- **Базовий клас** `Carriage` з атрибутами:
  - Унікальний ідентифікатор (тип int або string)
  - Тип вагона (тип string)
  - Вага вагона (тип double)
  - Довжина вагона (тип double)
- **Похідні класи від** `Carriage` зі специфічними атрибутами для кожного типу вагона:
  - `PassengerCarriage` з атрибутами загальної кількості місць для сидіння та рівня комфорту
  - `FreightCarriage` з атрибутами максимальної ваги вантажу та типу вантажу
  - `DiningCarriage` з атрибутами кількості столів та наявності кухні
  - `SleepingCarriage` з атрибутами кількості купе та наявності душових
- **Клас** `Train` який представляє зв'язний список вагонів з методами для додавання, видалення, пошуку та виведення інформації про вагони.

Завдання:

- **Реалізація класів:** Реалізувати базовий клас `Carriage` та похідні класи.
- **Створення класу** `Train`**:** Цей клас повинен представляти динамічний контейнер даних із зв'язним списком вагонів.
- **Функціонал додавання, видалення та пошуку вагонів:** Реалізувати в класі `Train`.
- **Функції для агрегації даних:** Наприклад, загальна кількість пасажирів, вагон з найбільшою вантажопідйомністю.
- **Тестові сценарії:** Розробити для перевірки функціональності реалізованих класів та методів.
- **Додаткові атрибути та методи:** Додати 10 власних атрибутів в базові класи та 5 методів обробки власних атрибутів.

```
#include <iostream>
```

```
#include <string>
#include <list>

class Carriage {
protected:
    int id;
    std::string type;
    double weight;
    double length;

public:
    Carriage(int _id, std::string _type, double _weight, double _length)
: id(_id), type(_type), weight(_weight), length(_length) {}

    virtual void displayInfo() {
        std::cout << "ID: " << id << "\nType: " << type << "\nWeight: " <<
weight << "\nLength: " << length << std::endl;
    }

    // Getter for Carriage type
    std::string getType() {
        return type;
    }
```

```
    virtual ~Carriage() {}
};
```

Похідні класи

```
class PassengerCarriage : public Carriage {
private:
    int seatingCapacity;
    std::string comfortLevel;

public:
    PassengerCarriage(int _id, double _weight, double _length, int _seatingCapacity, std::string _comfortLevel) : Carriage(_id, "Passenger", _weight, _length), seatingCapacity(_seatingCapacity), comfortLevel(_comfortLevel) {}

    void displayInfo() override {
        Carriage::displayInfo();
        std::cout << "Seating Capacity: " << seatingCapacity << "\nComfort Level: " << comfortLevel << std::endl;
    }
};

class FreightCarriage : public Carriage {
private:
    double maxLoadWeight;
    std::string cargoType;
```

```
public:
    FreightCarriage(int _id, double _weight, double _length, double
_maxLoadWeight, std::string _cargoType) : Carriage(_id, "Freight",
_weight, _length), maxLoadWeight(_maxLoadWeight),
cargoType(_cargoType) {}

    void displayInfo() override {
        Carriage::displayInfo();
        std::cout << "Max Load Weight: " << maxLoadWeight <<
"\nCargo Type: " << cargoType << std::endl;
    }
};

// Аналогічний спосіб може бути використаний для реалізації
DiningCarriage та SleepingCarriage
```

Клас `Train`:

Оскільки ми маємо розглянути управління колекцією за допомогою зв'язного списку, ми скористаємось стандартною бібліотекою C++, яка надає реалізацію зв'язного списку через шаблонний клас `list`.

```
class Train {
private:
    std::list<Carriage*> carriages;

public:
    void addCarriage(Carriage* carriage) {
```

```
    carriages.push_back(carriage);
}

void removeCarriage(int id) {
    for(auto it = carriages.begin(); it != carriages.end(); ++it) {
        if((*it)->getID() == id) {
            delete *it;
            carriages.erase(it);
            break;
        }
    }
}

void displayTrainInfo() {
    for(auto& carriage : carriages) {
        carriage->displayInfo();
        std::cout << "----------------------\n";
    }
}

~Train() {
    for(auto& carriage : carriages) {
        delete carriage;
    }
```

```
    }
};

// Головна функція для демонстрації
int main() {
    Train myTrain;
    myTrain.addCarriage(new PassengerCarriage(1, 20000, 30, 120, "High"));
    myTrain.displayTrainInfo();

    return 0;
}
```

При запуску наведеного вище прикладу коду виконуються наступні дії:

- **Створення об'єкту `myTrain` класу `Train`.**
  - Це створює порожній поїзд без вагонів.
- **Додавання вагона до поїзда.**
  - Здійснюється виклик методу `addCarriage` класу `Train`, якому передається новий об'єкт `PassengerCarriage` з ідентифікатором 1, вагою 20000, довжиною 30, кількістю місць для сидіння 120 та рівнем комфорту "High". Цей об'єкт динамічно створюється за допомогою оператора `new`.
  - Створений об'єкт `PassengerCarriage` додається до зв'язного списку `carriages` у класі `Train`.
- **Виведення інформації про поїзд.**
  - Здійснюється виклик методу `displayTrainInfo()` класу `Train`, який обходить весь зв'язний список `carriages`, викликаючи метод `displayInfo()` для кожного елемента (вагона) у списку.
  - Для кожного об'єкта вагона виводиться інформація: ID, тип, вага, довжина, а також додаткові характеристики специфічні для даного типу вагона (наприклад, кількість місць для сидіння та рівень комфорту для `PassengerCarriage`).
- **Кінець програми.**
  - Після завершення виконання програми, деструктор класу `Train` автоматично видаляє всі динамічно створені об'єкти вагонів, запобігаючи витоку пам'яті.

Результат виведення у консоль буде приблизно таким:

```
ID: 1
```

Type: Passenger

Weight: 20000

Length: 30

Seating Capacity: 120

Comfort Level: High

- **лас** `Restaurant` - ресторан, що містить атрибути назва, адреса, тип, рейтинг, та список доступних страв.
- **Клас** `Dish` - страва, що містить атрибути, такі як назва, опис, ціна, та специфічні атрибути, наприклад калорійність, алергени.
- **Клас** `Courier` - кур'єр, який має атрибути ім'я, контактні дані, рейтинг, та тип транспорту.
- **Клас** `Client` - клієнт, що містить атрибути ім'я, адреса доставки, контактний номер, та історію замовлень.
- **Клас** `Order` - замовлення, що включає список страв, загальну суму, статус замовлення, інформацію про заклад, кур'єра та клієнта.
- **Клас** `DeliveryManager` - менеджер доставки, який забезпечує логіку обробки замовлення, вибір кур'єра, та відстеження статусу доставки.
- **Клас** `MockTester` - використовується для тестування функціональності системи, включає список доступних клієнтів та об'єкт менеджера доставки.

Завдання включає розробку випадкового вибору закладу та формування випадкового замовлення, систему визначення та призначення кур'єра для замовлення, моделювання процесу доставки, тестові сценарії для перевірки функціональності, обробку помилок та виняткових ситуацій,

додавання власних атрибутів та методів, та імплементацію 3 власних класів.
Через складність з задачею, її реалізація випливатиме крок за кроком, починаючи з реалізації основних класів та методів, далі інтеграції класів між собою для моделювання процесу доставки, і нарешті, добавлення тестових сценаріїв та обробки помилок.

Крок 1. Реалізація класу `Dish`

Почнемо з простішої компоненти - класу `Dish`.

```
#include <iostream>

#include <string>

#include <vector>


// Клас для представлення страви

class Dish {

private:

    std::string name;

    std::string description;

    double price;

    // Додати інші специфічні атрибути, наприклад калорійність, алергени тощо


public:

    // Конструктор

    Dish(std::string name, std::string description, double price)

    : name(name), description(description), price(price) {}
```

```
    // Метод для виведення інформації про страву
    void display() const {
        std::cout << "Страва: " << name << "\nОпис: " << description << "\nЦіна: " << price << " грн\n";
    }
    // Додати інші методи згідно з потребами
};

int main() {
    // Примітка: цей приклад лише демонструє створення одного об'єкта класу Dish
    Dish dish("Піца", "Соковита піца з подвійним сиром і пепероні", 250.0);
    dish.display();
    return 0;
}
```

Цей код демонструє базову структуру класу `Dish` з конструктором та методом для виведення інформації. Наступні кроки включатимуть реалізацію інших класів та інтеграцію їх разом, щоб сформувати функціональну систему доставки.

Давайте продовжимо з реалізацією інших класів для нашої системи доставки на мові C++.

Реалізація класу `Restaurant`

Клас `Restaurant` буде мати інформацію про ресторан, включаючи його назву, адресу, тип кухні, рейтинг, та список страв, доступних для замовлення.

```
class Restaurant {
```

```
private:
    std::string name;
    std::string address;
    std::string type; // Наприклад, італійська, японська кухня
    double rating;
    std::vector<Dish> menu; // Перелік страв доступних у ресторані

public:
    Restaurant(std::string name, std::string address, std::string type, double rating)
    : name(name), address(address), type(type), rating(rating) {}

    void addDish(const Dish& dish) {
        menu.push_back(dish);
    }

    void displayMenu() const {
        std::cout << "Меню \"" << name << "\":\n";
        for(const auto& dish : menu) {
            dish.display(); // Використовуємо метод класу Dish
        }
    }
    // Додати інші методи згідно з потребами
};
```

