Lecture 11.09

# Object-Oriented Design & Standard Template Library (2)

SE271 Object-Oriented Programming (2020)

Yeseong Kim

Original slides from Prof. Shin at DGIST

# Short Notice

- Will upload HW4 by Wednesday
  - We will review it today

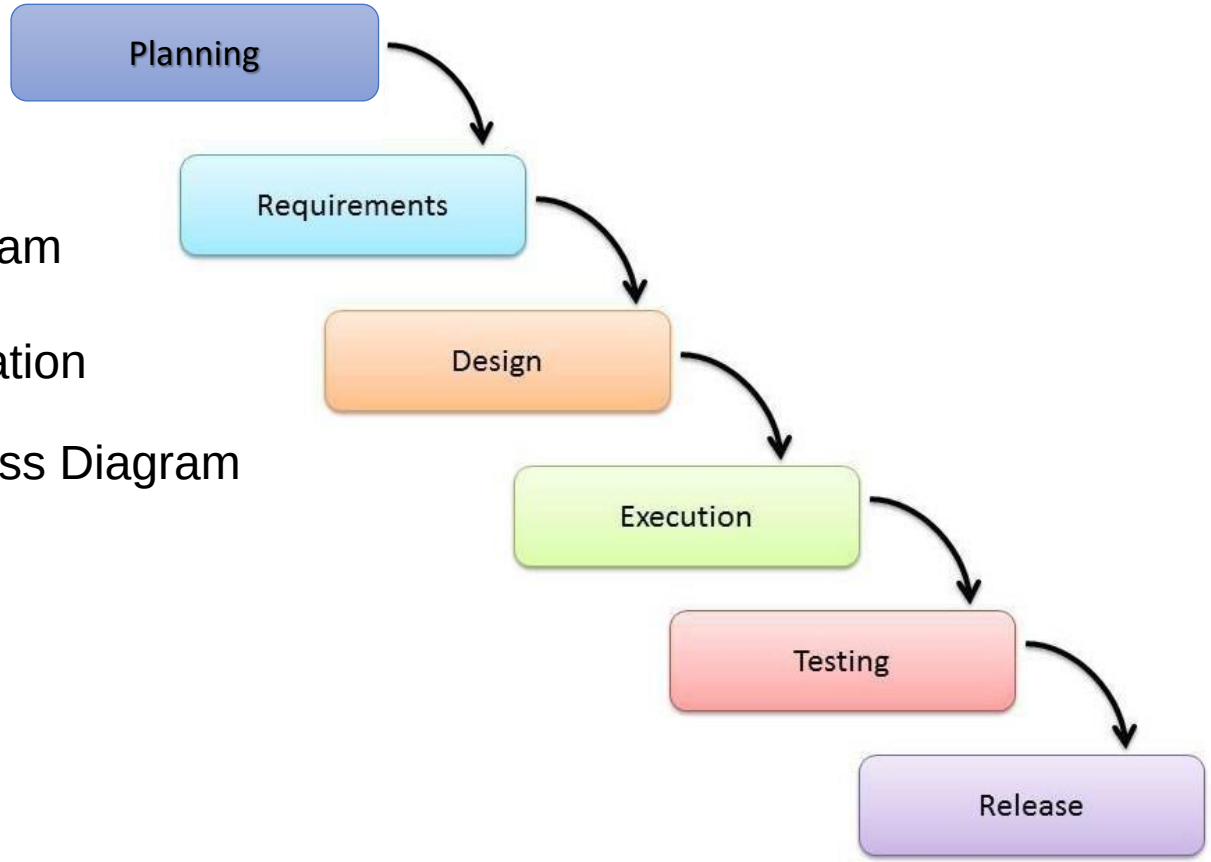
# Software Development Life Cycle

Required Functions

Use case diagram

Specification

Class Diagram



# Object-Oriented Design

- OOP
  - Abstraction
  - Encapsulation
  - Polymorphism
  - Inheritance

- Class Design
  - List up functions the project should provide
  - Break down the functions until each function performs only one job
  - Design class to handle each function
    - Add data(attributes)
    - Add methods
  - Decide relationship between objects and how to communicate (messages)

# Object-Oriented Design

- Purpose of Object-Oriented Design
  - Easy maintenance
  - High understanding
  - Reusable codes
  - Easy to change if requirement changes
  - Better Performance
  - …

