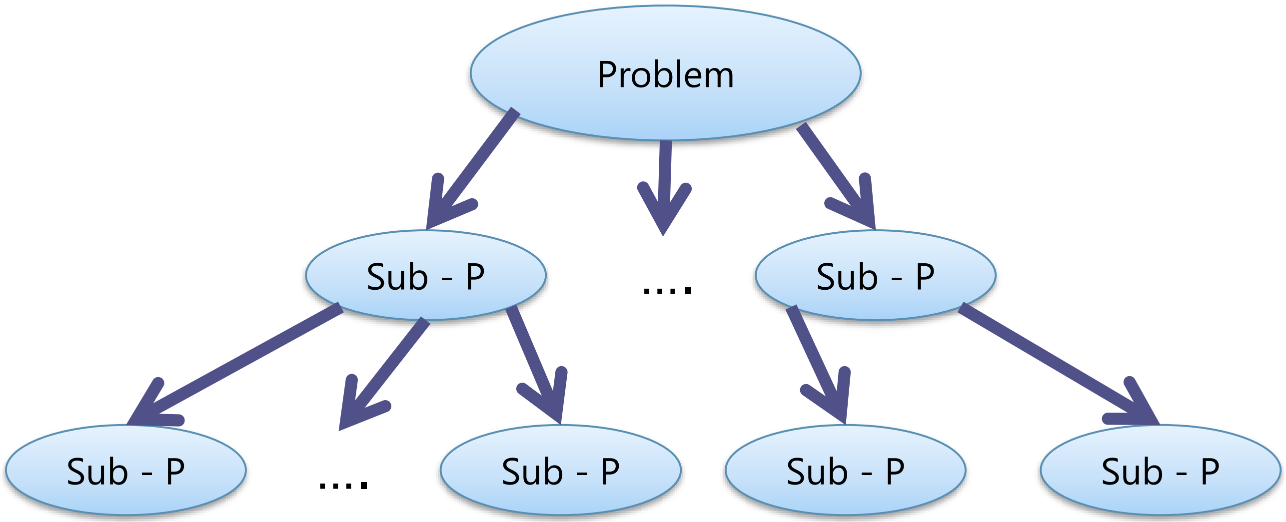
# Introduction

- Divide: decompose problems into sub-problems
- Conquer: solve sub-problems (hopefully easier)
- Assemble : compose the complete solution from the sub-solutions

- We decompose
  - To simplify the problem
  - To find solutions in terms of the abstract machine we can employ
  - When completed, we can compose
- However, we may choose where we begin:
  - Stepwise Refinement (top-down)
  - Assemblage and Composition (bottom-up)
  - Design from the middle (middle-out)

# 1. Decomposition

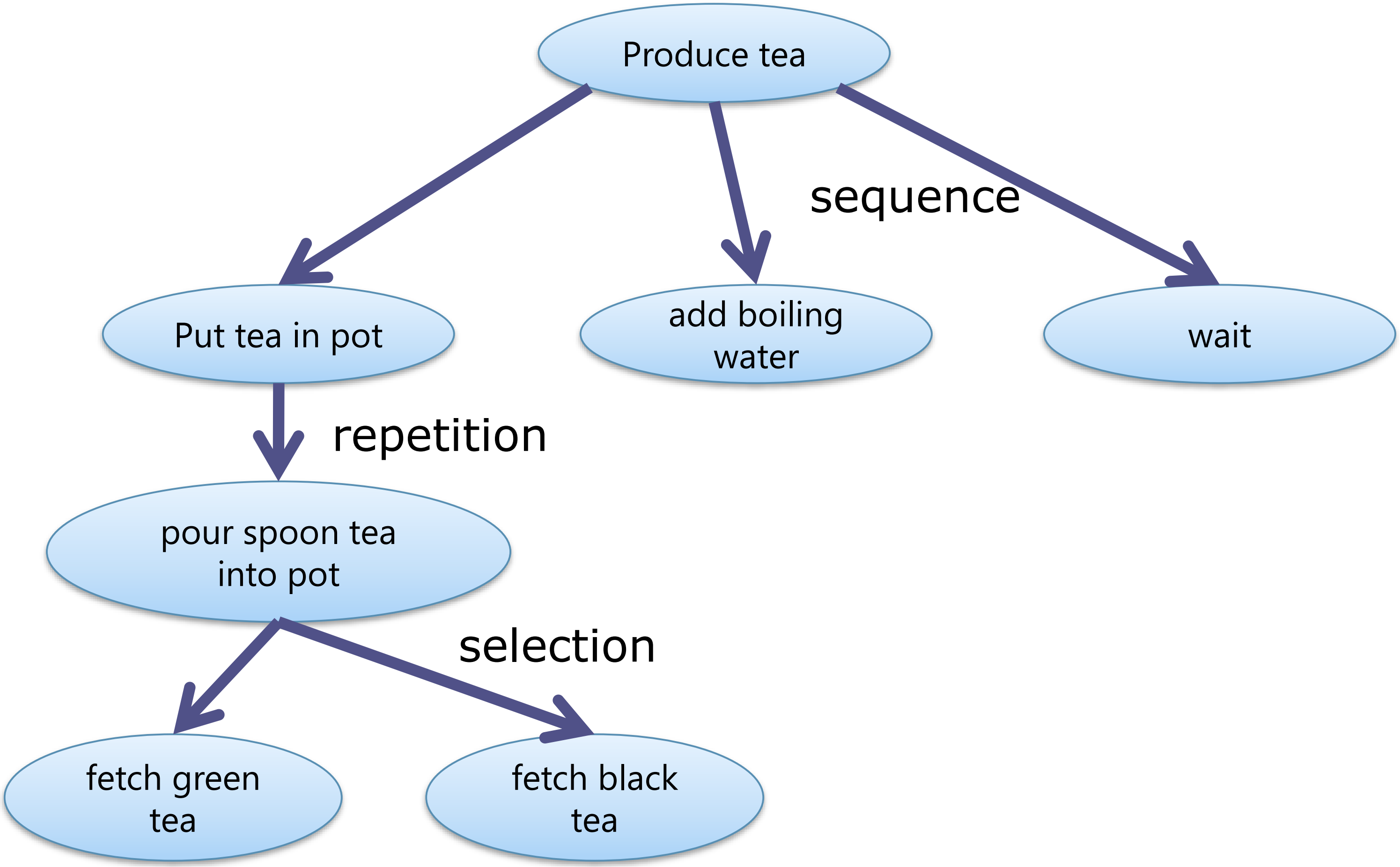
- The process by which a complex problem or system is broken down into parts that are easier to conceive, understand, program, and maintain



# 1.1. Functional decomposition

- Break a function down into its constituent parts in such a way that the original function can be reconstructed (i.e., recomposed) from those parts by function composition.
  - What are the functions of my software?
  - What does the software do?
  - Which sub-functions exist?
  
