PART A

1.DRY (Don't Repeat Yourself)In your own words, define DRY and provide a short example of repeated code that violates DRY.Briefly explain how you would refactor that code to adhere to DRY.

DRY (Don't Repeat Yourself) means avoiding duplication of code. Instead of repeating the same logic in multiple places, it should be written once and reused.

```
//Example of Repeated Code (Violating DRY):
double calculateTotalWithTax(double price) {
    return price + (price * 0.1);  // 10% tax
}

double calculateTotalWithDiscount(double price) {
    return price - (price * 0.1);  // 10% discount
}

//Refactored Code to Adhere to DRY:
double calculateTotal(double price, double percentage, bool isTax) {
    if (isTax) {
        return price + (price * percentage);  // Calculate tax
    } else {
        return price - (price * percentage);  // Calculate discount
    }
}
```

The function calculateTotal now handles both tax and discount.

2. KISS (Keep It Simple, Stupid):Describe KISS and why it is crucial for maintainable code.Name one potential drawback of oversimplifying code—why might too little complexity sometimes be problematic?

KISS (Keep It Simple, Stupid) means writing code in the simplest way possible without unnecessary complexity. Simple code is easier to understand, fix, and update, which makes it more maintainable and less likely to have bugs.

Why KISS is Important:

Simple code is quicker to understand for anyone who reads it. Less complex code has fewer chances of bugs. It's easier to update and modify simple code.

Potential Drawback of Oversimplifying:

Sometimes, making code too simple can cause problems, like: The code might not handle all situations or requirements properly. And simple solutions might not be efficient enough for bigger or more complex tasks.

3. Introduction to SOLID (High-Level)

The SOLID principles are:Single Responsibility Principle (SRP),Open-Closed Principle (OCP),Liskov Substitution Principle (LSP),Interface Segregation Principle (ISP),Dependency Inversion Principle (DIP).Pick two of these (your choice) and provide a one-sentence explanation of each.Why do the SOLID principles matter in large codebases?

Open-Closed Principle (OCP): You should be able to add new features to a class without changing its existing code. This keeps the current code working while allowing for improvements.

Dependency Inversion Principle (DIP): The main parts of the program should not rely directly on the small, detailed parts. Instead, both should depend on general rules (abstractions), making the system easier to change.

In large projects, SOLID principles help keep the code flexible and easy to update. They make sure that adding new features or fixing bugs won't break other parts of the system, making it easier to maintain and grow over time.

PART B

1. DRY Violation & Fix

Scenario: You have two functions in a class that perform very similar tasks (e.g., sending email and sending text).

Task: Show how you would rewrite or refactor them into a single function or method, removing duplicated logic.

```
//DRY Violation Example:
void sendEmail(string recipient, string message) {
    cout << "Sending email to " << recipient << endl;
    cout << "Message: " << message << endl;
}

void sendText(string recipient, string message) {
    cout << "Sending text to " << recipient << endl;
    cout << "Message: " << message << endl;
}

//Refactored Code (No DRY Violation):
void sendNotification(string recipient, string message, string type) {
    cout << "Sending " << type << " to " << recipient << endl;
    cout << "Message: " << message << endl;
}
```

In the refactored version, both sending email and text are handled by the same method sendNotification, with the type of message specified as a parameter. This eliminates duplicated logic.

2. KISS Principle Example

Scenario: A method that calculates the discount for a product uses overly complex conditional checks.

Task: Sketch or pseudo-code a simpler approach that follows KISS, ensuring the method is easy to read and modify.

```
if (isSale) {
    if (isNewCustomer) {
        discount = price * (purchaseAmount > 5 ? 0.3 : 0.2);
    } else if (isVIP) {
        discount = price * (purchaseAmount > 3 ? 0.25 : 0.15);
    }
} else {
    if (isNewCustomer) discount = price * 0.1;
    else if (isVIP) discount = price * 0.2;
}
```

3. SOLID Application

Scenario: Suppose you have a Shape interface with draw() and computeArea() methods. Circle and Rectangle implement draw() differently, but the same method to compute area.

