Singleton and Factory patterns

Table of Contents

2020.10

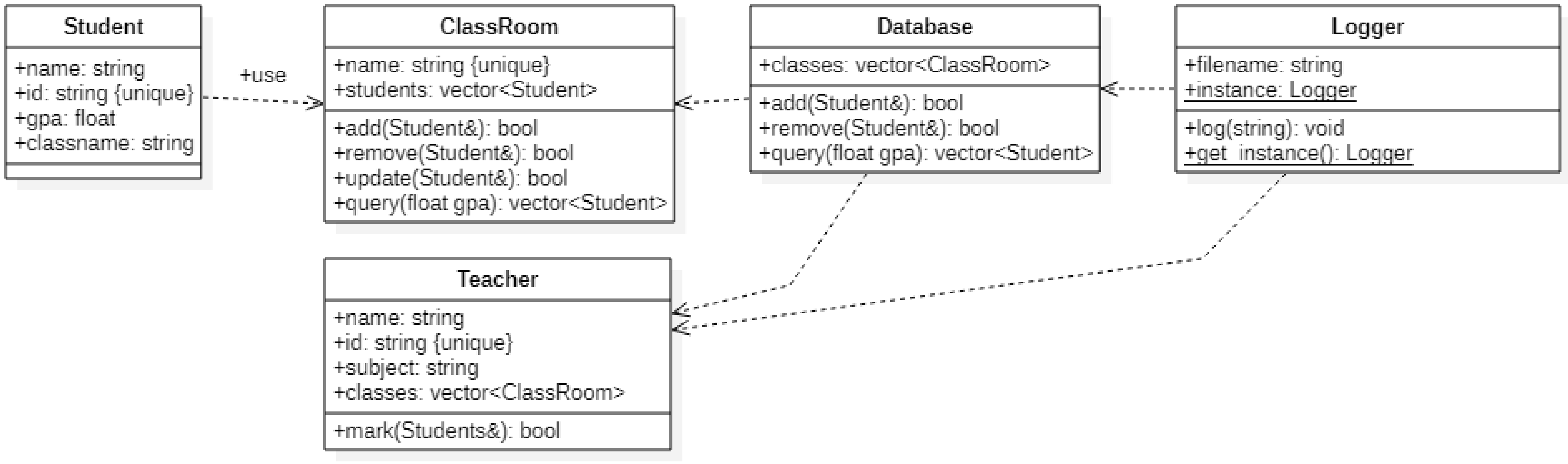
VC DCV

Singleton and Factory patterns

SINGLETON PATTERN

1. Problem

A school wants to create a software to manage its classes and students, each time add/remove/edit data of students/classes, the software shall log the changes to a file for tracking. School has teachers, they can update gpa of students in classes.