  → functions are grouped to modules/components

# Example: abstract machine



# Working: Banking system

Transfer

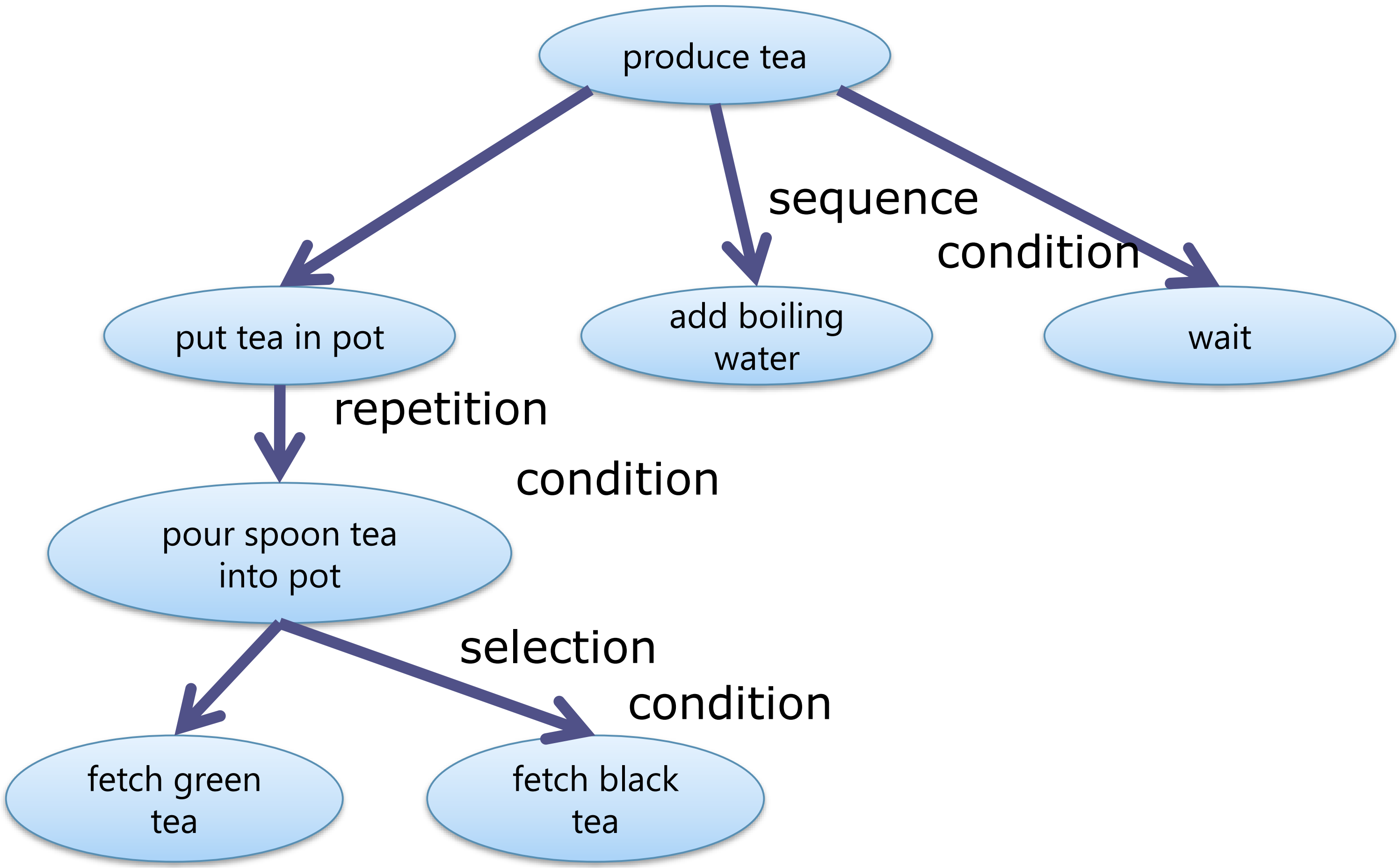
Deposit

Withdraw

# 1.2. Algorithmic decomposition

- Break a process down into well-defined steps
- Case 1: Each step corresponds to an action
  - What are the actions of my software process?
  - What are the states on which the action is performed?
  - actions are grouped to modules/components
- Case 2: Each step corresponds to a couple event-action
  - Rule-based
  - What are the events that may occur?
  - How does my software react on them?

# Example: abstract machine



# Working: Banking system

Transfer

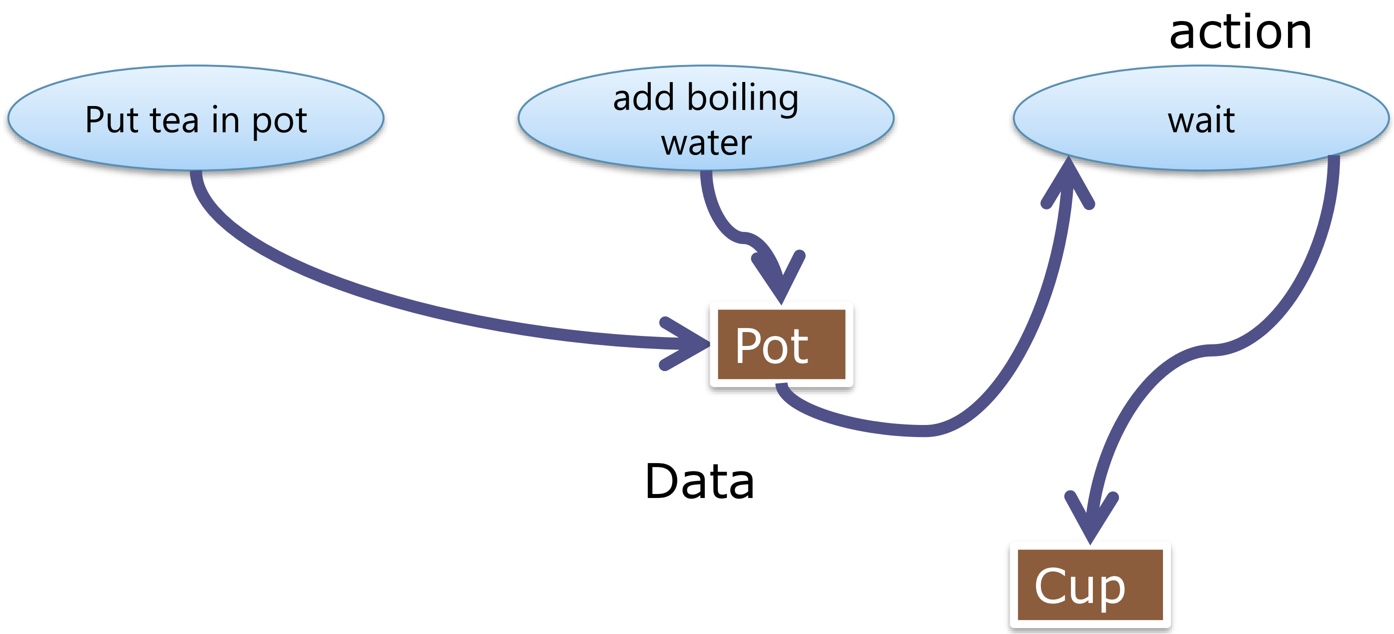
Deposit

Withdraw

# 1.3. Object-oriented decomposition

- Breaks a large system down into progressively smaller objects (sub systems) that are responsible for some part of the problem domain
- Data and actions are developed together
- What are the objects and their associated actions/functions?
  - Case 1: Data-oriented:
    - The algorithm is isomorphic to the data
  - Case 2: Structure-oriented:
    - What are the components of the system and their relations

# Example: abstract machine



# Exercise: Banking system

Transfer

Deposit

Withdraw

# 2. Composition: bottom-up

- Given an abstract machine implementing the partial solution, stepwise construction includes:
  – assemble more complex operations of a higher-level abstract machine
  – assemble the more complex data structures
- Good:
  – always realistic
  – a running partial solution
- Bad:
  – Design might become clumsy since global picture was not taken into account

# 3. Stepwise refinement: top-down

- Iterative process
- Operations and their associated data in every step of an algorithm are the keys to solve a problem
- Given an abstract machine implementing the whole solution, refining the solution includes:
  - Operation refinement: refine the operations in every step of the corresponding algorithm
  - Data refinement: refine the structures of data in every step of the corresponding algorithm

