Київський національний університет імені Тараса Шевченка

Кафедра інтелектуальних програмних систем

**Основи об’єктно-орієнтованого програмування**

2 курс ОКР „бакалавр”, 2 семестр

**Екзаменаційний білет № 48**

1. Принципи підтримки та OCP. Обов’язково навести приклади з власного коду.
2. Порівняти патерни Proxy, Facade. Приклади з власного коду.
3. Спроектувати (наприклад, використовуючи UML) універсальну платформу для підтримки різних настільних ігор.

Затверджено на засіданні кафедри інтелектуальних програмних систем 14.05.2020р., протокол № 12

Екзаменатор Жереб К.А.

Зав. кафедри Провотар О.І.

**1.Принципи підтримки та OCP. Обов’язково навести приклади з власного коду.**

Open closed principle один з яскравих прикладів принципів підтримки який говорить: бути відкритим для розширення, але закритим для переписування старого коду

Взагалі принципи підтримки, по суті, мають на увазі що код написаний раніше ще буде модернізуватися, а не буде закинутий назавжди, тобто: потрібно писати такий, не сильно пов'язаний, код щоб його було реально модернізувати

**2.Порівняти патерни Proxy, Facade. Приклади з власного коду.**

Facades і Proxy - два структурних патерна, проксі, можна сказати, окремий випадок фасаду.

У чому сенс фасаду: надати клієнту простий API, який приховує всю складність реалізації.

У чому сенс проксі: перехоплює звернення до об'єкту і робить якусь додаткову логіку, тобто контролює процес роботи з об'єктом

Приклади взяті з refactoring.guru

**Приклад Facade**

/ \*\*

\* Підсистема може приймати запити безпосередньо з фасаду або клієнта.

\* У будь-якому випадку для підсистеми «Фасад» - це ще один клієнт, і це не так

\* частина Subsystem.

\* /

class Subsystem1 {

public:

std::string Operation1() const {

return "Subsystem1: Ready!\n";

}

// ...

std::string OperationN() const {

return "Subsystem1: Go!\n";

}

};

/ \*\*

\* Деякі фасади можуть працювати з декількома підсистемами одночасно.

\* /

class Subsystem2 {

public:

std::string Operation1() const {

return "Subsystem2: Get ready!\n";

}

// ...

std::string OperationZ() const {

return "Subsystem2: Fire!\n";

}

};

/ \*\*

\* Клас «Фасад» забезпечує простий інтерфейс до складної логіки одного або

\* кілька підсистем. Фасад делегує запити клієнта до

\* відповідні об'єкти в підсистемі. Фасад також відповідає

\* управління своїм життєвим циклом. Все це захищає клієнта від небажаних

\* складність підсистеми.

\* /

class Facade {

protected:

Subsystem1 \*subsystem1\_;

Subsystem2 \*subsystem2\_;

/ \*\*

\* Залежно від потреб вашої програми, ви можете надати Фасад

\* існуючі об'єкти підсистеми або змушують Фасад створювати їх самостійно.

\* /

public:

/ \*\*

\* У цьому випадку ми делегуємо право власності на пам'ять до класу Facade

\* /

Facade(

Subsystem1 \*subsystem1 = nullptr,

Subsystem2 \*subsystem2 = nullptr) {

this->subsystem1\_ = subsystem1 ?: new Subsystem1;

this->subsystem2\_ = subsystem2 ?: new Subsystem2;

}

~Facade() {

delete subsystem1\_;

delete subsystem2\_;

}

/ \*\*

\* Методи Фасаду - це зручні ярлики до складних

\* функціональність підсистем. Однак клієнти отримують лише частину

\* можливості підсистеми

\* /

std::string Operation() {

std::string result = "Facade initializes subsystems:\n";

result += this->subsystem1\_->Operation1();

result += this->subsystem2\_->Operation1();

result += "Facade orders subsystems to perform the action:\n";

result += this->subsystem1\_->OperationN();

result += this->subsystem2\_->OperationZ();

return result;

}

};

/ \*\*

\* Клієнтський код працює зі складними підсистемами через простий інтерфейс

\* надається Фасадом. Коли фасад управляє життєвим циклом підсистеми,

\* клієнт може навіть не знати про існування підсистеми. Це

\* підхід дозволяє тримати складність під контролем.