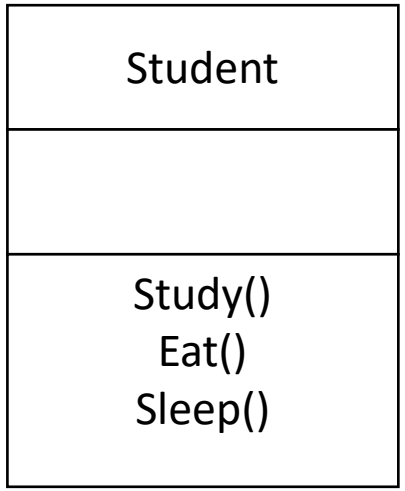
# 5 Design Principles (SOLID)

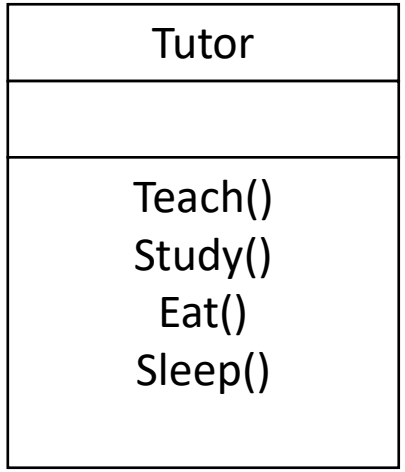
- **S**RP : Single responsibility principle
- **O**CP : Open-Closed principle
- **L**SP : Liskov Substitution Principle
- **I**SP : Interface Segregation Principle
- **D**IP : Dependency Inversion Principle

# 5 Design Principles (SOLID)

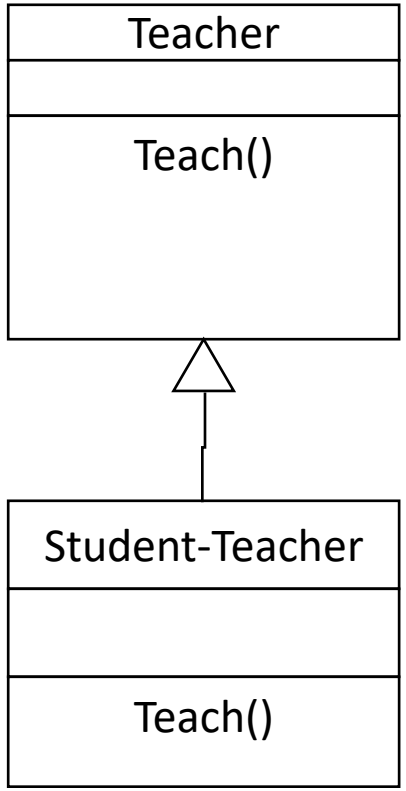
- **S**RP : Single responsibility principle
  - Each class has a single responsibility → modification by only one reason

- Issue:



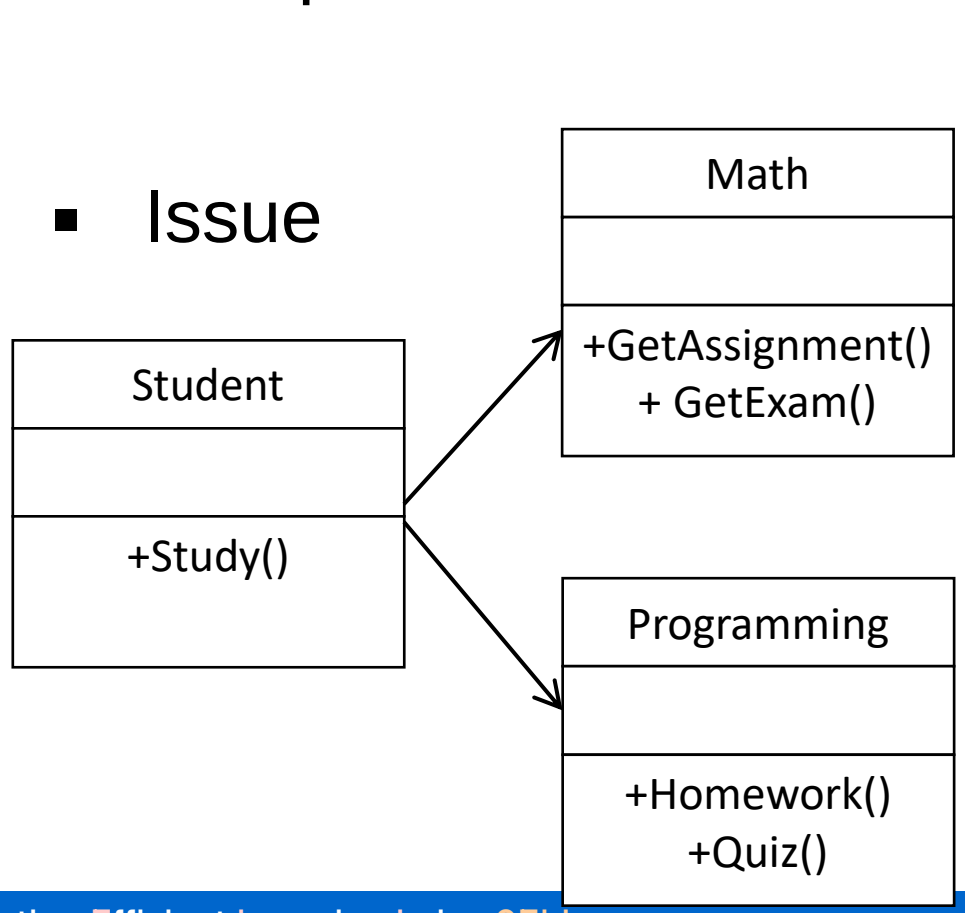

**vs.**

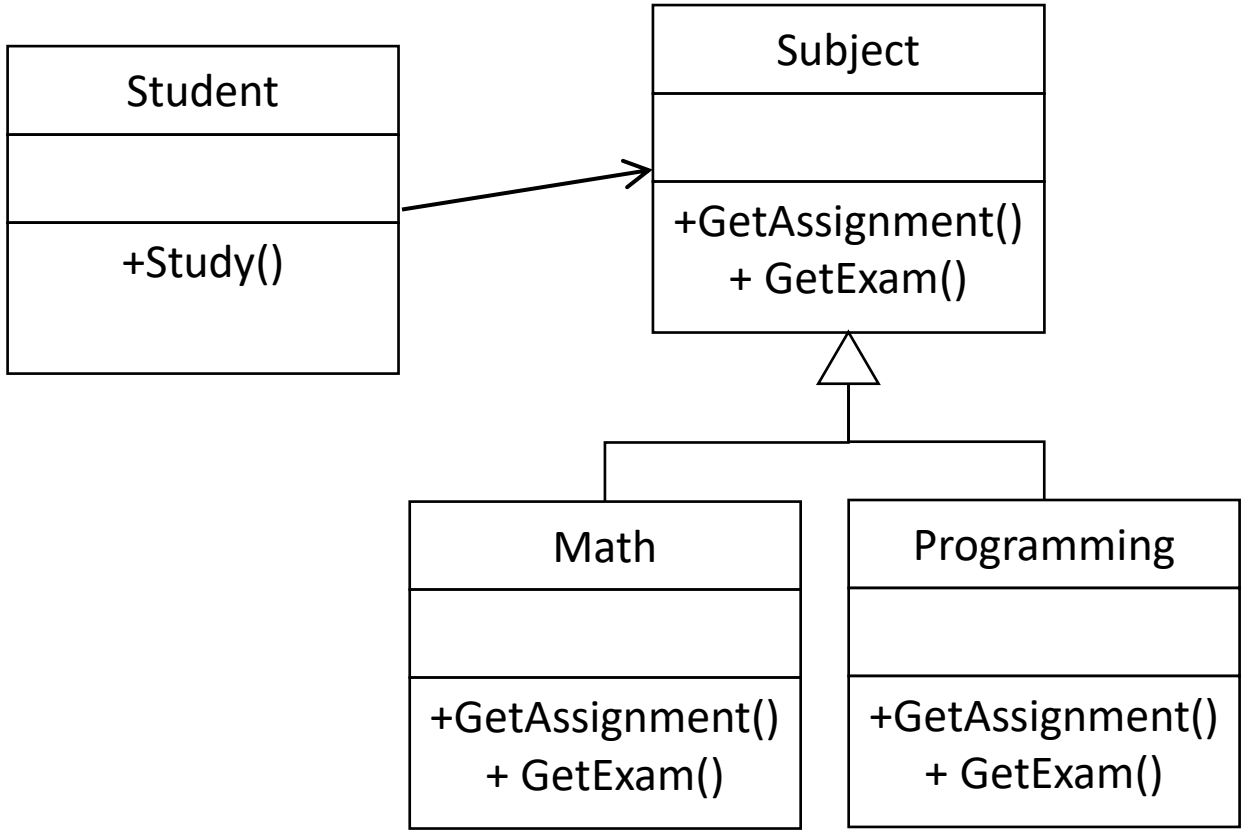


# 5 Design Principles (SOLID)

- **O**CP : Open-Closed principle
  - Open for the extension, closed for the modification