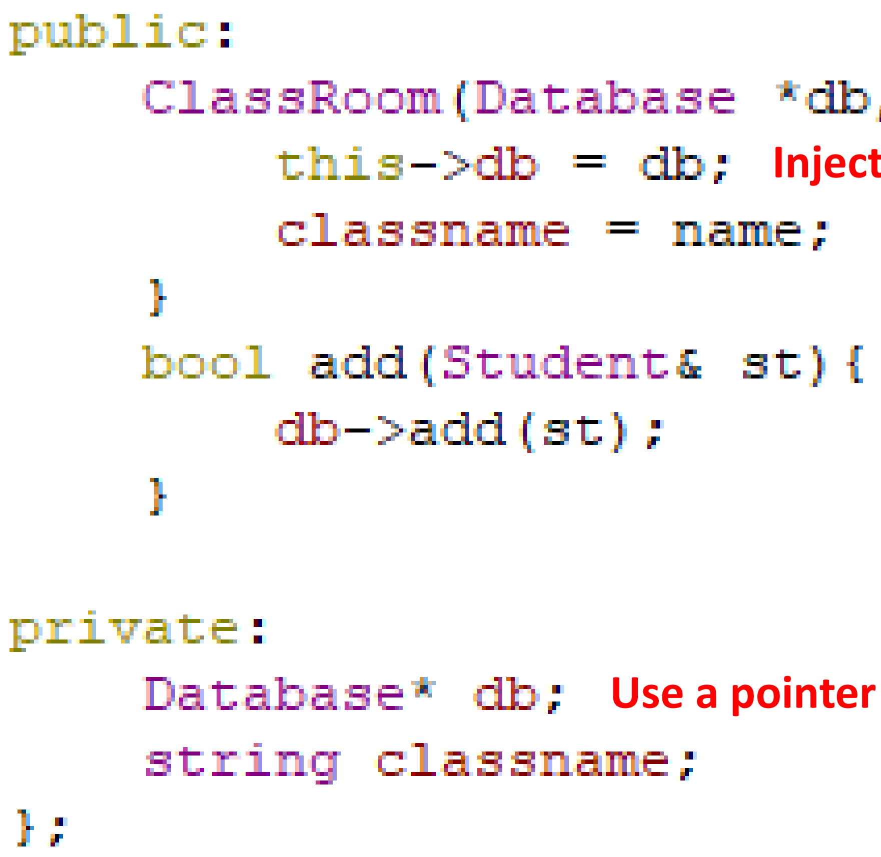


- A school can have many classes, each class has many students.
- A school can only have 1 database. Both ClassRoom and Teacher can update database
- Logger shall writes log to only 1 file.

=> Key point:

- Use a pointer to access a common Database instance (Dependency Injection).
- Use static class.
- Use singleton pattern

2. Some solutions



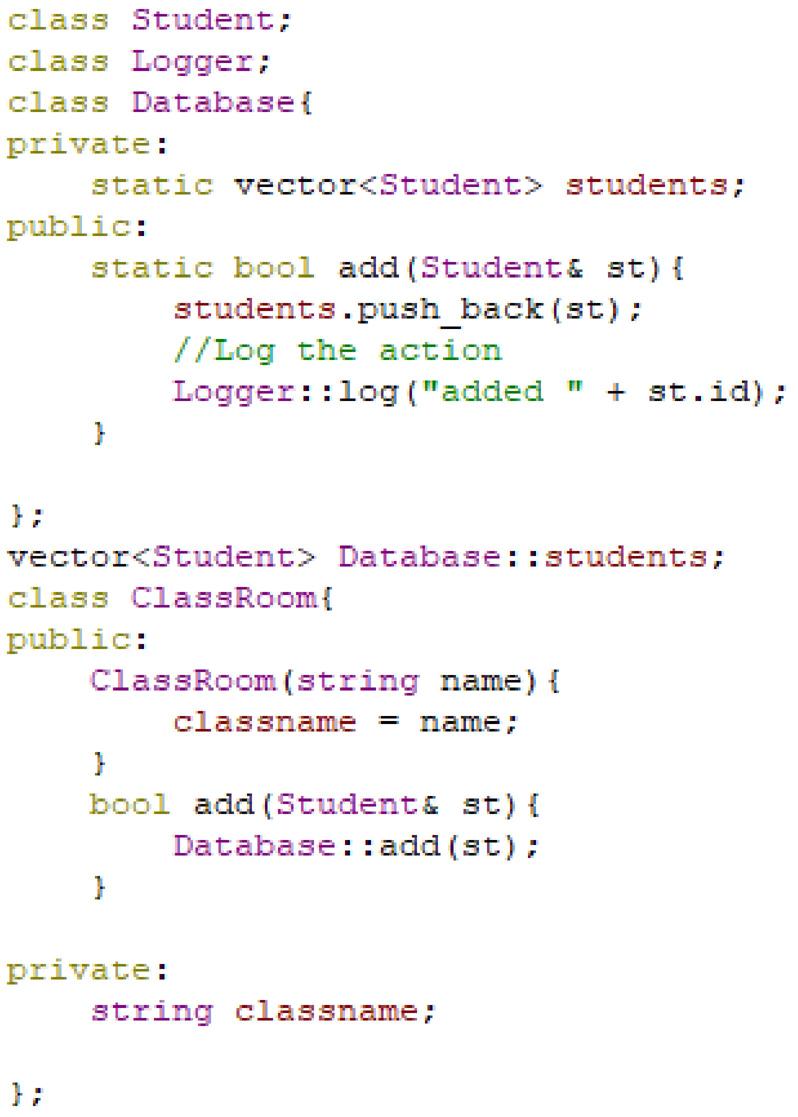
```
public:
    ClassRoom(Database *db, string name){
        this->db = db;
        classname = name;
    }
    bool add(Student& st){
        db->add(st);
    }

private:
    Database* db;
    string classname;
};
```

Inject database object in constructor

Use a pointer to access a database object

2. Some solutions



```
class Student;
class Logger;
class Database{
private:
    static vector<Student> students;
public:
    static bool add(Student& st){
        students.push_back(st);
        //Log the action
        Logger::log("added " + st.id);
    }

};
vector<Student> Database::students;
class ClassRoom{
public:
    ClassRoom(string name){
        classname = name;
    }
    bool add(Student& st){
        Database::add(st);
    }

private:
    string classname;

};
```

