**Екзаменаційний білет № 58**

Принципи YAGNI та Law of Demeter. Обов’язково навести приклади з власного коду.  
  
Law of Demeter – Закон Деметри, принцип ООП що збудований на спілкуванні між класів. А точніше на зменшення кількості спілкування між класів. Головна ідея полягає у тому, що чим менше клієнтський клас знає о інших класах, тим краще. Це спрощує сам вид класу, коли він викликає лише один метод, а далі системі класи, самовизначають що треба зробити. Існує один добрий приклад для цього, що б собака бігла, треба лише дати команду «бігти», а далі вона сама визначиться як керувати лапами. Представимо це у вигляді класів, клієнтський клас буде А, клас мозку собаки Б, та лапи клас В. Трьох класів буде достатньо що б продемонструвати цей принцип.   
Клас А має зв'язок з класом Б, але він нічого не знає про клас В. К А, дає команду бігти a.run(), а далі в цій функції викликається клас Б, зі своїми процесами що самі визначають що саме треба робити, а би собака побігла. Але ні в якому разі не повинно бути a.b.run(), бо таким чином, клас А звертається до класу В, але через Б, що позрощує закон Деметри. Метод run об'єкта А має викликати методи тільки наступних типів об'єктів:

1. власне самого А
2. параметрів run()
3. будь-яких об'єктів, створених у межах run()
4. прямих компонентних об'єктів А
5. глобальних змінних, доступних А, у межах run()

Через це, класи маю гарну структуру, та незалежність друг від другу, що дозволить переписати функцію run() у класі Б без потреби змінювати А, полегшуючи роботу та працю коду. Таким чином облегшує класи та їх читабельність. Через це будується ієрархія класів, що є одним з недоліків цього закону, бо через це, для виклику інших класів та методів, потрібно робити проміжні класи, що будуть як адаптери, делегати, роблячи зв’язок між класами. Через що сам дизайн коду та ООП може стати навантаженим та жахливим через велику кількість класів адаптерів.   
Class A  
{  
private:  
 B local\_b;  
public:  
 void run()  
 {  
 local\_b.run();  
 }  
};  
Class B  
{  
 private:  
 C local\_C;  
public:   
 void run()  
 {  
 local\_C.move\_left\_second;  
 local\_C.move\_right\_first;  
 local\_C.move\_bode;  
 local\_C.move\_left\_first;  
 local\_C.move\_right\_second;  
 local\_C.move\_bode;  
 }  
};  
Class C  
{  
  
 void move\_left\_second()  
 {  
 move(left\_second);  
 }  
 void move\_right\_first()  
 {  
 move(right\_first);  
  
 }  
 void move\_bode()  
 {  
 move(move\_bode);  
  
 }  
 void move\_left\_first()  
 {  
 move(left\_first);  
 }  
 void move\_right\_second()  
 {  
 move(right\_second);  
 }  
};  
Ось і псевдо код, демонстрації цього закону, на принципу бігу собаки.   
  
YAGNI «You aren't gonna need it», тобто «вам це не потрібно». Принцип, головна ідея котрого, це у реалізації ТІЛЬКИ потрібних функцій. Тобто відмовлення від усього зайвого. Якщо проще кажучи, то головная ідея у оптимізації роботи, а точніше, робити те що треба і тільки це. Зазвичай це більше для економії часу чи бізнесу, якщо нам заплатили за цю функцію, то ми її і буде робити, бо іноді буває що програмісти можуть, лізти поперед батька в пекло, та таким чином робити чи додаткові функції чи покращувати цю котра є, через це, програміст робить роботу за котру йому не платили. Ще одна ідей, це робити те, що зараз потрібно для програми, без думки «Це може бути корисно потім», бо і більшості випадків трапляється інакше. У тому ж числі, роблячі функції на «майбутне» вони могут заважати додаванню потрібних функцій. Якщо брати простий приклад з життя, то якщо замовник замовив, мопед, то потрібно зробити саме це, а не мотоцикл. Звісно мотоцикл буде швидше та краще за мопед. Тому інженер у момент праці може замислитися та почати робити щось краще за мопед, на що він витратить більше часу та ресурсів, але ніхто йому за такі покращення не платив, через що він може піти в убиток, через те що, умовно кажучи продав мотоцикл за ціною мопеда.   
  
