DGIST

Lecture 12.02

Much, More, Potpourri

SE271 Object-Oriented Programming (2020)

Yeseong Kim

Short Notice

- The due of HW4 & project presentation video is this Sunday
  - We will run a script to decide who will do the demo on Monday and Wednesday

- 기말고사: 12월 16일 (수)
  - E1, 컨벤션홀 A

Today’s Topic

- Smart Pointer: weak_ptr review
- Lambda
- Design pattern
- UML
- OOD, again

Using a **weak_ptr**

usingweak.cc

```cpp
#include <cstdlib>   // for EXIT_SUCCESS
#include <iostream>  // for std::cout, std::endl
#include <memory>    // for std::shared_ptr, std::weak_ptr

int main(int argc, char **argv) {
  std::weak_ptr<int> w;

  {  // temporary inner scope
    std::shared_ptr<int> x;
    {  // temporary inner-inner scope
      std::shared_ptr<int> y(new int(10));
      w = y;
      x = w.lock();  // returns "promoted" shared_ptr
      std::cout << *x << std::endl;
    }
    std::cout << *x << std::endl;
  }
  std::shared_ptr<int> a = w.lock();
  std::cout << a << std::endl;

  return EXIT_SUCCESS;
}
```

Recall: Function Object (functor)

- Function object
  - Function object is a function-like object
  - Has the same function with status (cf. function)
  - Are able to use it as a template parameter

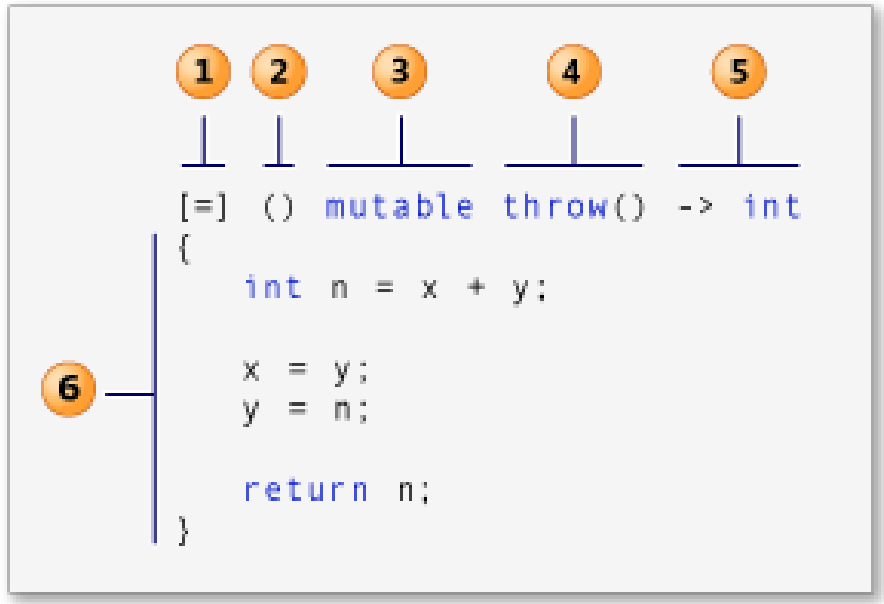
```
template<typename T>
T Plus(T n1, T n2) {
    return n1+n2;
}

cout << Plus<int>(10, 20);
```

```
template<typename T>
class Plus {
public:
    T operator()(T n1, T n2){
        return n1+n2;
    }
};
Plus<int> p;
cout << p.operator()(10, 20);
cout << p(10, 20);
cout << Plus<int>()(10, 20);
```

Lambda Expression: [] () {}

- Supported in C++11 and later
  - A convenient way of defining an anonymous function object
  - Encapsulate a few lines of code that are passed to algorithms or asynchronous methods



1. capture clause (Also known as the *lambda-introducer* in the C++ specification.)
2. parameter list Optional. (Also known as the *lambda declarator*)
3. mutable specification Optional.
4. exception-specification Optional.
5. trailing-return-type Optional.
6. lambda body.

https://docs.microsoft.com/ko-kr/cpp/cpp/lambda-expressions-in-cpp?view=msvc-160

Lambda as an Object

- Treat a lambda function as an object

```
#include <functional>
void print(std::function<bool()> func) {
    std::cout << func() << std::endl;
}

int main() {
    auto lambda_function = []() { return 10; };
    print(lambda_function);
}
```