Use static function of class

2. Some solutions

Use singleton pattern

Private constructor and destructor

Delete copy constructor and assign operator

Public a static method to get the instance

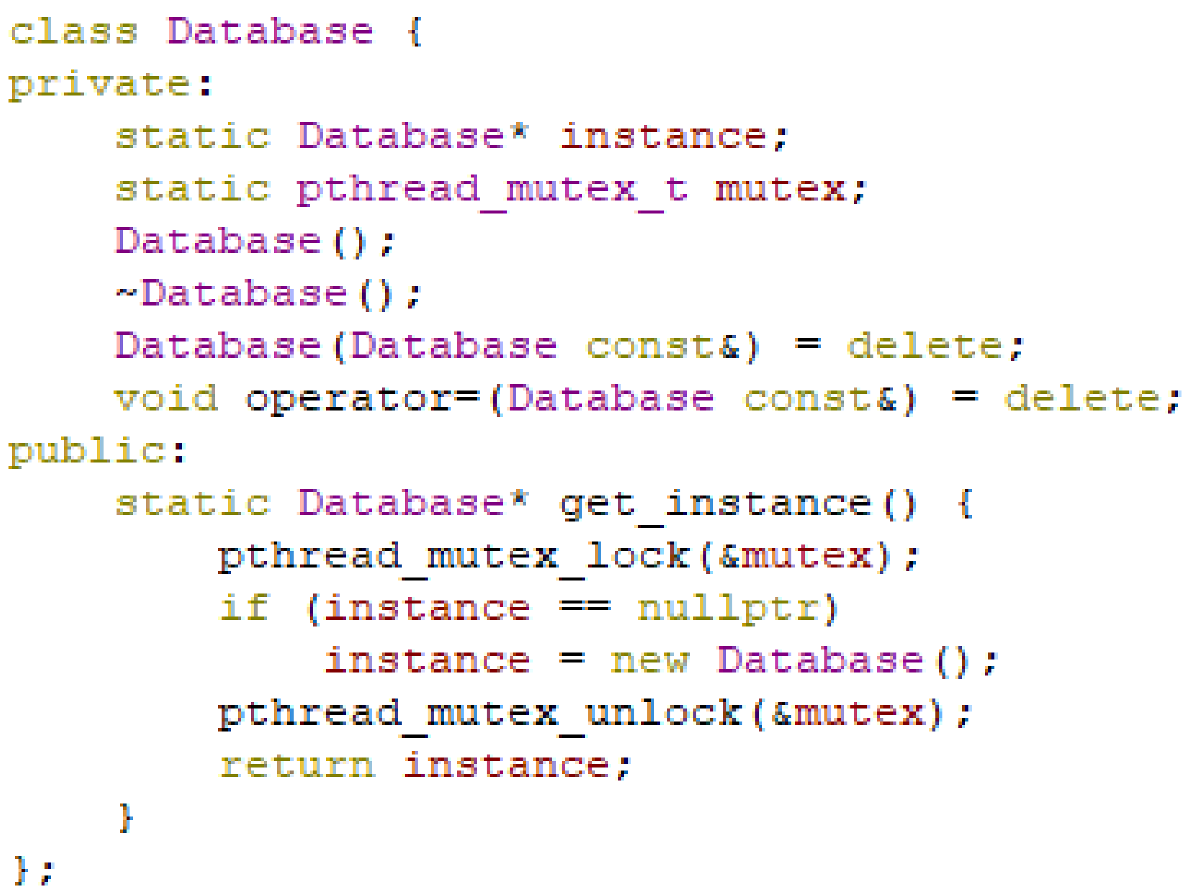
```
class Student;
class Logger;
class Database{
private:
    Database(){}
    ~Database(){}
    Database(Database const&) = delete;
    void operator=(Database const&) = delete;
    static Database* instance;
    vector<Student> students;
public:
    static Database* get_instance() {
        if (instance == nullptr) {
            instance = new Database();
        }
        return instance;
    }
    bool add(Student& st){
        students.push_back(st);
        //Log the action
        Logger::get_instance()->log("added " + st.id);
    }
};
Database* Database::instance = nullptr;

class ClassRoom{
public:
    ClassRoom(string name){
        classname = name;
    }
    bool add(Student& st){
        Database::get_instance()->add(st);
    }

private:
    string classname;
};
```