Task: Show (in pseudo-code or UML) how one relevant SOLID principle (e.g., Single Responsibility or Interface Segregation) applies here.Optional: If you want to illustrate a second principle, you can do so, but keep it minimal.

In simple terms, the Interface Segregation Principle (ISP) says that a class should only be forced to implement the methods it actually needs.In this case, instead of having one big Shape interface with both draw() and computeArea() methods, we split it into two smaller interfaces:Drawable (for drawing shapes),AreaCalculable (for calculating the area)

```
// Smaller interfaces
interface Drawable {
    draw()
}

interface AreaCalculable {
    computeArea()
}

// Circle implements both interfaces
class Circle implements Drawable, AreaCalculable {
    radius: float

    draw() { /* Draw circle */ }
    computeArea() { return 3.14 * radius * radius }
}

// Rectangle implements both interfaces
class Rectangle implements Drawable, AreaCalculable {
    width: float
    height: float

    draw() { /* Draw rectangle */ }
    computeArea() { return width * height }
}
```

This version separates the drawing and area calculation functionality into two distinct interfaces, so each class only implements what it needs.

PART C

1. Trade-Offs:Sometimes code repetition can look like a DRY violation but might be simpler to read.Provide a short scenario where repeating code might be more readable than forcing a “clever” DRY solution.

Imagine you're checking two different types of inputs (an integer and a string) for errors. Each has its own error message.

```cpp
//Repeating Code
void processInteger(int value) {
    if (value < 0) {
        cout << "Value cannot be negative." << endl;
    } else {
        cout << "Value is valid." << endl;
    }
}

void processString(const string& str) {
    if (str.empty()) {
        cout << "String cannot be empty." << endl;
    } else {
        cout << "String is valid." << endl;
    }
}

//DRY Code
void validate(function<bool()> condition, const string& message) {
    if (!condition()) {
        cout << message << endl;
    } else {
        cout << "Input is valid." << endl;
    }
}
```

In the original code, you have two separate functions to validate an integer and a string, which have similar structures but different conditions and error messages. DRY would suggest that you refactor the code to avoid writing the validation logic twice.

2. Combining Principles:How can adhering to both DRY and KISS simultaneously guide your design decisions? Give a brief example.

Example: Vehicle Speed Calculation

```cpp
//Without DRY & KISS
double speedCar = distanceCar / timeCar;
double speedBike = distanceBike / timeBike;
double speedTruck = distanceTruck / timeTruck;


//With DRY & KISS
double calculateSpeed(double distance, double time) {
    return distance / time;
}

// Usage
double speedCar = calculateSpeed(100.0, 2.0);
double speedBike = calculateSpeed(50.0, 1.5);
double speedTruck = calculateSpeed(200.0, 4.0);
```

The speed calculation is repeated for different vehicles, making the code redundant and harder to maintain. A single function calculateSpeed() handles the speed calculation, keeping the code reusable and simple.

3. SOLID in Practice:In a small project or code snippet, is it always necessary to strictly follow every SOLID principle?Why might an early-stage or small codebase not adhere strictly to these guidelines?

For small projects, focusing on simplicity and speed of development is often more important than following all SOLID guidelines. Strict adherence can lead to over-engineering and unnecessary complexity.

```cpp
//Without Strict SOLID (Simple & Practical)
class Vehicle {
public:
    void move() {
        cout << "Vehicle is moving" << endl;
    }
};

// Usage
Vehicle car;
car.move();

//With Strict SOLID
class IVehicle {
public:
    virtual void move() = 0; // Interface for movement
};

class Car : public IVehicle {
public:
    void move() override {
        cout << "Car is moving" << endl;
    }
};
```

Simple and fast :One class and method to move the vehicle. Using SOLID interface (IVehicle) and creating an extra class (Car) adds unnecessary abstraction for a small project.