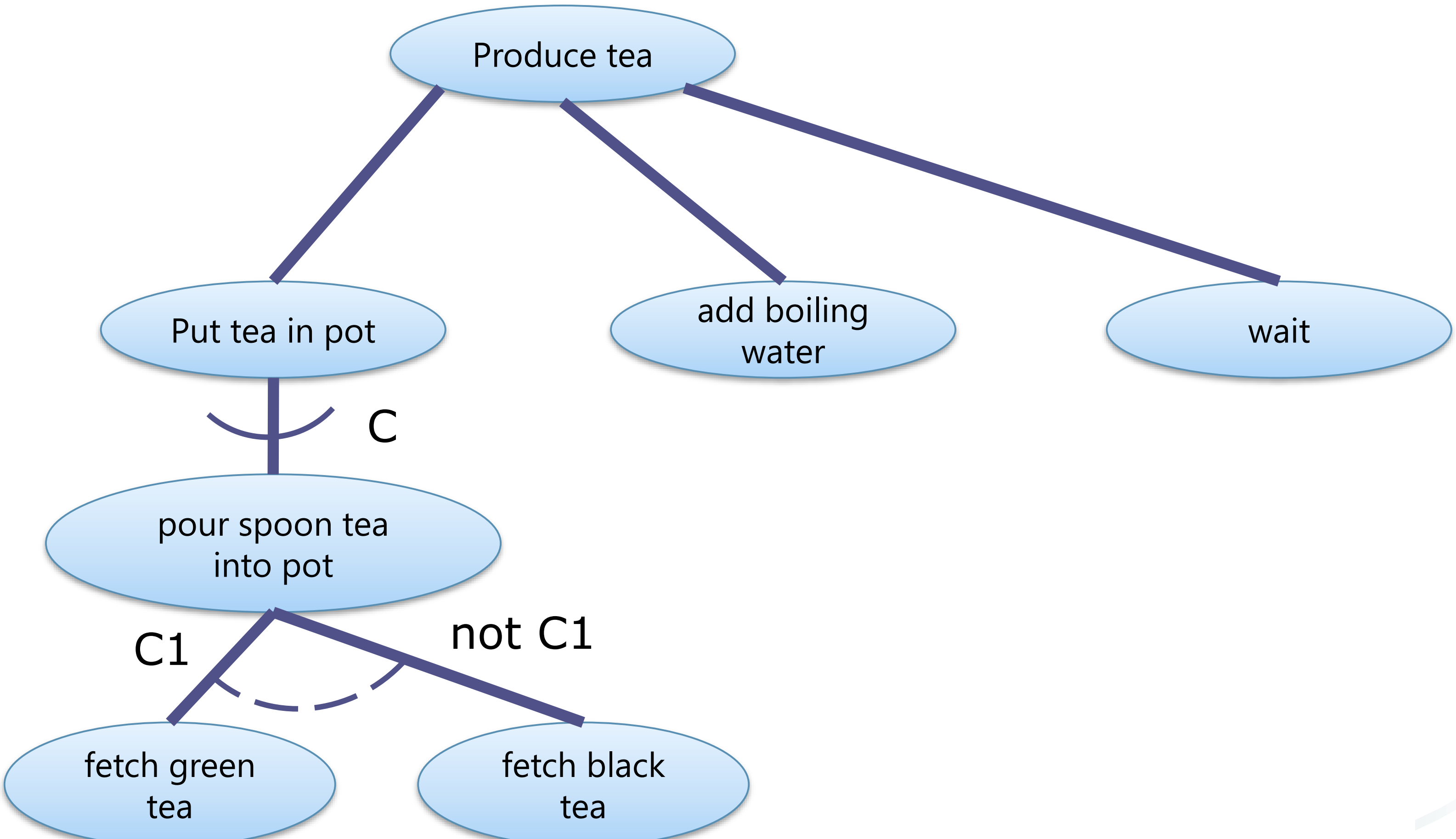
# 2. Stepwise refinement: top-down

- Iterative process:
  - Start: overall description of the problem to be solved (top function)
  - Decompose (in a recursive manner) the problem into sub-problems according to control structures
  - Terminate: when each sub-problems is easy to express in terms of a few lines of code in the chosen programming language.
- Bad:
  - we might never reach a realization
  - often “warehouse solutions” are developed

# Decomposition tree

- Stepwise refinement process may be depicted by a decomposition tree
  - root labeled by name of top problem
  - sub-problem nodes labeled as children of parent node corresponding to problem
  - children from left to right represent sequential order of execution
  - if and while nodes denoted by suitable decoration

# Example



Produce tea
Put tea in pot
add boiling water
wait
C
pour spoon tea into pot
C1
not C1
fetch green tea
fetch black tea

# Exercise: Banking system

# II. MODULARIZATION TECHNIQUES

1. **Module**
2. Relations between modules
3. Specific techniques for design for change

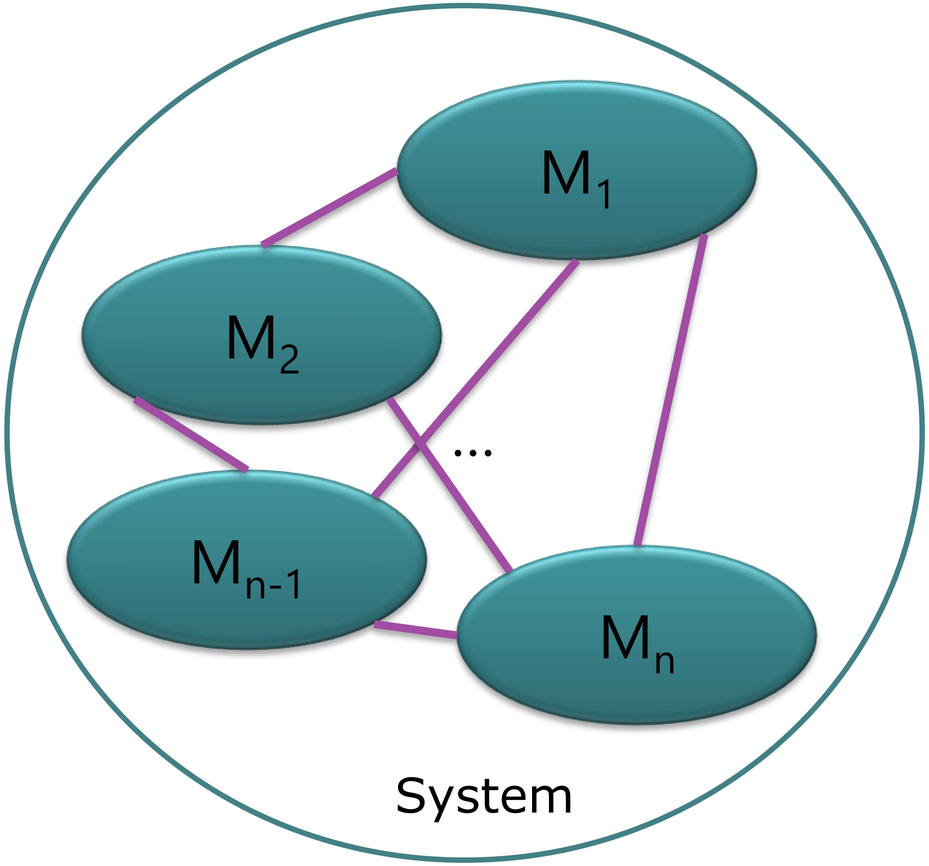
# 1.1. What is a module ?

- Software system should, according to the divide-and-conquer principle, also physically be divided into basic parts, modules
  - 1 module can be developed independently
    - errors can be traced down to modules
    - modules can be tested before assembling

    But not normally be considered as neither a separate system nor an independent system
  - 1 module can be exchanged independently
  - 1 module can provide a set of computational elements to other modules
  - modules can be reused
- The terms module and component are pretty much the same: a module is a well-defined component of a software system
  - a programming-language supported component (often composite)
  - a simple component

# Questions

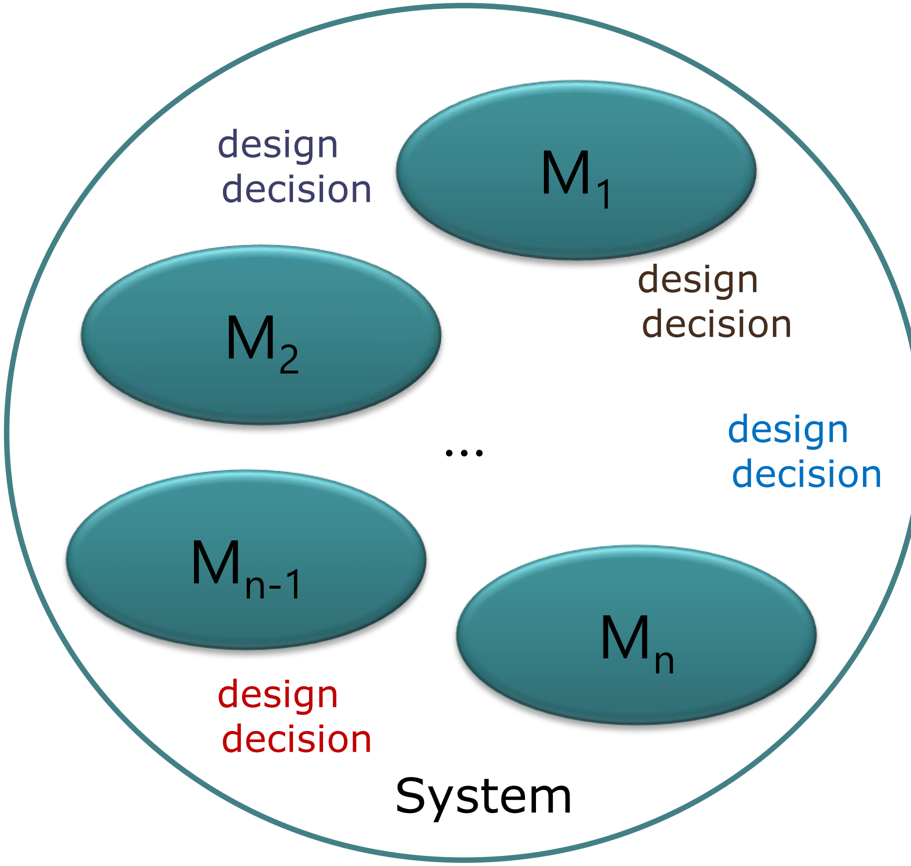
- How to define the structure of a modular system?
- What are the desirable properties of that structure?