2. Some solutions

Singleton pattern with thread safe: use mutex lock.



```
class Database {
private:
    static Database* instance;
    static pthread_mutex_t mutex;
    Database();
    ~Database();
    Database(Database const&) = delete;
    void operator=(Database const&) = delete;
public:
    static Database* get_instance() {
        pthread_mutex_lock(&mutex);
        if (instance == nullptr)
            instance = new Database();
        pthread_mutex_unlock(&mutex);
        return instance;
    }
};
```

Singleton and Factory patterns

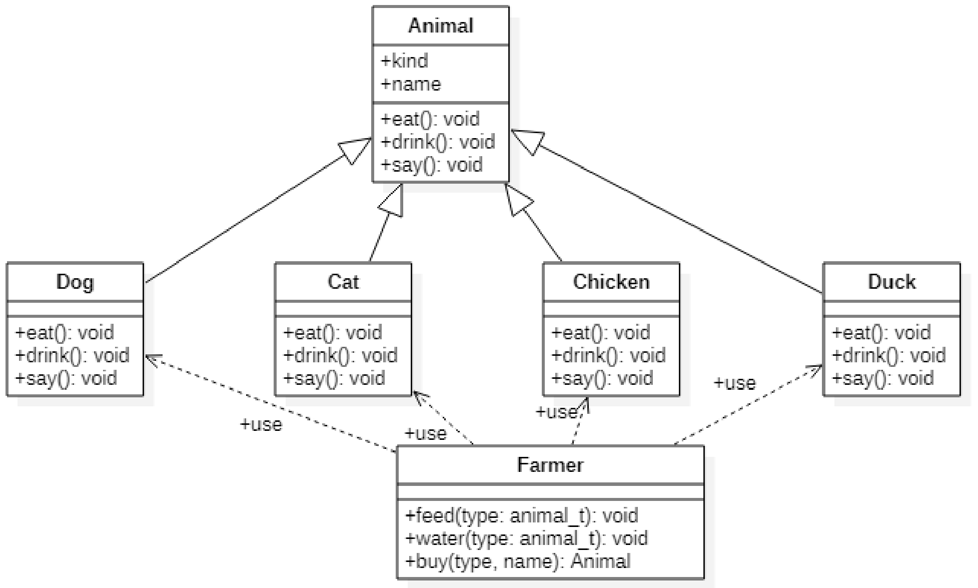
FACTORY PATTERN

1. Problem

The Funny Farm:

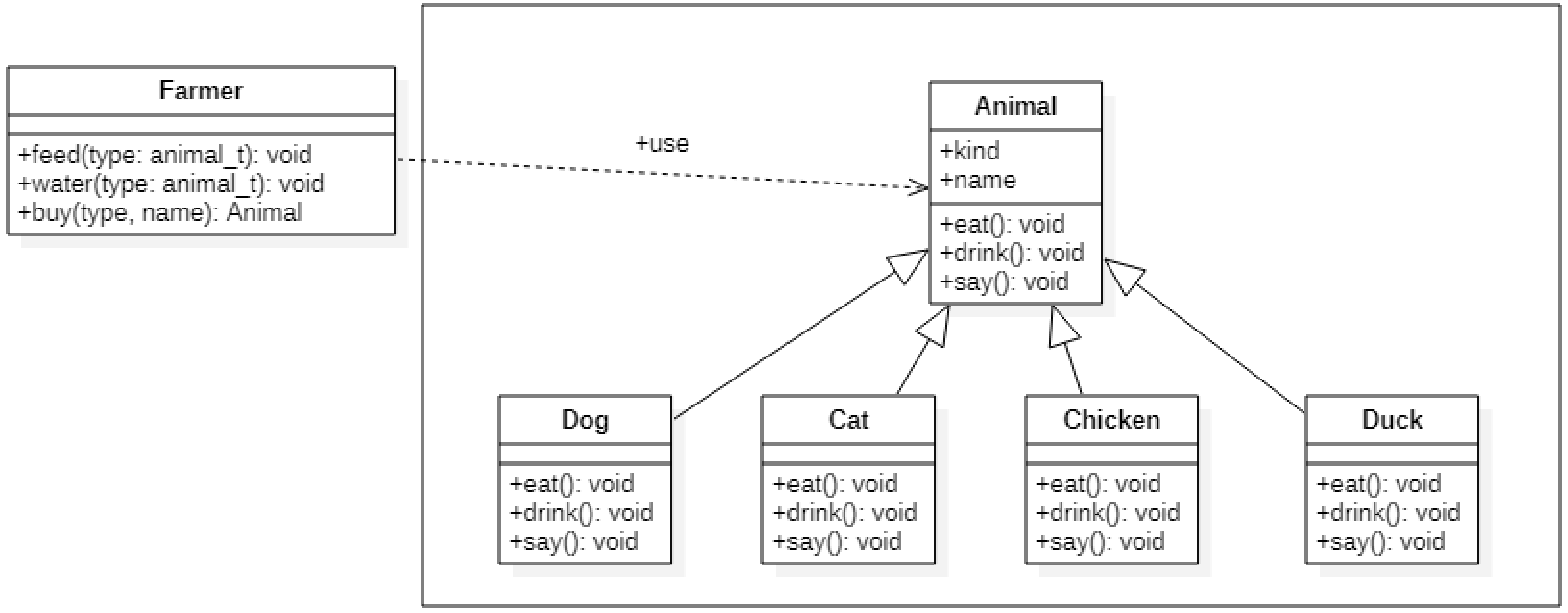
- A farm has many kind of animals: Dog, Cat, Chicken, Duck, …
- The farmer can buy new animals for his farm.
- The Farmer can feed an animal by an amount of food and water depends on its kind.