Lambda Capture

- Capture allows to use variables outside of the lambda function

```
#include <functional>
void print(std::function<bool()> func) {
    std::cout << func() << std::endl;
}

int main() {
    int x = 10;
    auto lambda_function = [=]() { return x; };
    print(lambda_function);
}
```

Change this with
- & (call by reference)
- x
- &x

Simple Lambda Example With STL

```
[CAPTURE](PARAMETERS){ BODY }
```

```
int main(){
    std::vector<int> v1{ 1,2,3,4 }, v2{4,5,6,7};
    // merge
    std::vector<int> dst(v1.size()+v2.size());
    std::merge(v1.begin(), v1.end(), v2.begin(), v2.end(), dst.begin());
    std::for_each(dst.begin(), dst.end(),
        [](auto& v) {
            std::cout << v << std::endl;
        }
    );
}
```

Design Pattern Example: Factory

```
class Unit {
public:
    static Unit *factory(int index);
    virtual void info();
};
class Snake : public Unit {
public:
    void info() { cout << "Snake!" << endl; }
};
class Fruit : public Unit {
public:
    void info() { cout << "Fruit!" << endl; }
};
```

```
Unit* Unit::factory(int index) {
    if (index == 1)
        return new Fruit();
    else if (index == 2)
        return new Snake();
    return NULL;
}
int main() {
    vector<Unit*> data;
    while (true) {
        cout << "1: fruit, 2: snake, 0: END" << endl;
        std::cin >> index;
        if (index == 0) break;
        data.push_back(Unit::factory(index));
    }
}
```

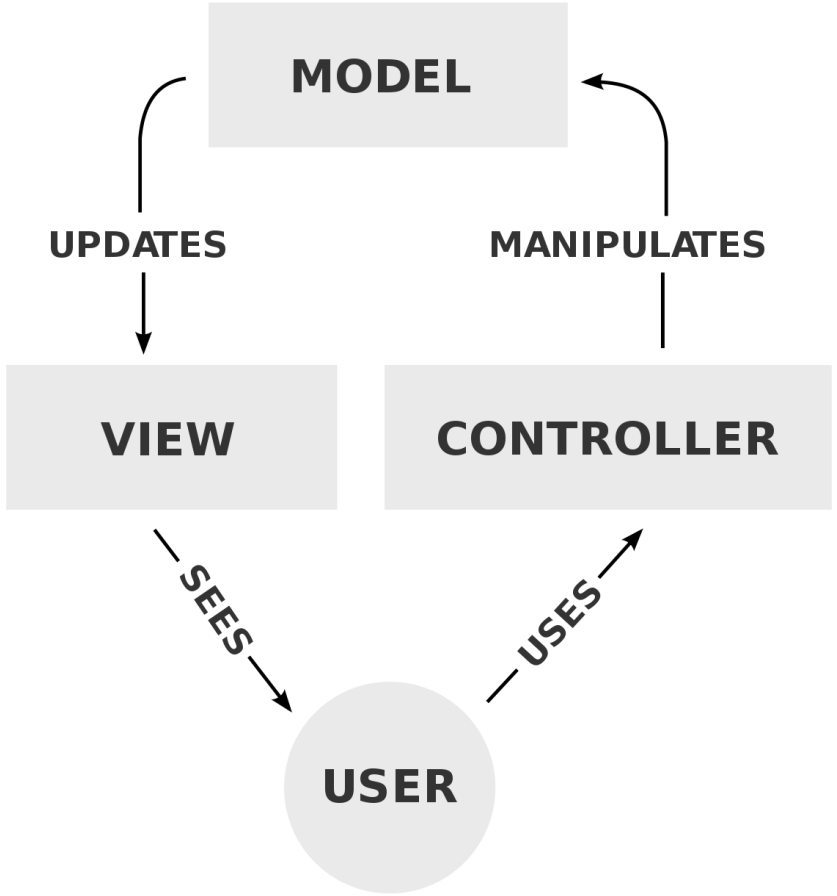
Design Pattern Example: Singleton

- Only allow to use a single instance for a class

```
class Env {
public:
    static Env& instance() {
        static Env* instance = new Env();
        return *instance;
    }
private:
    Env() {}
}
```

MVC: Model – View – Control

- Split data, UI, and management



wikipedia

```
class Model {
private:
    string data;
    Model() {};
    virtual ~Model() {};
public:
    std::string getData() { return data; }
    void setData(const string& _data) {
        data = _data;
    }
};
```

MVC: Model – View – Control

```
class View {
public:
    View(void) {};
    virtual ~View(void) {};
    void showMsg(const string& msg) {
        cout << msg << endl;
    }
    std::string receiveMsg() {
        std::string msg;
        cin >> msg;
        return msg;
    }
};
```