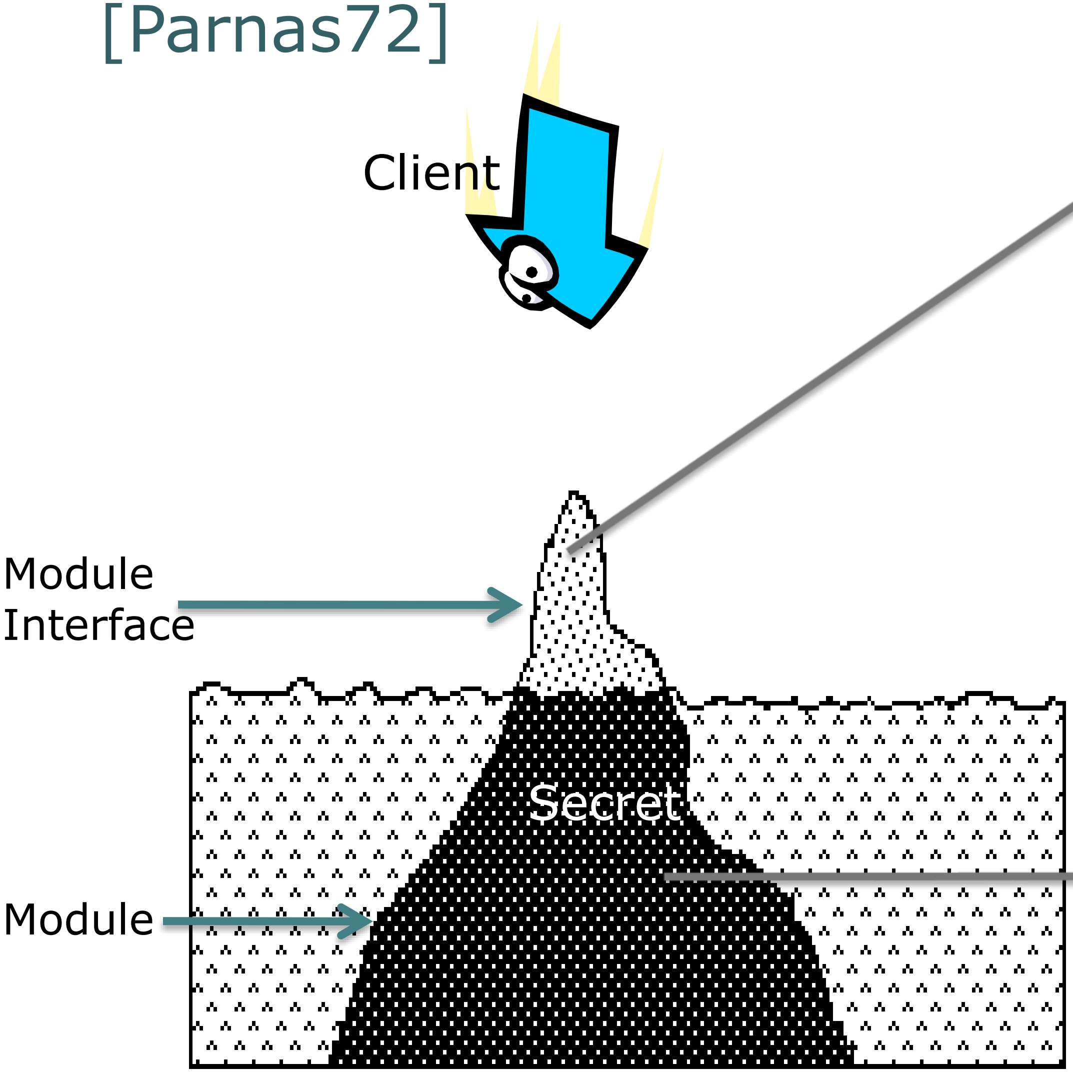
# 1.2. Principle of modularization [Parnas72]

- Fix all design decisions that are likely to change
- Attach each of those decisions to a new module
- The design decision is now called module secret
- Design **module interface** which does not change if module secret changes

→ This principle (that modules hide design decisions) is also called **information hiding**



# 1.2. Principle of modularization
[Parnas72]



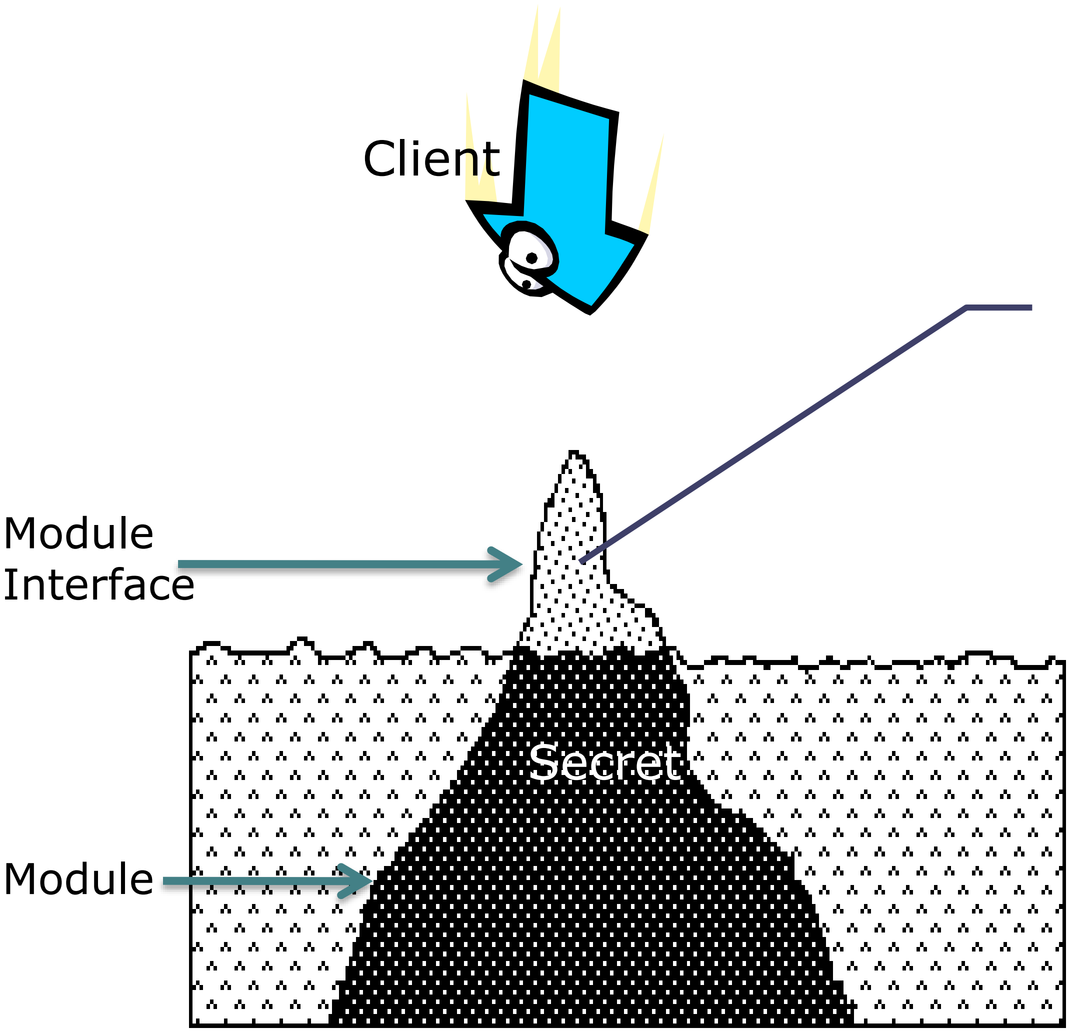

Resources to be exported: types; variables, attributes, functions, events, exceptions, etc.

```
interface Bicycle {
   void changeCadence (int newValue);
   void changeGear(int newValue);
   void speedUp(int increment);
   void applyBrakes(int decrement);
}
```