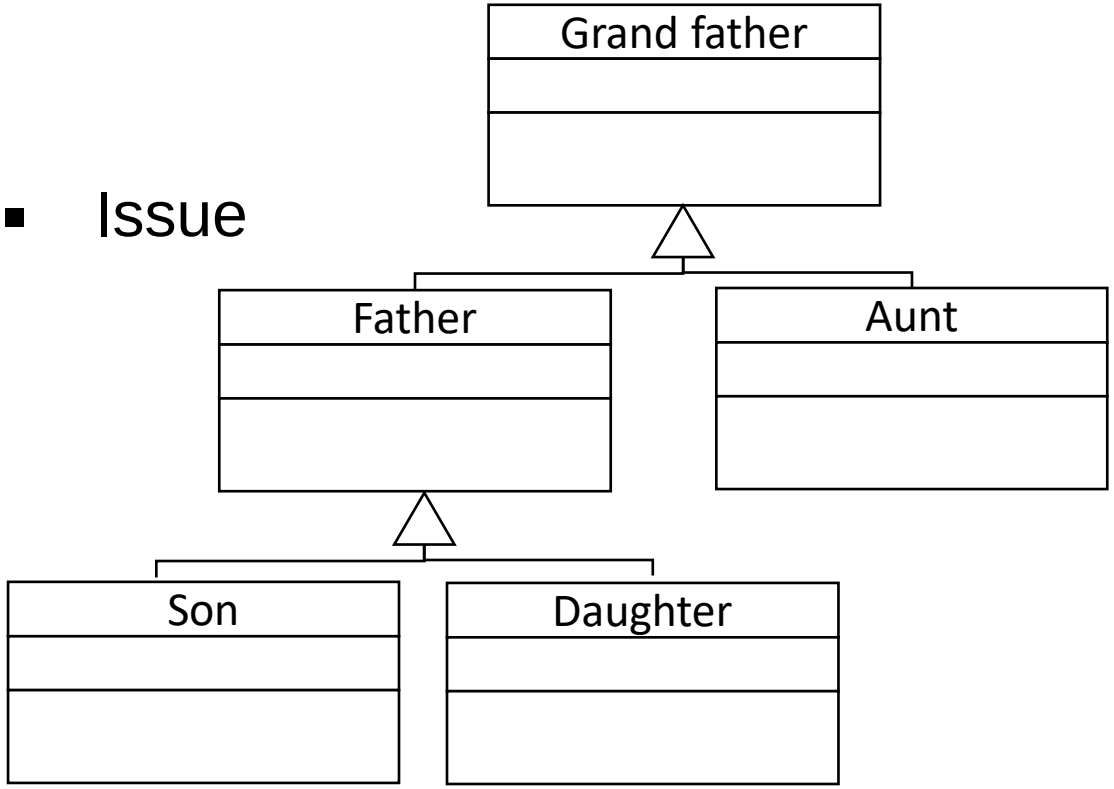
- Issue

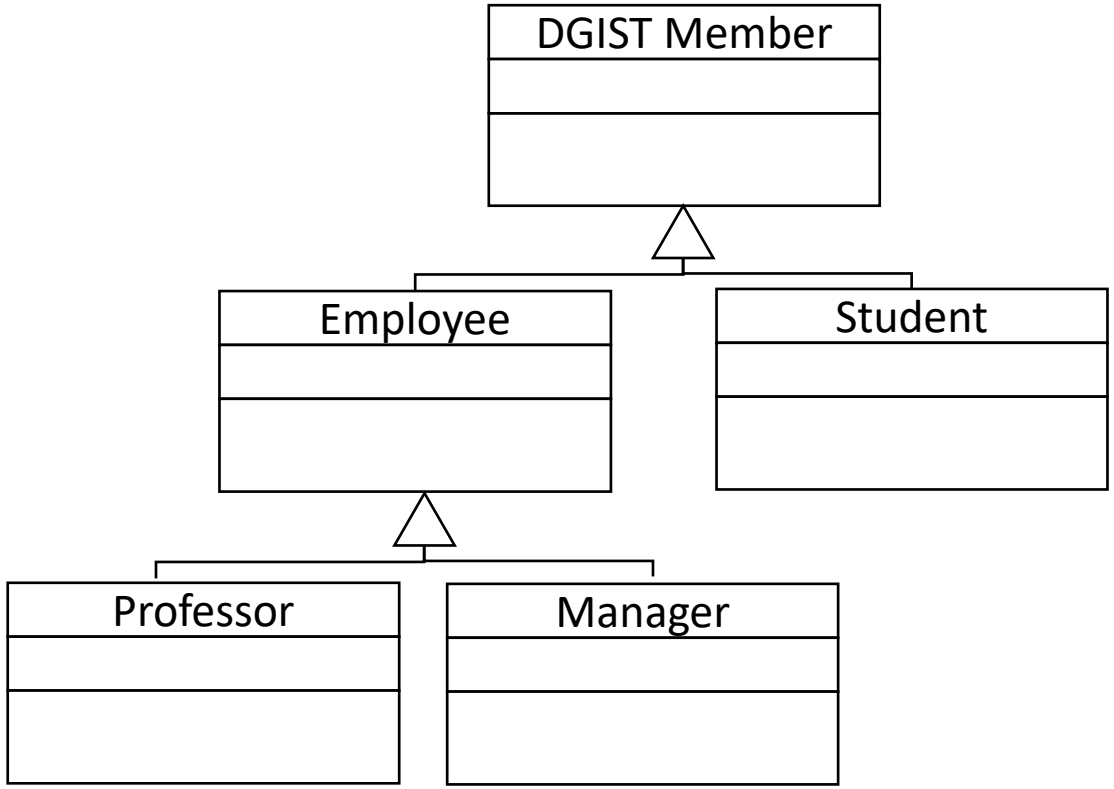


# 5 Design Principles (SOLID)

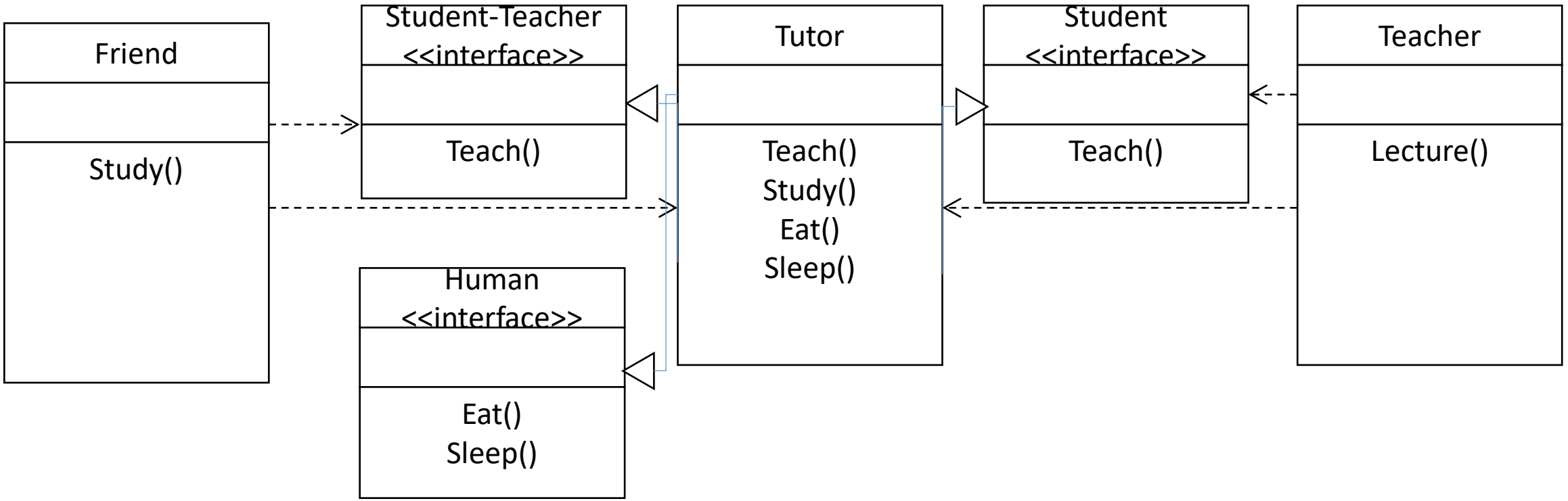
- **L**SP : Liskov Substitution Principle
  - Derived classes can be substituted by its Base class
  - Minimalize use of override
- Issue