Важко принести код у приклад, що б точно показати цей принцип, але можна описати словами.  
  
Наприклад, замовник попросив, код, що має зберігати назви станцій у вигляді Б дерева, та сортування бульбашкою за нумерацією станцій, на дорозі.  
Тобто ми і повині це зробити, тобто не треба ні Б дерево, ні quick\_sort хоча вона можуть бути швидші та краще, але таким чином сам код стане дороще, або замовник взаголі може відмовитися.   
  
Звістно, якщо продивитися плюси і мінуси, може здатися що YAGNI схожий на KISS. АЛЕ! Ідея KISS - це спрощення, YAGNI - не робити зайве.  
  
Якщо YAGNI не виконується, а навпаки то:  
- потрібно більше часу.  
- можлива кількість збільшення кількості помилок.  
- навантаження коду та структури.  
- можливе ускладнення додавання нових функції.   
- зайві функції.  
- збільшення кількості пам’яті на збереження файлів.

2 Порівняти патерни Proxy, Memento. Приклади з власного коду.

Proxy – замісник – це структурний патерн.

Memento – знімок – це патерн поведінки.

Proxy застосовується, коли треба створити класс-дубляр для основного красу, який буде повністю копіювати інтерфейс основного класу, але дозволить додавати виконання додаткових дій до або після виконання операції з основного класу. Також Proxy дозволяє вирішити проблему відкладеної ініціалізації (нам немає необхідності створювати екземпляр основного класу, ми зможемо зробити це потім, «по акту» необхідності в цьому)

Нехай у нас є база даних і деякий інтерфейс доступу: шукати данні, додавати данні, видаляти данні.

Ось приклад коду для додавання данних в базу

class InterfaceDataBase

{

public:

bool write(const Record& record) = 0;

};

class DabaBase

{

public:

bool write(const Record& record) override   
 {

}

};

class ProxyDataBase

{

public:

bool write(const Record& record) override {

if (db == nullptr)

init();

bool result = db->write(record);

LOG("New record was added to the database " +

(result ? "successfully" : "unsuccessfully!!!") );

return result;

}

private:

void init();

DabaBase\* db {nullptr};

};

Memento патерн потрібен, коли нам треба зберігати поточний стан певного об’єкту, а також потім мати змогу повернутись до цього збереженого стану (тобто щоб об’єкт мав таку внутрішню структура, як на момент збереження). При правильному розумінні інкапсуляції таке неможливо зробити ззовні, або це буде вимагати великої кількості додаткових дій. Тому Memento – це саме поведінка об’єкта.

Розглянемо приклад: є симулятор машини, і ми б хотіли мати змогу зберігати стан машини. Причому не всі поля треба зберігати: наприклад ім’я машини (на або держ.номер, марка) не буде змінюватися під час водіння, але поточна позиція швидкість і навіть нанесені пошкодження потрібно вміти зберігати і відтворювати.

class Car

{

public:

class Memento

{

public:

Memento(const double& speed,

const Position& position,

const double& fuel,

const Damage& damage)

: speed{speed}

, position{position}

, fuel{fuel}

, damage{damage}

{

}

double speed;

Position position;

double fuel;

Damage damage;

};

Memento get\_state()

{

rerurn Memento(speed, position, fuel, damage);

}

void restore\_state(const Memento& state)

{

speed = state.speed;

position = state.position;

fuel = state.fuel;

damage = state.damage;

}

private:

std::string name;  
 double speed;

Position position;

double fuel;

Damage damage;

};

state\_storage.push( my\_car.get\_state() );

my\_car.restore\_state( state\_storage.pop() );

3 Спроектувати (наприклад, використовуючи UML) засіб для пошуку копій програмного коду (плагіату).   
  
Картинка лежить у репозіторії.  
.

4 Задано список заданого типу, а також декілька функцій, які перетворюють цей список (наприклад, алгоритми сортування). Застосувати кожну з функцій до початкового списку (незалежно, а не послідовно). Потім порівняти результати роботи функцій, повернути таблицю попарних відмінностей (чи однакові результати, якщо ні – чи відрізняються довжини списків, якщо відрізняються елементи – індекс першої відмінності).  
   
Файл .cpp лежить у репозіторії.