Implementation of exported resources

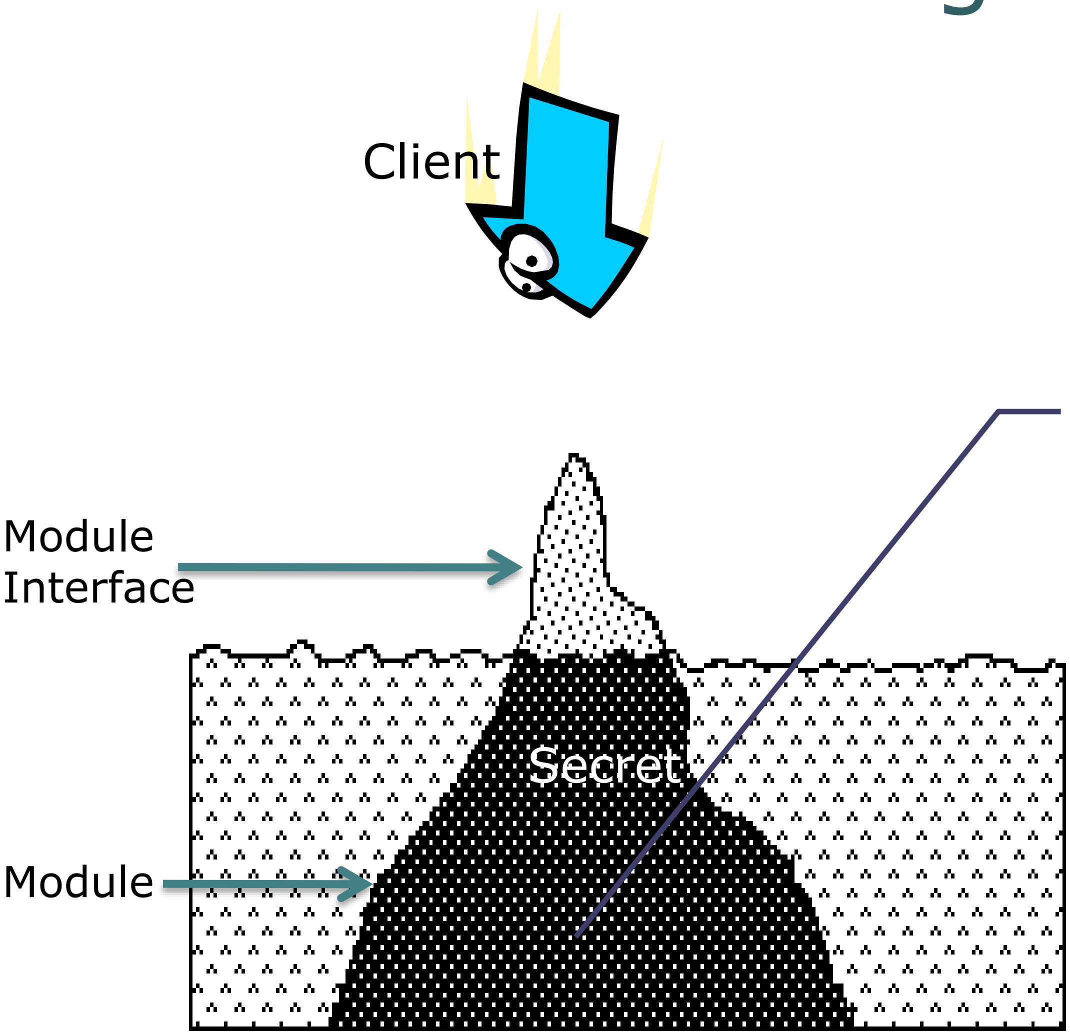
```
class Bike implements Bicycle {
}
class Motor-Bike implements Bicycle {
}
```

# Interface




Act as a firewall preventing access to hidden parts
Should never change
Should not reveal what we expect may change later
Should not reveal unnecessary details
Should consist only of functions
access to data is efficient, but cannot easily be exchanged
e.g., set/get methods for fields of objects
Should specify what is
provided (exported)
required (imported)
many languages support only the former

# Information hiding




Possible module secrets:
implementation
how the algorithm works
data formats
representation of data structures, states
user interfaces (e.g., AWT)
texts (language e.g., get text library)
ordering of processing (Design Pattern Strategy, Visitor)
location of computation in a distributed system