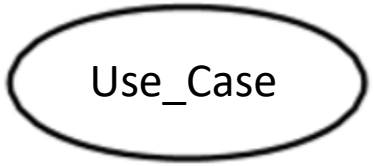
```
class Controller {
public:
    Model model;
    View view;
    Controller(void) {};
    virtual ~Controller(void) {};
    void run() {
        view.showMsg("Put data");
        model.setData(view.receiveMsg());
        view.showMsg(model.getData());
    }
}
```

Unified Modeling Language

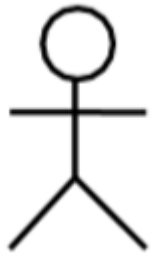
- UML
  - Object Management Group (OMG) Standard (since 1997)
  - Based on work from Booch, Rumbaugh, Jacobson
  - General-purpose, developmental, modeling language
    - Providing a standard way to visualize the design of a system (SE)
    - Particularly useful for OO design

Use Case Diagram

- Use cases
  - A sequence of interaction
  - Use case specification
    - Unique name, Participating actors, Entry conditions, Flow of events, Exit conditions, Special requirements
  - Use case model consists of all use cases(representing functionality)



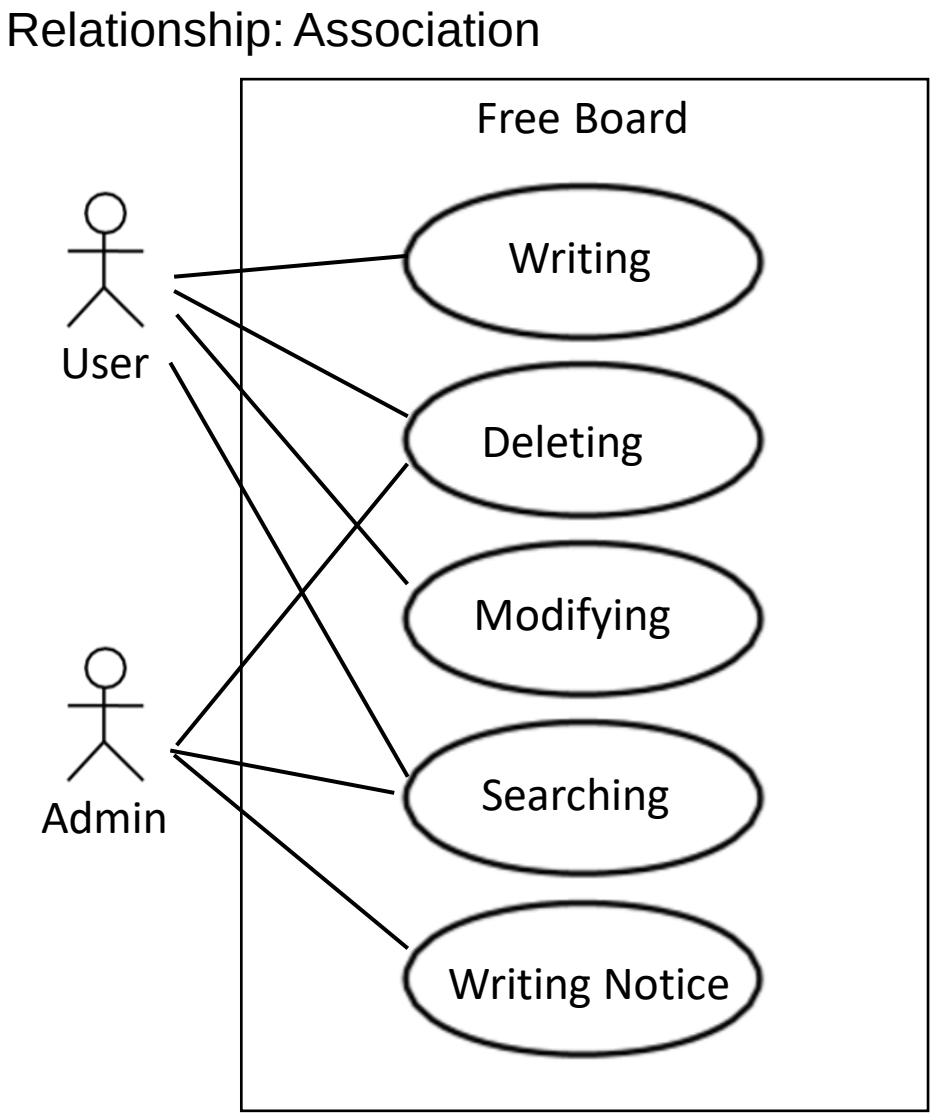
- Actors
  - External entity which communicates with the system : has role
    - User, External system, Physical environment
    - E.g.) Student, employee, Manger, GPS



