МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

“НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ”

Кафедра защиты информации

Лабораторная работа №1 по дисциплине «Технологии и методы программирования»

### Рассмотрение принципов Singleton, относительно паттернов, используемых в ООП.

### Вариант 2.

Группа: АИ-02

Выполнил: Григорьев Д.А.

Проверил: Медведев М.А.

Оценка:

Новосибирск 2022

**Цели и задачи работы:** рассмотреть сильные и слабые стороны внедрения паттерна Singleton и выполнить реализацию, если бы в задаче использовался паттерн SOLID. Рассмотреть проблематику тестирования программного кода и связанности кода.

**Задание**

Рассмотреть предметную область на основании изначальных данных, построить модели программного кода с использованием паттерна Singleton и SOLID. Реализовать задачу и выстроить аргументированную защиту/критику кода оппонента. Результатом работы программного кода должна быть имитация процесса задания.

**Вариант 2.** Регулятор громкости в компьютере

**Текст программы:**

***SOLID:***

#include <iostream>

#include <memory>

class System {

public:

System() {};

~System() {}

protected:

friend class Mute;

friend class Unmute;

friend class getVolume;

friend class IncreaseVolume;

friend class DecreaseVolume;

int value = 0, muteVolume = false;

};

class getVolume {

public:

void printVolume(std::shared\_ptr<System> sys) {

std::cout << "Volume is " << sys->value << std::endl;

};

};

class Mute {

public:

void mute(std::shared\_ptr<System> sys) {

sys->muteVolume = true;

std::cout << "Volume is muted\n";

}

};

class Unmute {

public:

void unmute(std::shared\_ptr<System> sys) {

sys->muteVolume = false;

std::cout << "Volume is unmuted\n";

}

};

class DecreaseVolume {

public:

void decreaseVolume(std::shared\_ptr<System> sys, int val) {

sys->value -= val;

if (sys->muteVolume == true) {

std::cout << "Current volume is 0 because volume is muted\n";

}

else if (sys->value >= 0) {

if (sys->value < 0) {

sys->value = 0;

std::cout << "Volume is below zero, making it equal to zero\n";

}

else {

std::cout << "Current volume is " << sys->value << ", volume is decreased by " << val << "\n";

}

}

}

};

class IncreaseVolume {

public:

void increaseVolume(std::shared\_ptr<System> sys, int val) {

sys->value += val;

if (sys->muteVolume == true) {

std::cout << "Current volume is 0 because volume is muted\n";

}

else if (sys->value <= 100) {

if (sys->value > 100) {

sys->value = 100;

std::cout << "Volume is maxed out, making it equal to one hundred\n";

}

else {

std::cout << "Current volume is " << sys->value << ", volume is increased by " << val << "\n";

}

}

}

};

void main() {

auto sys = std::make\_shared<System>();

auto decVol = std::make\_shared<DecreaseVolume>();

auto incVol = std::make\_shared<IncreaseVolume>();

auto getVol = std::make\_shared<getVolume>();

auto muteVol = std::make\_unique<Mute>();

auto unmuteVol = std::make\_unique<Unmute>();

getVol->printVolume(sys);

decVol->decreaseVolume(sys, 10);

muteVol->mute(sys);

incVol->increaseVolume(sys, 30);

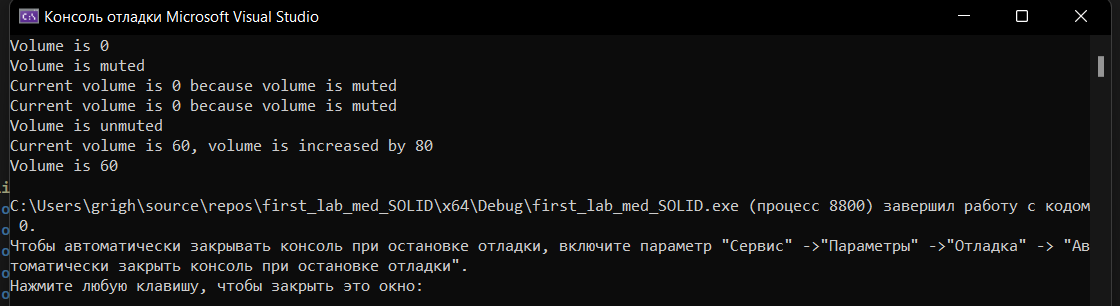
decVol->decreaseVolume(sys, 40);

unmuteVol->unmute(sys);

incVol->increaseVolume(sys, 80);

getVol->printVolume(sys);

}



**Анализ кода:**

*Single Responsibility Principle*: Изначальноне было соблюдено, в дальнейшем после ревью было исправлено (класс Volume содержал 2 метода, было решено разделить на 2 класса)

*Open-Closed Principle*: Было соблюдено изначально.