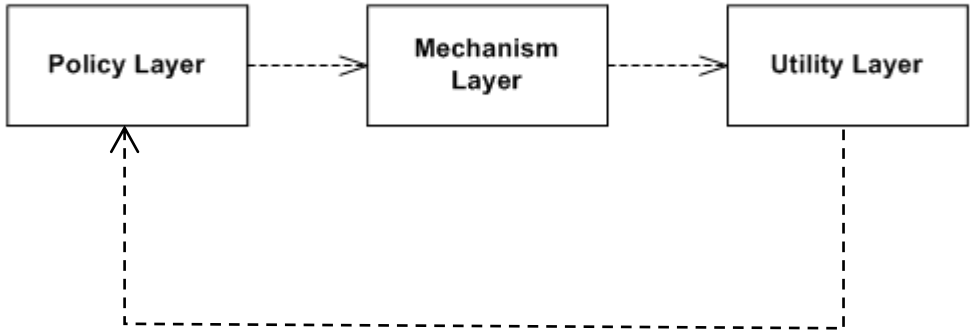
# 5 Design Principles (SOLID)

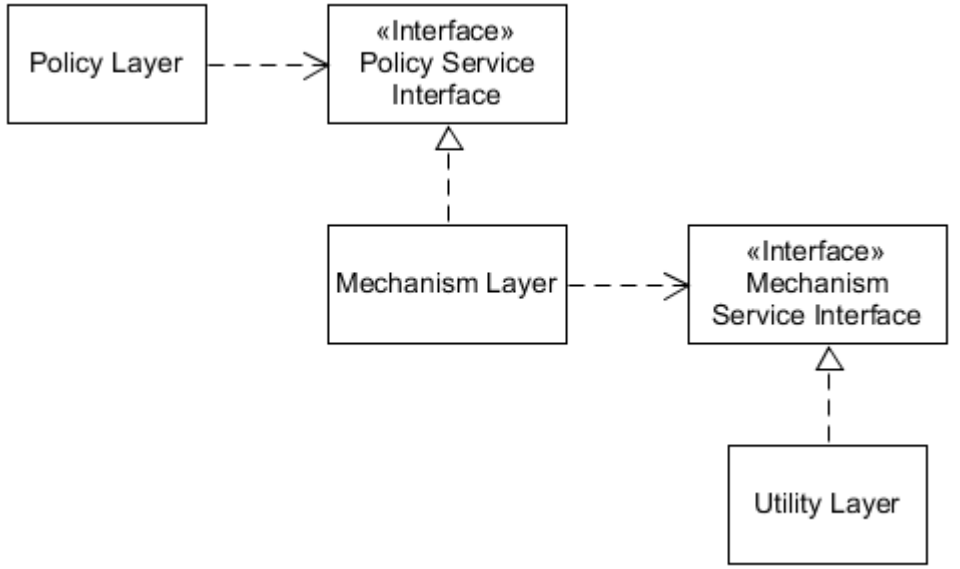
- ISP : Interface Segregation Principle
  - No client should be forced to depend on methods it does not use



# 5 Design Principles (SOLID)

- **D**IP : Dependency Inversion Principle
  - High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).
  - Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.