Реалізація класу `Courier`

Кур'єр буде мати ім'я, контактну інформацію, рейтинг та тип транспортного засобу, який він використовує.

```
class Courier {
private:
    std::string name;
    std::string contactInfo;
    double rating;
    std::string transportType; // Наприклад, велосипед, автомобіль
```

```
public:
    Courier(std::string name, std::string contactInfo, double
rating, std::string transportType)
    : name(name), contactInfo(contactInfo), rating(rating),
transportType(transportType) {}

    void display() const {
        std::cout << "Кур'єр: " << name << "\nКонтакти: " <<
contactInfo << "\nРейтинг: " << rating << "\nТранспорт: " <<
transportType << "\n";
    }
    // Додати інші методи згідно з потребами
};
```

Реалізація класу `Client`

Клієнт матиме ім'я, адресу, контактну інформацію, та історію замовлень.

```
class Client {
private:
    std::string name;
    std::string deliveryAddress;
    std::string contactNumber;
    // Історія замовлень може бути реалізована як список або
вектор ідентифікаторів замовлень

public:
    Client(std::string name, std::string deliveryAddress, std::string
contactNumber)
    : name(name), deliveryAddress(deliveryAddress),
contactNumber(contactNumber) {}

    void display() const {
        std::cout << "Клієнт: " << name << "\nАдреса доставки: "
<< deliveryAddress << "\nКонтактний номер: " <<
contactNumber << "\n";
```

```
    }
    // Додати інші методи згідно з потребами
};
```

Реалізація класу `Order`

Клас `Order` буде включати список замовлених страв, загальну суму, статус замовлення (напр., "у процесі", "доставлено"), інформацію про клієнта, кур'єра, та ресторан.

Оскільки реалізація цього класу залежить від інтеракції з іншими класами та включає розширений набір функцій для управління замовленням, її бажано обдумати більш детально, особливо в контексті взаємодії з класами `DeliveryManager` та `MockTester`.

Реалізація класу `Order`

```
#include <vector>
#include <iostream>

// Попередні класи тут...

class Order {
private:
    std::vector<Dish> dishes;
    Client client;
    Courier courier;
    Restaurant restaurant;
    double totalPrice;
    std::string status;   // Наприклад,
"Processing", "Delivered"

public:
    Order(const std::vector<Dish>& dishes, const
Client& client, const Courier& courier, const
Restaurant& restaurant)
```

```
    : dishes(dishes), client(client),
courier(courier), restaurant(restaurant),
status("Processing") {
        totalPrice = calculateTotal();
    }

    double calculateTotal() const {
        double total = 0;
        for(const auto& dish : dishes) {
            // Припустимо, що Dish клас має метод
getPrice()
            total += dish.getPrice();
        }
        return total;
    }

    void displayOrder() const {
        std::cout << "Замовлення для клієнта: " <<
client.getName() << "\n";
        std::cout << "Ресторан: " <<
restaurant.getName() << "\n";
        std::cout << "Страви:\n";
        for(const auto& dish : dishes) {
            dish.display();
        }
        std::cout << "Загальна ціна: " <<
totalPrice << " грн\n";
        std::cout << "Статус замовлення: " <<
status << "\n";
    }

    void setStatus(const std::string& newStatus) {
        status = newStatus;
    }
```

```
    // Додати інші методи згідно з потребами
};
```

Реалізація класу `DeliveryManager`

Наступним кроком є реалізація класу, який буде керувати логікою обробки замовлення.

```
#include <list>

class DeliveryManager {
private:
    std::list<Order> orders;
    // Потенційно, може включати списки доступних ресторанів, кур'єрів тощо

public:
    void addOrder(const Order& order) {
        orders.push_back(order);
    }

    void processOrders() {
        // Тут може бути логіка вибору кур'єра, оновлення статусу замовлення тощо
    }

    // Додати інші методи по роботі зі замовленнями
};
```

Реалізація тестового класу `MockTester`

Цей клас демонструє використання розроблених компонентів системи.

```
class MockTester {
public:
    void test() {
        // Створіть екземпляри класу Dish, Restaurant, Courier, Client та Order
```

```
        // Додайте створені замовлення до
екземпляру DeliveryManager і викликайте методи для
їх обробки
    }
};
```
Цей огляд дає загальне уявлення про структуру системи доставки та принципи об'єктно-орієнтованого програмування, використовуючи C++. Реалізація включає базові аспекти, як-то створення класів, їх атрибути та методи, але може бути розширена з додатковою логікою обробки, валідації вхідних даних та управління винятковими ситуаціями.