# II. MODULARIZATION TECHNIQUES

1. Module
2. **Relations between modules**
3. Specific techniques for design for change

# 2.1. Definition

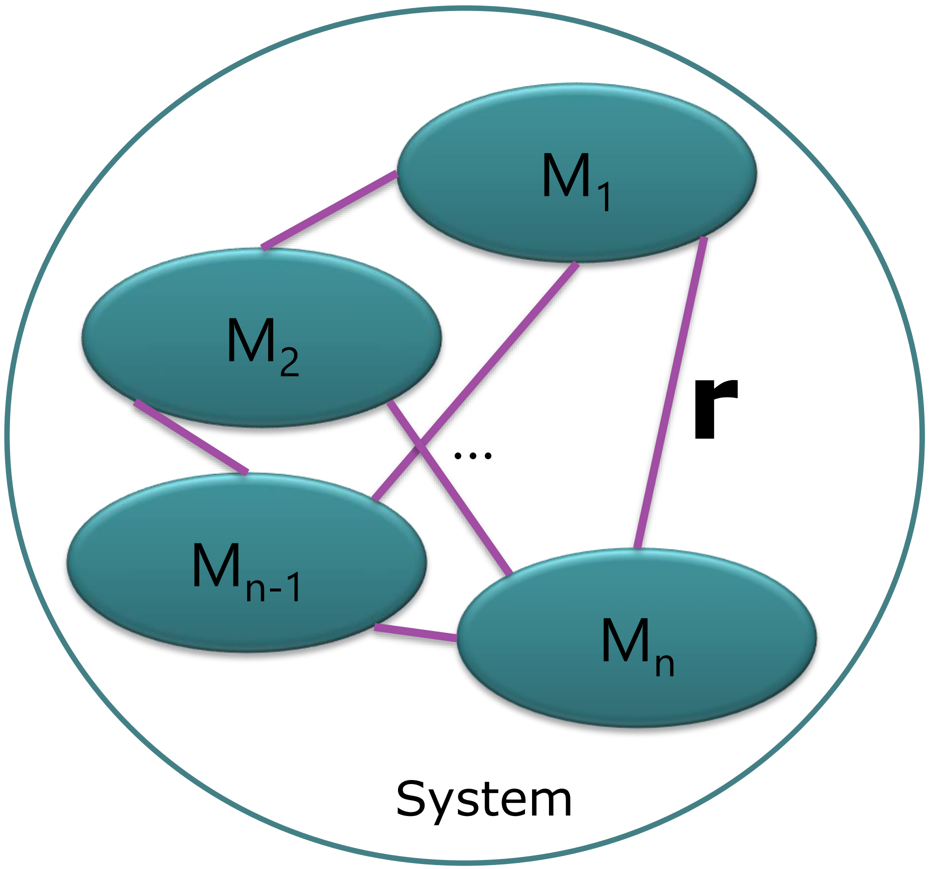
- Let S be a set of modules

  $S = \{M_1, M_2, . ., M_n\}$
- A binary relation **r** on S is a subset of

  S **x** S
- If $M_i$ and $M_j \in S$,

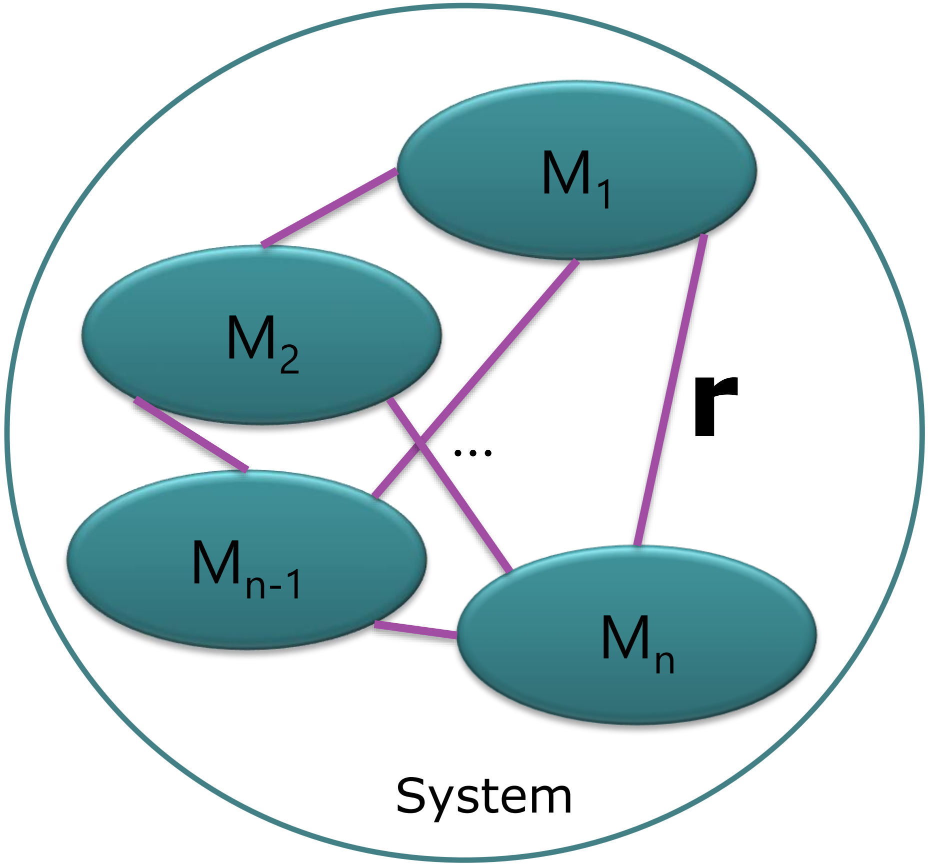
  a pair $<M_i, M_j> \in r$ can be written as

  $M_i$ **r** $M_j$



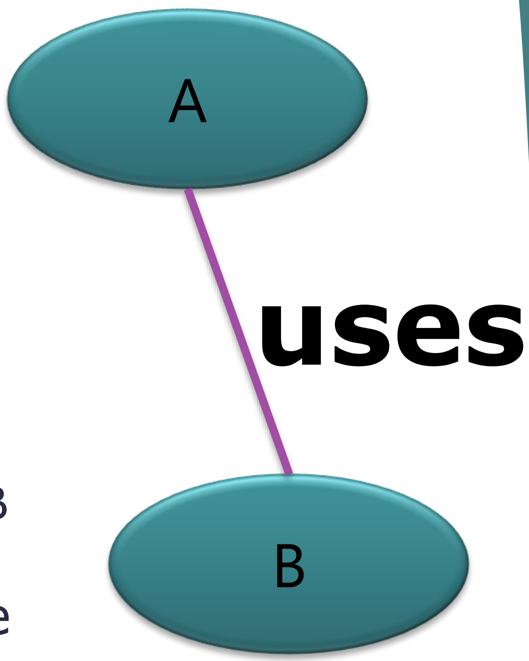
# 2.2. Classification

- delegates-to (delegation)
- is-a (inheritance)
- has-a (aggregation)
- is-component-of, comprises (composition)
- accesses-a (access relation)
- is-privileged-to, owns-a (security)
- calls
- is-called-by
- **relies-on (uses)**



# 2.3. USES relation: A uses B

- Module A uses (relies-on) module B iff A requires a correct implementation of B for its own correct execution.
  - It is "statically" defined
  - A is a client of B; B is a server
- Requires an implementation may mean:
  - A delegates work to B (delegation relation)
  - A accesses a variable of B through the B's interface (access relation)
  - A depends on B to provide its functions: A calls B OR B calls A
- The uses relation may be a partial order (a tree or a dag) or a total order, then the system is called hierarchical or layered
- USES should be a hierarchy to make the software easier to understand, build and test



# Example:
# modular representation with OOP

## USES

```
class B
{
    …
    …
}

class A
{
    …
    …
    void test() {
        B obj = new B();
        ….
    }
}
```

## HAS

```
class B
{
    …
    …
}

class A
{
    …
    …

    B obj = new B();
        ….
}
```

## IS

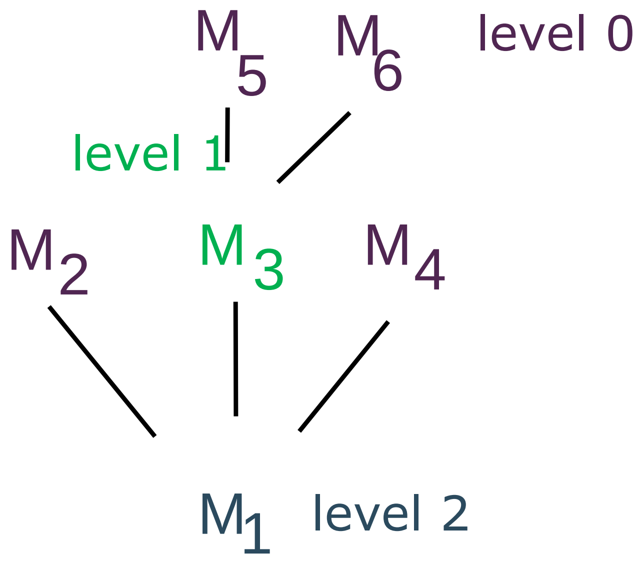
```
class B
{
    …
    …
}

class A extends B
{
    …
    …

}
```

# 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** $M_j$
  - 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



# Discussion: USE relation

## Delegation

```
class B
{
	methodB(){…}
	…
}

class A
{
	B obj = new B ();
	methodA() {
		obj.methodB();
	}	….
}
```

## With hierarchy

```
class Employee {
	float salary = 30000;
}
class Developer extends Employee {
	float bonus = 2000;
	public static void main(String args[]) {
		Employee obj = new Employee();
		System.out.println
			("Salary is:"+obj.salary);
	}
}
```

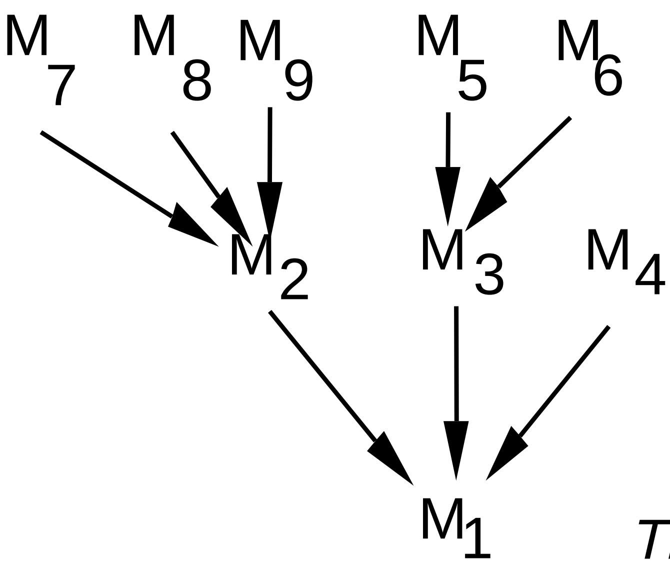
# 2.4. IS-COMPONENT-OF relation

- Used to describe a higher level module as constituted by a number of lower level modules