\* /

void ClientCode(Facade \*facade) {

// ...

std::cout << facade->Operation();

// ...

}

/ \*\*

\* У клієнтському коді можуть бути вже створені деякі об'єкти підсистеми. В

\* у цьому випадку, можливо, варто ініціалізувати Фасад із цими об’єктами

\* замість того, щоб дозволяти Фасаду створювати нові екземпляри.

\* /

int main() {

Subsystem1 \*subsystem1 = new Subsystem1;

Subsystem2 \*subsystem2 = new Subsystem2;

Facade \*facade = new Facade(subsystem1, subsystem2);

ClientCode(facade);

delete facade;

return 0;

}

**Приклад Proxy**

#include <iostream>

/ \*\*

\* Інтерфейс Subject оголошує загальні операції як для RealSubject, так і для

\* Проксі. Поки клієнт працює з RealSubject за допомогою цього інтерфейсу,

\* ви зможете передати йому проксі-сервер замість реальної теми.

\* /

class Subject {

public:

virtual void Request() const = 0;

};

/ \*\*

\* RealSubject містить деяку основну логіку бізнесу. Зазвичай RealSubjects є

\* здатний виконувати корисну роботу, яка також може бути дуже повільною або чутливою -

\* напр. виправлення вхідних даних. Проксі-сервер може вирішити ці проблеми без будь-яких

\* зміни коду RealSubject.

\* /

class RealSubject : public Subject {

public:

void Request() const override {

std::cout << "RealSubject: Handling request.\n";

}

};

/ \*\*

\* Проксі має інтерфейс, ідентичний RealSubject.

\* /

class Proxy : public Subject {

/\*\*

\* @var RealSubject

\*/

private:

RealSubject \*real\_subject\_;

bool CheckAccess() const {

// Деякі реальні перевірки повинні пройти тут.

std::cout << "Proxy: Checking access prior to firing a real request.\n";

return true;

}

void LogAccess() const {

std::cout << "Proxy: Logging the time of request.\n";

}

/ \*\*

\* Проксі підтримує посилання на об'єкт класу RealSubject. Це

\* може бути ліниво завантажений або переданий клієнтові проксі.

\* /

public:

Proxy(RealSubject \*real\_subject) : real\_subject\_(new RealSubject(\*real\_subject)) {

}

~Proxy() {

delete real\_subject\_;

}

/ \*\*

\* Найпоширеніші програми схеми проксі - це ледача завантаження,

\* кешування, контроль доступу, ведення журналів тощо. Проксі може виконувати один із них

\* ці речі, а потім, залежно від результату, передайте виконання

\* той самий метод у пов'язаному об’єкті RealSubject.

\* /

void Request() const override {

if (this->CheckAccess()) {

this->real\_subject\_->Request();

this->LogAccess();

}

}

};

/ \*\*

\* Клієнтський код повинен працювати з усіма об'єктами (як з предметами, так і з)

\* проксі-сервери) через інтерфейс Subject для підтримки як реальних предметів, так і

\* проксі. У реальному житті, однак, клієнти в основному працюють зі своїми реальними предметами

\* безпосередньо. У цьому випадку, щоб легше реалізувати шаблон, ви можете продовжити

\* ваш проксі від класу реального предмета.

\* /

void ClientCode(const Subject &subject) {

// ...

subject.Request();

// ...

}

int main() {

std::cout << "Client: Executing the client code with a real subject:\n";

RealSubject \*real\_subject = new RealSubject;

ClientCode(\*real\_subject);

std::cout << "\n";

std::cout << "Client: Executing the same client code with a proxy:\n";

Proxy \*proxy = new Proxy(real\_subject);

ClientCode(\*proxy);

delete real\_subject;

delete proxy;

return 0;

}

**Спроектувати (наприклад, використовуючи UML) універсальну платформу для підтримки різних настільних ігор.**

