Part A: Conceptual Questions

1. DRY (Don’t Repeat Yourself)

Definition

- DRY means not repeating the same code or logic in multiple places. If you find yourself copying and pasting code, there’s probably a better way to combine it or reuse it.

Example of repeated code (violation):

```
void printUserInfo1(string name, int age) {
   cout << "Name: " << name << endl;
   cout << "Age: " << age << endl;
}
void printUserInfo2(string name, int age, string email) {
   cout << "Name: " << name << endl;
   cout << "Age: " << age << endl;
   cout << "Email: " << email << endl;
}
```

Refactored version (DRY):

```
void printUserInfo(string name, int age, string email = "") {
   cout << "Name: " << name << endl;
   cout << "Age: " << age << endl;
   if (!email.empty()) {
      cout << "Email: " << email << endl;
   }
}
```

Now I only write the shared code once, and I use an optional parameter to keep it flexible.

2. KISS (Keep It Simple, Stupid)

What it means:

- KISS reminds us not to overcomplicate code. If something can be done in a simple way, don't try to make it too fancy. Simpler code is easier to understand, debug, and maintain.

Drawback of oversimplifying:

- Sometimes if you make code too simple, you lose flexibility or accuracy. For example, a super simple discount rule might not work for all cases in a business and could break with new requirements.

3. SOLID Principles (Two Selected)

- ❖ SRP (Single Responsibility Principle): A class should only have one job or responsibility.

- ❖ OCP (Open-Closed Principle): A class should be open for extension but closed for modification — meaning you can add new features without changing existing code.

Why SOLID matters:

- In big projects, code becomes hard to manage if it's not well-organized. SOLID principles help keep things modular, easier to test, and less likely to break when adding new features.

Part B: Minimal Examples or Scenarios

1. DRY Violation & Fix

Before (duplicated logic):

```
void printAdminDetails(string name, int level) {
    cout << "Admin: " << name << " - Level " << level << endl;
}

void printUserDetails(string name, int level) {
    cout << "User: " << name << " - Level " << level << endl;
}
```

After (DRY fix):

```
void printDetails(string name, int level, string role) {
    cout << role << ": " << name << " - Level " << level << endl;
}
```

Now the function works for both users and admins with one definition.

2. KISS Example

Overcomplicated version:

```
double getDiscount(double price, string category, bool isHoliday) {
    if ((category == "electronics" && !isHoliday) ||
        (category == "clothing" && isHoliday) ||
        (price > 100 && category == "books")) {
        return price * 0.1;
    }
    return 0;
}
```

Simplified version (KISS):

```
double getDiscount(double price, double rate) {
    return price * rate;
}
```

Let the caller handle when to apply which rate. The logic becomes easier to maintain.

3. SOLID Application Example

Scenario: We have a Shape interface.

```
class Shape {
public:
    virtual void draw() = 0;
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing Circle" << endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() override {
        cout << "Drawing Rectangle" << endl;
    }
};
```

Which SOLID Principle?
This shows the Open-Closed Principle — we can add new shapes (like Triangle) without changing the existing code. Everything uses the same draw() method.

If computeArea() is different per shape, we could also apply Interface Segregation by splitting interfaces for drawing and for area calculation.

Part C: Reflection & Short Discussion

1. When Repeating Code Might Be OK

- Let's say I'm writing a short script and need to print three lines of output in two different places. If I only use the code twice, creating a new function might make it harder to follow than just repeating the lines. So in small, simple cases, repeating can be more readable.

2. Combining DRY and KISS

- Both principles help you avoid bloated code. DRY stops you from copying logic everywhere, and KISS reminds you not to over-engineer your solution. For example, instead of making five functions to handle user types, I might write one printUser() function and pass in a role — fewer lines, no repetition, and easy to follow.

3. SOLID in Small Projects

- In a small project, you don't always need to apply every SOLID principle strictly. It might make the code more complex than it needs to be. But it's good to keep the ideas in mind because they help when the project starts growing or needs new features.