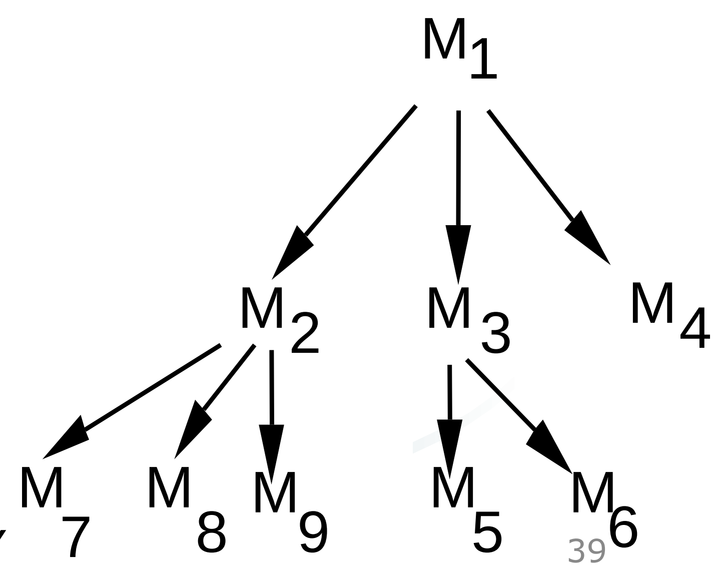
A IS-COMPONENT-OF B

B consists of several modules, of which one is A

B COMPRISES A

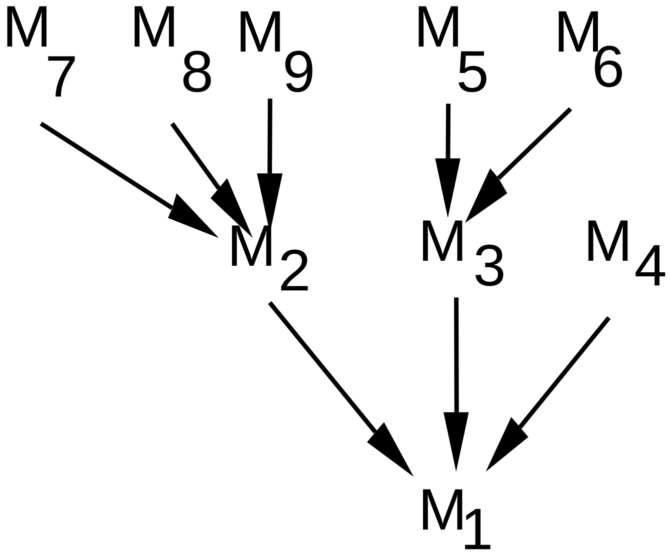



*They are a hierarchy*



# IS-COMPONENT-OF relation

- Used to describe a higher level module as constituted by a number of lower level modules
- If $M_{S,i}=\{M_k | M_k \in S \wedge M_k$ IS-COMPONENT-OF $M_i\}$ we say that $M_{S,i}$ IMPLEMENTS $M_i$

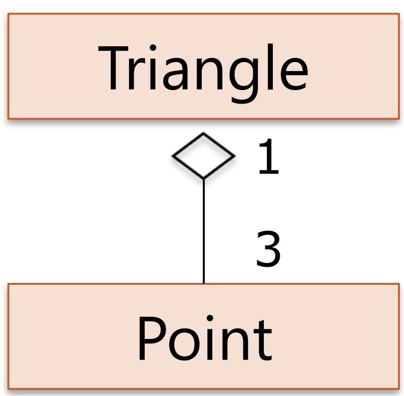



$M_{S,2}=\{M_7, M_8, M_9\}$ IMPLEMENTS $M_2$

# Example: OO relations

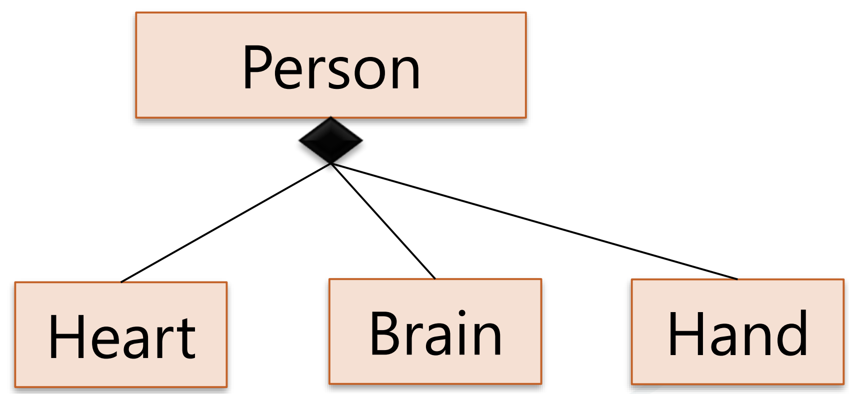
- Aggregation:
  - Defines objects or object classes as composed of simpler components that constitute the parts of other objects.
  - This is a PART-OF relation (differs from IS-COMPOSED-OF)

**Cardinality of the constituents**




Triangle USES Point
Point IS-A-PART-OF Triangle
Not Triangle IS-COMPOSED-OF Point

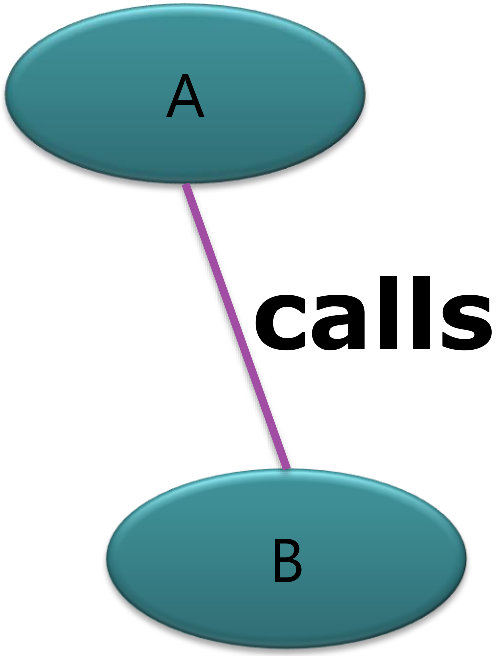
**Behaviors of the constituents**




Person HAS Heart, Brain, Hand
Person IS-COMPOSED-OF Heart, Brain, Hand

# 2.5. CALLS relation

- A allocates an instance of B
- A calls B by exception or event (not by other resources such as types, functions, procedures, variables, etc.)




```
public class className {
      public void deposit(double amount) throws RemoteException {
           // Method implementation
           throw new RemoteException();
       }
       //Remainder of class definition
}
```

# II. MODULARIZATION TECHNIQUES

1. Module
2. Relations between modules
3. **Specific techniques for design for change**

# 3.1. Constant configuration

- Factoring constant values into symbolic constants is a common implementation practice
- Example:

In C:

```
#define MaxSpeed 5600;
```

In Python:

```
class _const:
    class ConstError(TypeError): pass
    def _ _setattr_ _(self, name, value):
        if self._ _dict_ _.has_key(name):
            raise self.ConstError, "Can't rebind const(%s)"%name
        self._ _dict_ _[name] = value
    def _ _delattr_ _(self, name):
        if self._ _dict_ _.has_key(name):
            raise self.ConstError, "Can't unbind const(%s)"%name
        raise NameError, name

import sys
sys.modules[_ _name_ _] = _const(  )
```

# 3.2. Conditional compilation

- Have several versions of a (part of) program in the same source file.
- Compile the program accordingly to
  - Its state at runtime
  - Target platform
  - Ambitious testing
  - ..
- Example: C pre-processor

```
//...source fragment common to all versions...
#ifdef hardware-1
    //...source fragment for hardware 1 ...
#endif
#ifdef hardware-2
    //...source fragment for hardware 2 ...
#endif
```

# 3.3. Software generation

- Example: interface prototyping tools
  - Once an interface is defined, implementation can be done
    - first quickly but inefficiently
    - then progressively turned into the final version
  - Initial version acts as a prototype that evolves into the final product

# III. SOFTWARE ARCHITECTURES

1. Definition
2. Common architectures
3. Domain specific architectures

# 1.1. What is software architecture ?

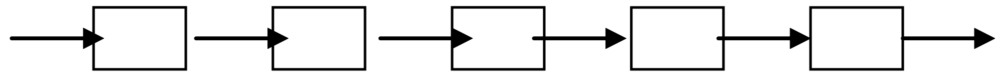
- Shows
  - gross structure of the system to be defined
  - organization of the system to be defined
- Describes
  - main components of a system
  - external visible properties of those components
  - relationships among those components
  - rationale for decomposition into its components
  - constraints that must be respected by any design of the components
- Guides
  - the development of the design

# 1.2. What are software components?

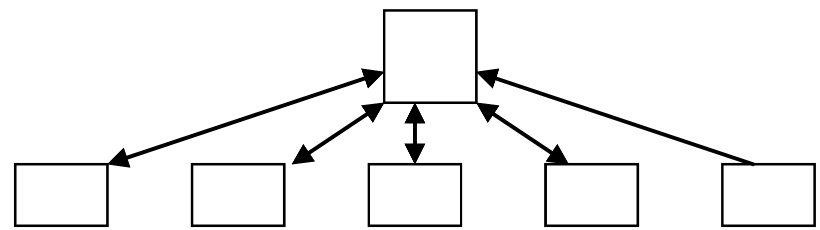
- Goal
  - build systems out of pre-existing libraries of components
  - as most mature engineering areas do
- Examples
  - STL for C++
  - JavaBeans and Swing for Java

# 2. Common architectures

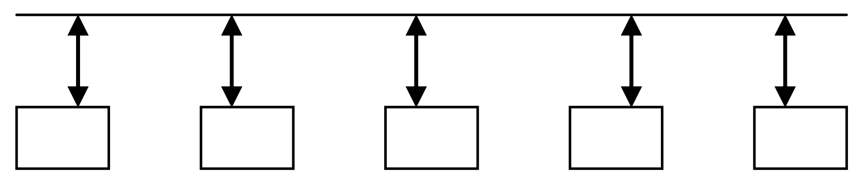
- Pipes (data flow) and filters (operations): A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order.
- Blackboard (shared data structure): Data store can contact and trigger actions in components when data becomes available.
- Event-based control (implicit invocation): Each component can respond to externally generated events from other components or the system's environment.