Example: Use Case Diagram

- Client Requirements
  - We need free board
  - It is able to search by name, date, ..
  - Administrator maintains the board

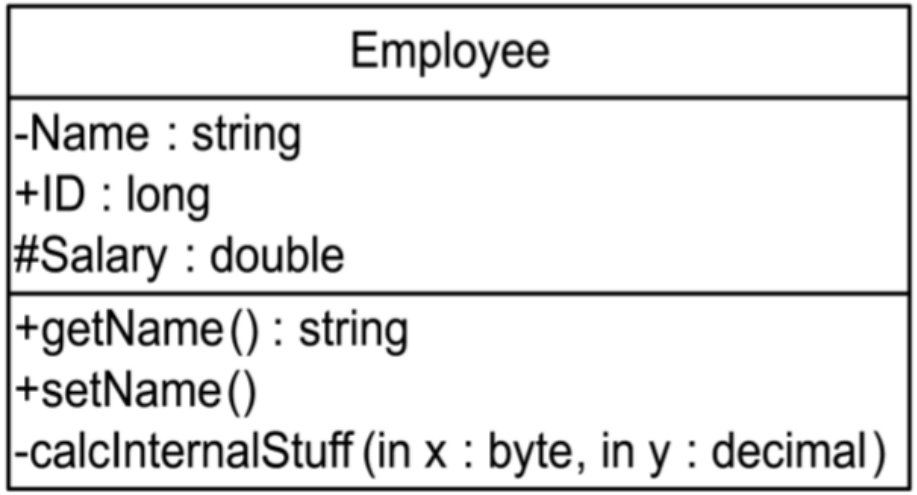
- Functional behaviors
  - E.g.) Writing a new article, Modifying, Deleting, …



Class Diagram

- Give an overview of a system by showing its classes and the relationships among them

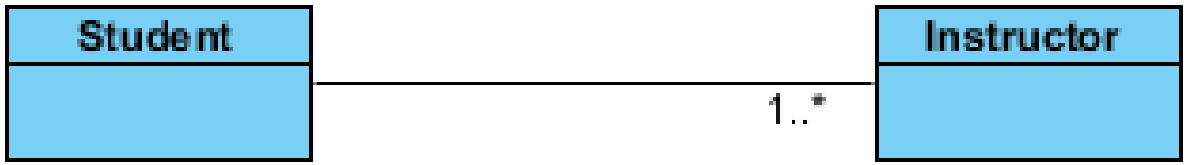
- A class is a rectangle divided into three parts
  - Class name
  - Class attributes (i.e., member variables)
  - Class operations (i.e., member functions)
- Modifiers
  - Private: -
  - Public: +
  - Protected: #
  - Static: underlined
- Abstract class: name in italics



Class Diagram: Relationship

- Association ——————— ⟶
  - A relationship between instances of two classes, where one class must know about the other to do its work,
  - e.g., client communicates to server

Dependency?