# Example: HW2

# Example: HW4

# Today’s Topic

- Containers
  - Set
  - Map

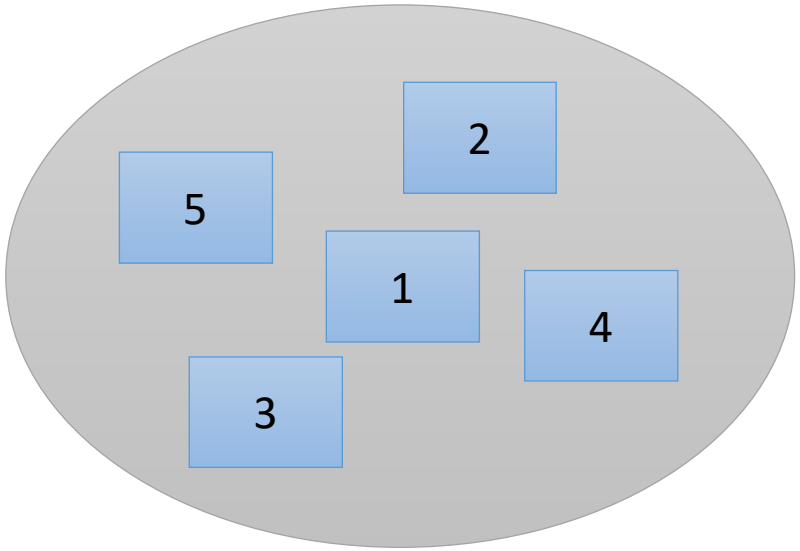
# Associative Containers

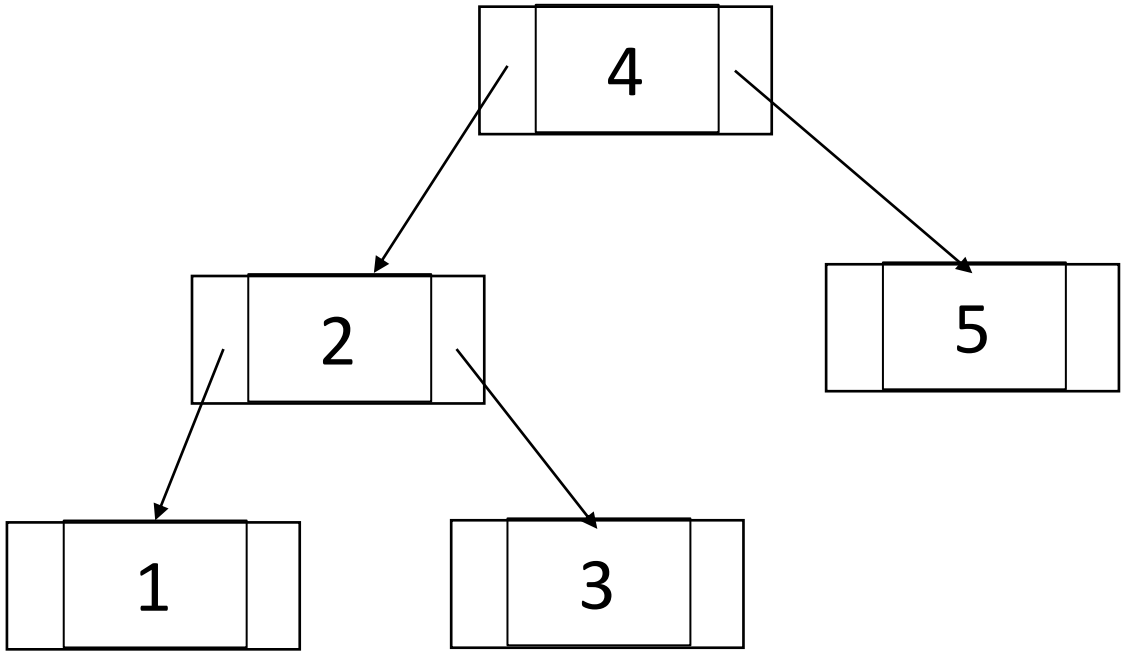
- Set

Node-based containers

- Map

data is stored and ordered
not by input sequence



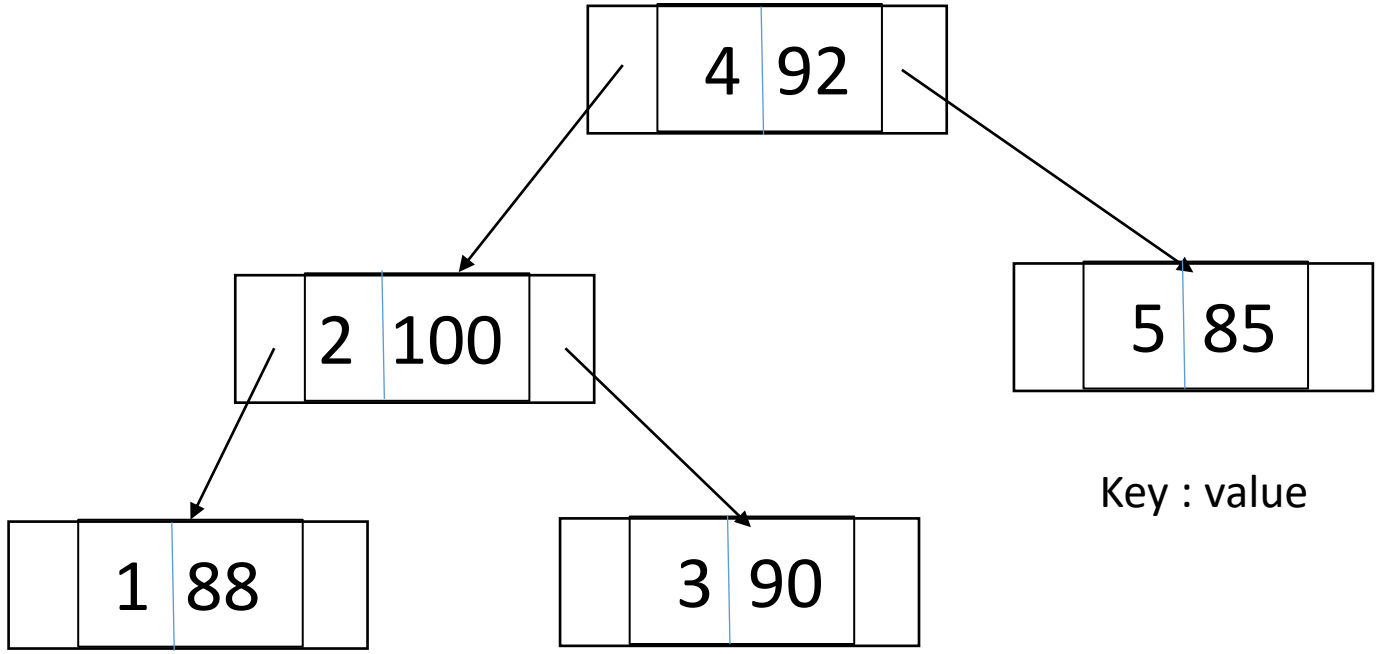
# Associative Containers

- Set

Node-based containers

- Map

data is stored and ordered
not by input sequence



# Example: Map

```
template<typename Key,
         typename T,
    typename Compare = std::less<key>,
typename Allocator =
std::allocator<std::pair<const Key, T>
>
class map;
```

```
#include<iostream>
#include<map>
using namespace std;
int main() {
    map<string, int> m;
    m.insert(pair<string, int>("Bob", 20));
    m.insert(pair<string, int>("Alice", 22));
    m.insert(pair<string, int>("Carol", 21));
cout << m["Bob"]  << endl;
for (pair<string, int> p : m)
    cout << p.first << ": " << p.second<< endl;
for (map<string, int>::iterator it = m.begin(); it != m.end(); ++it)
    cout << it->first << ": " << it->second << endl;
}
```

# Common operators for STL containers

| Function | Description |
|---|---|
| `T()` | create empty container (default constructor) |
| `T(const T&)` | copy container (copy constructor) |
| `T(T&&)` | move container (move constructor) |
| `~T()` | destroy container (including its elements) |