*Liskov Substitution Principle*: Не соблюдено.

*Interface Segregation Principle*: Не было соблюдено изначально, в дальнейшем было исправлено после ревью (в абстрактном классе System был виртуальный метод, которая по итогу нигде не использовалась)

*Dependency Inversion Principle*: Было соблюдено изначально

**Плюсы использования паттерна SOLID:**

* Принцип единственной ответственности. Если код программы будет написан в соответствии с этим принципом, то он будет легок для восприятия и понимания, а также его удобно тестировать. Ломается одна функция, а не вся программа, в соответствии с этим легко обнаружить место ошибки.

Пример: у нас есть раздельные классы Mute и Unmute для оббезвучивания и включения громкости

. . .

class Mute {

public:

void mute(std::shared\_ptr<System> sys) {

sys->muteVolume = true;

std::cout << "Volume is muted\n";

}

};

. . .

* Принцип открытости – закрытости. При условии, что этот принцип выполняется, мы можем лишь добавить код и функционал, не затронув предыдущий код. Что также положительно скажется на тестировании.

Пример: если мы затронем изначальную абстракцию System, мы не сможем в дальнейшем использовать поле muteVolume.

. . .

class System {

public:

System() {};

~System() {}

protected:

friend class Mute;

friend class Unmute;

friend class getVolume;

friend class IncreaseVolume;

friend class DecreaseVolume;

int value = 0;

};

. . .

class Mute {

public:

void mute(std::shared\_ptr<System> sys) {

sys->muteVolume = true; // Класс “System” не содержит члена “mutevolume”

std::cout << "Volume is muted\n";

}

};

. . .

* Принцип подстановки Лисков. В соответствии с этим принципом есть возможность заменять наследниками родителей без лишнего переписывания кода.
* Принцип разделения интерфейса. Каждая функция у отдельного класса имеет свою реализацию в виде другого класса. Также положительно сказывается на тестировании и позволяет не искать ошибки во всём коде.

Пример: есть свой класс получения громкости и свой класс увеличения громкости, вместо того чтобы иметь общий класс Volume

. . .

class getVolume {

public:

void printVolume(std::shared\_ptr<System> sys) {

std::cout << "Volume is " << sys->value << std::endl;

};

};

. . .

void main() {

auto sys = std::make\_shared<System>();

auto decVol = std::make\_shared<DecreaseVolume>();

auto incVol = std::make\_shared<IncreaseVolume>();

auto getVol = std::make\_shared<getVolume>();

auto muteVol = std::make\_unique<Mute>();

auto unmuteVol = std::make\_unique<Unmute>();

getVol->printVolume(sys);

. . .

* Соблюдение принципа инверсии зависимостей. В данном принципе суть состоит в том, что обеспечивается гибкость программы и возможность заменять одни классы другими при условии, что у них общий интерфейс.

Пример:

У нас есть класс System и допустим класс Volume, у класса System было бы большое количество методов, среди которых были incVol и decVol. Класс Volume наследуется от класса System, и ему не нужны другие методы кроме этих двух. Поэтому лучше создать отдельные классы на каждый метод, включая incVol и decVol, а дальше прописать наследование для System, а именно этих двух методов – и для System, и для Volume.

**Минусы использования паттерна SOLID:**

* Объёмный код за счёт создания классов;

Код занимает более 90 строк на реализацию, тогда, как тот же Singleton будет занимать до 60 строк

* Больше ресурсов системы;

Выделение памяти для каждого класса, описанного в коде:

class System {

public:

System() {};

~System() {}

protected:

friend class Mute;

friend class Unmute;

friend class getVolume;

friend class IncreaseVolume;

friend class DecreaseVolume;

int value = 0, muteVolume = false;

};

В огромных проектах может быть надобность разобраться в внутренней составляющей классов

* Несмотря на единственность, очень много зависимостей от абстракций, в большем проекте будет трудно разобраться что к чему:

Пример: В моей реализации от System зависят 5 классов. Реализация одного из них:

class Unmute {

public:

void unmute(std::shared\_ptr<System> sys) {

sys->muteVolume = false;

std::cout << "Volume is unmuted\n";

}

};

**Singleton:**

#include <iostream>

class System {

public:

System() {};

friend class Volume;

protected:

int value = 0;

};

class Windows : public System {};

class Volume {

public:

static Volume\* GetInstance() {

static Volume\* singleton = new Volume();

return singleton;

}

void setUniqueSystem(System\* n\_sys) {

sys = n\_sys;

}

void increaseVolume(int val) {

sys->value += val;

if (sys->value > 100) {

sys->value = 100;

std::cout << "Volume is maxed out\n";

}

else {

std::cout << "Current volume is " << sys->value << ", volume is increased by " << val << "\n";

}

}

void decreaseVolume(int val) {

sys->value -= val;

if (sys->value < 0) {

sys->value = 0;

std::cout << "Volume is below zero\n";

}

else {

std::cout << "Current volume is " << sys->value << ", volume is decreased by " << val << "\n";

}

}

private:

System\* sys;

Volume() = default;

Volume(const Volume&) = delete;

Volume& operator=(const Volume&) = delete;

Volume(Volume&&) = delete;

Volume& operator=(Volume&&) = delete;

};

void main() {

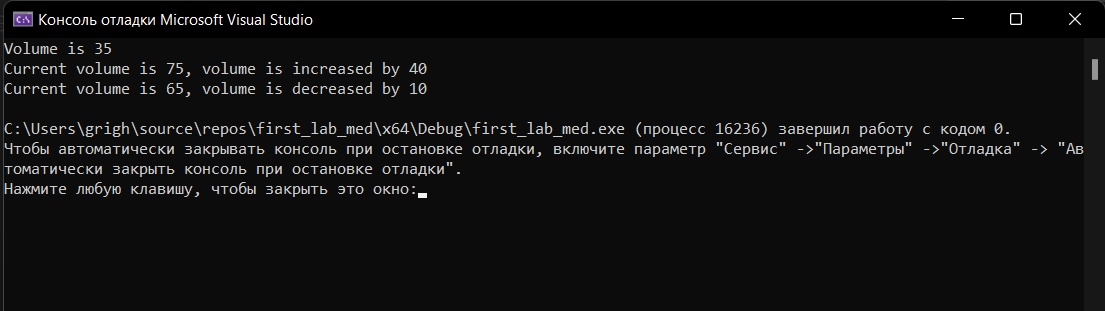
Volume\* system = Volume::GetInstance();

system->setUniqueSystem(new Windows());

system->increaseVolume(40);

system->decreaseVolume(10);

}



**Плюсы использования паттерна Singleton:**

* Затрачивается меньшее количество ресурсов, так как используется глобальное состояние;

Пример:

static Volume\* GetInstance() {

static Volume\* singleton = new Volume();

return singleton;

}

**Минусы использования паттерна Singleton:**

* Неудобен для расширения;
* Отсутствие интерфейса для взаимодействия;
* Единственность экземпляра под вопросом;
* Затрудняет юнит-тестирование программы;

**Выводы**

* Сильные стороны паттерна Singleton:  
  Позволяет контролировать создание экземпляров класса, ограничивая их количество одним. Предоставляет доступ к свойству через глобально доступный метод.
* Слабые стороны паттерна Singleton:  
  Singleton нарушает SRP (Single Responsibility Principle) — помимо своих обязанностей, он контролирует создание экземпляров класса. Неявная зависимость подсистем друг от друга введу наличия глобального состояния, изменения которого сложно контролировать. Это порождает ещё одну проблему — сложность тестирования, потому что появляется сильная связанность, и может оказаться так, что несколько тестов будут зависеть друг от друга.
* Сильные стороны паттерна SOLID:

Позволяет легко поддерживать/изменять код.

* Слабые стороны паттерна SOLID:

Увеличеное время на поддержку и не всегда понятное соблюдение принципов. Если речь идёт про бизнес, возможно и так что SOLID могут в какой-то момент вообще отбросить, уступив дорогу трате времени на проработку фич.

Вывод: мы изучили основополагающие паттерны, применили их на практике, а также выявили сильные и слабые стороны. Провёл работу для понимания данных паттернов хоть на каком-то удобоваримом уровне (но это не точно).