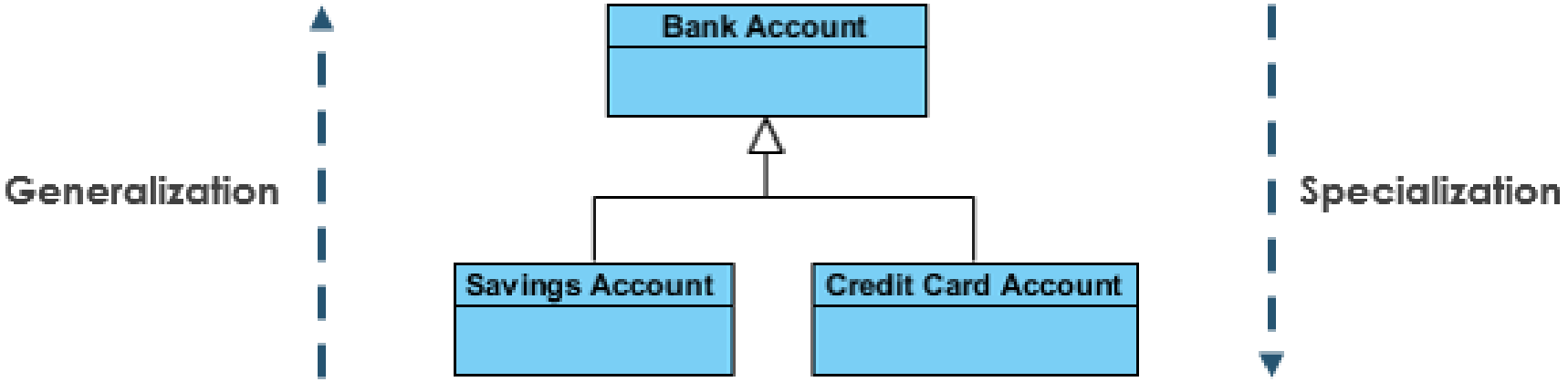


- n..m n to m instances
- * no limit on the number of instances(including none)
- 1 exactlyone instance
- 1..* at least one instance

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-

Class Diagram: Relationship

- Inheritance(Generalization/Specialization) 
  - **Generalization:** a mechanism for combining similar classes of objects into a single, more general class
  - **Specialization:** the reverse process of Generalization means creating new sub-classes from an existing class.



Let’s use Object Oriented Design

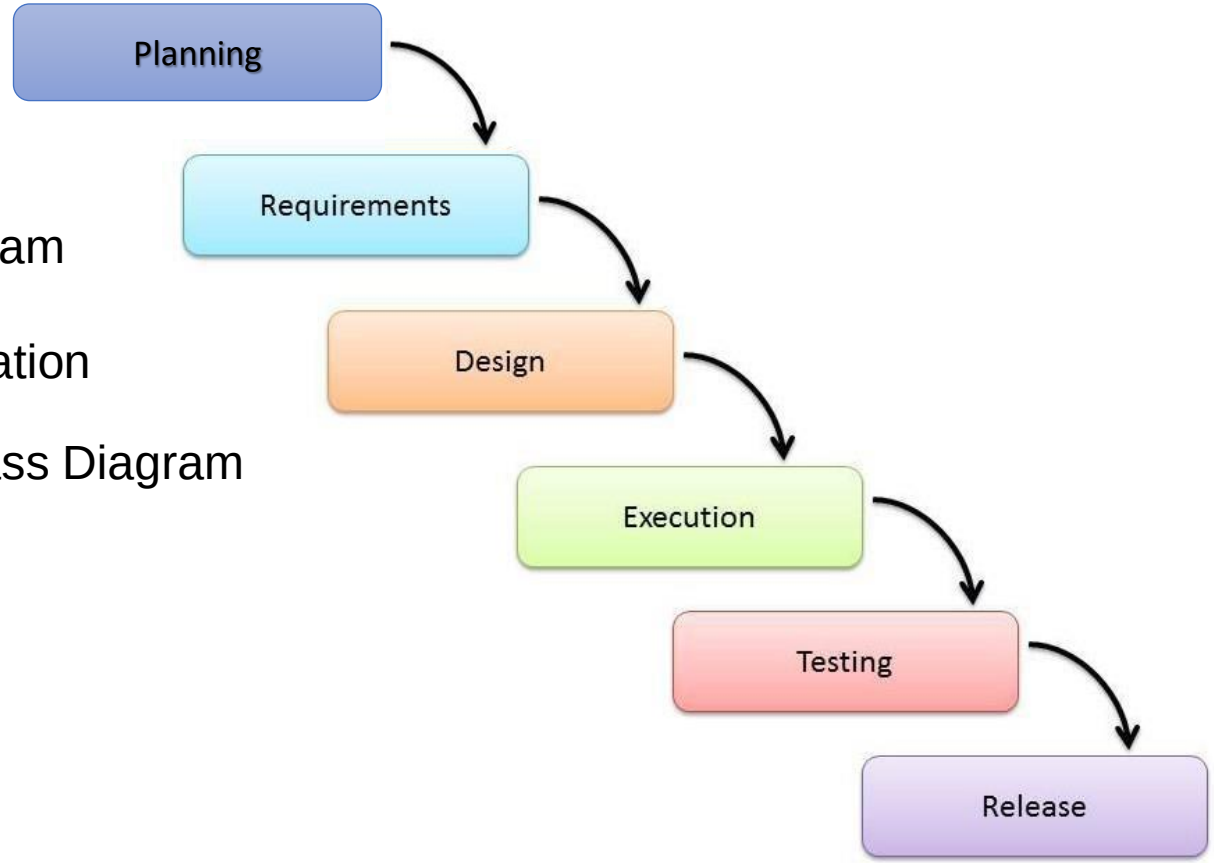
Development Life Cycle

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
  - …

Questions?