Pipes and filters



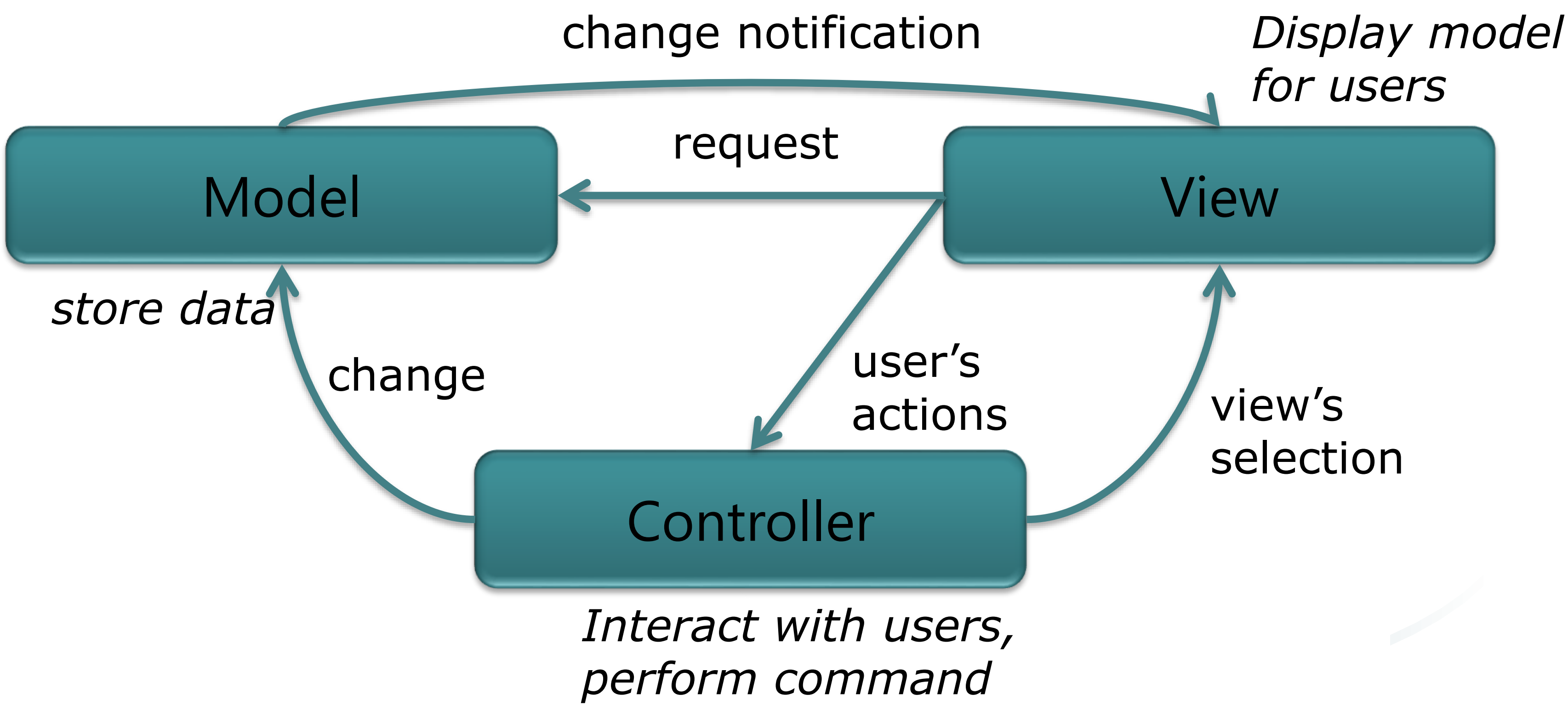
Blackboard



Event based control

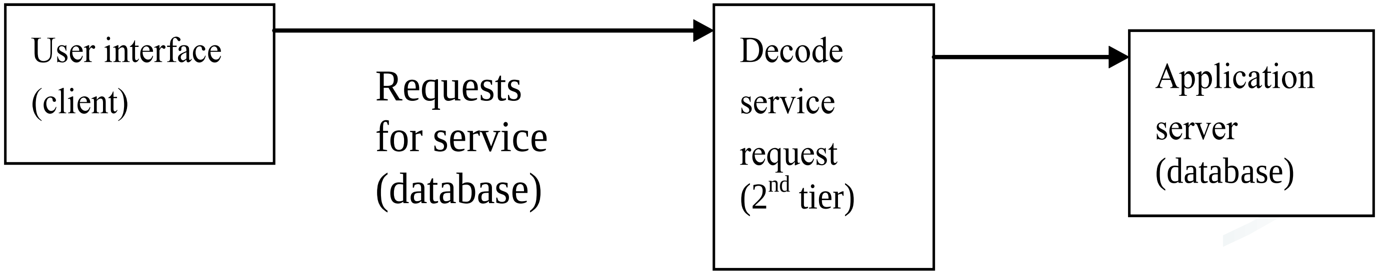
# 3. Domain-specific architectures

- MVC architecture:
  - Significant amount of user interaction



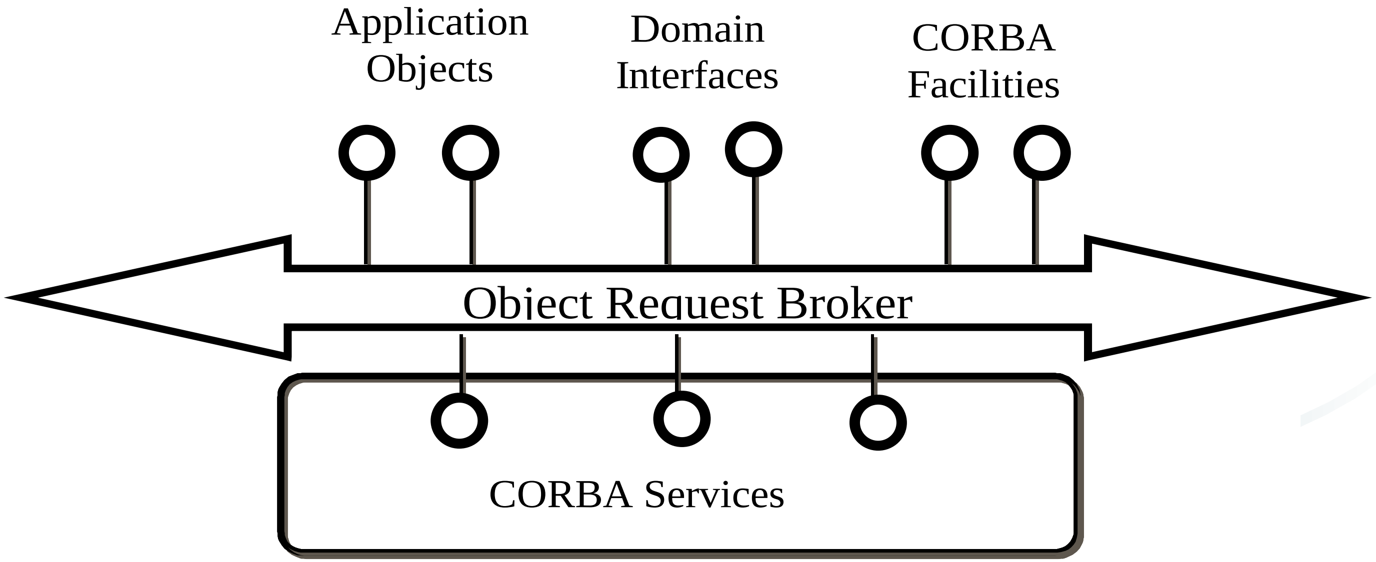
# Domain-specific architectures

- Distributed system architectures:
  - From two tiered
    - Client-server
  - to three tiered

# 4. Component integration

- The CORBA (Common Object Request Broker Architecture) Middleware
- Clients and servers connected via an Object Request Broker (ORB)
- Interfaces provided by servers defined by an Interface Definition Language (IDL)
- In the Microsoft world: DCOM (Distributed Component Object Model)



# Quiz and Exercises

- Now let’s go over what you have learned through this lesson by taking a quiz.
- When you’re ready, press Start button to take the quiz