| `empty()` | test if container empty |
| `size()` | get number of elements in container |
| `push_back()` | insert an element at end of container (sequential) |
| `insert()` | insert an element (associative/unordered) |
| `clear()` | remove all elements from container |
| `operator=()` | assign all elements of one container to other |
| `operator[]()` | access element in container |

Lecture note 2017

# Performance

| Container | Insertion | Access | Erase | Find |
|---|---|---|---|---|
| vector / string | Back: O(1) or O(n)<br>Other: O(n) | O(1) | Back: O(1)<br>Other: O(n) | Sorted: O(log n)<br>Other: O(n) |
| deque | Back/Front: O(1)<br>Other: O(n) | O(1) | Back/Front: O(1)<br>Other: O(n) | Sorted: O(log n)<br>Other: O(n) |
| list /<br>forward_list | Back/Front: O(1)<br>With iterator: O(1)<br>Index: O(n) | Back/Front: O(1)<br>With iterator: O(1)<br>Index: O(n) | Back/Front: O(1)<br>With iterator: O(1)<br>Index: O(n) | O(n) |
| set / map | O(log n) | - | O(log n) | O(log n) |
| unordered_set /<br>unordered_map | O(1) or O(n) | O(1) or O(n) | O(1) or O(n) | O(1) or O(n) |
| priority_queue | O(log n) | O(1) | O(log n) | - |

https://john-ahlgren.blogspot.com/2013/10/stl-container-performance.html

# References

- Learn C++ (https://www.learncpp.com/)
  - STL: Ch. 16

- STL
  - https://en.cppreference.com/w/cpp/algorithm


# ANY QUESTIONS?