=> A simple solution:



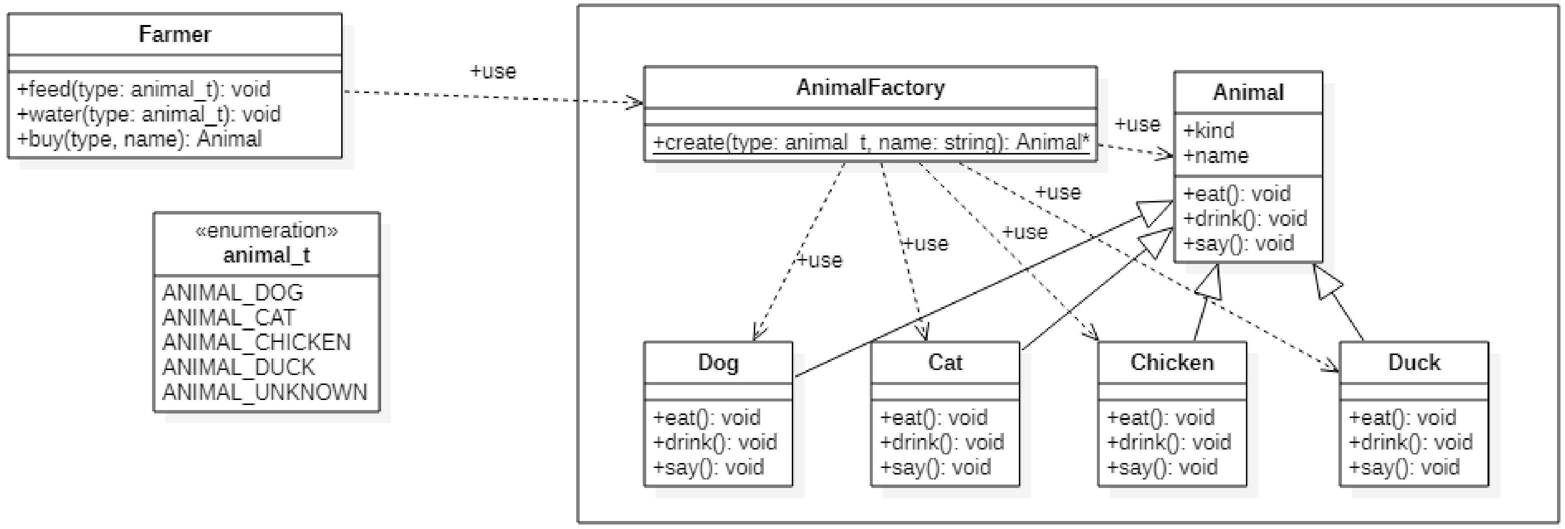
- Need to include all kinds in Farmer class
- When add/remove a kind of animal, need to update farmer source code.
- If the farm extends to hundreds of animal kinds => Hard to maintain.

2. Some solutions



- When add/remove a kind of animal, need to update farmer source code.
- If the farm extends to hundreds of animal kinds => Hard to maintain.

2. Some solutions



- Use another class as a factory to create a kind of Animal.
- When add/remove some kinds of animal, only need to update AnimalFactory class => Easier to maintain.

```
Animal* AnimalFactory::create(animal_t type, string name){
    Animal* animal = nullptr;
    switch (type)
    {
        case ANIMAL_DOG:
            animal = new Dog(name);
            break;
        case ANIMAL_CAT:
            animal = new Cat(name);
            break;
        ...
        default:
            animal = nullptr;
    }
    return animal;
}
```

Q&A

Thank you!