Name: Esther Ajuwon

Basic Design Principles in Software Development

PART A: Conceptual Questions

1. DRY(Don't Repeat Yourself)

DRY(Don't Repeat Yourself) is a programming approach that emphasizes reducing code duplication by centralizing logic such that each piece of knowledge or behavior appears only once.

Example of Code that Violates DRY

```
int area1 = length1 * width1;
int area2 = length2 * width2;
int area3 = length3 * width3;
```

Refactored Version

```
int calculateArea(int length, int width) {
   return length * width;
}

int area1 = calculateArea(length1, width1);
int area2 = calculateArea(length2, width2);
int area3 = calculateArea(length3, width3);
```

It removes repeated logic, making the code easier to maintain, test, and update if the area calculation ever changes.

2. KISS(Keep it Simple, Stupid)

KISS (Keep it Simple, Stupid) is a design approach that encourages developers to build code that is as basic and direct as possible, minimizing extraneous complexities. It is critical for maintainable code since simple code is easier to comprehend, debug, test, and update, particularly by developers who are not original authors.
Oversimplifying code, on the other hand, might be harmful if necessary structure or flexibility are ignored. For example, disregarding design patterns or abstractions in a large system can result in repetitive code, poor scalability, or rigid designs that are difficult to adapt in the future. Simplicity should never be at the expense of clarity or long-term adaptability.

3. Introduction to SOLID

Single Responsibility Principle (SRP):

A Child should only have one reason to change, which means it should handle a single job or duty.

Open-Closed Principle (OCP):
Software entities should be extensible but not modifiable, allowing for new behavior while preserving old code.
Why SOLID matters: The SOLID principles encourage clean, modular, and scalable architecture, which is critical in large codebases to reduce errors, improve collaboration, and make future modifications safer and easier to manage.

PART B: Scenarios & Task

1. DRY Violation & Fix

Before	After

void printUserBasic(User u) {
  cout << "Name: " << u.name << endl;
  cout << "Email: " << u.email << endl;
}
 	
void printUserDetailed(User u) {
  cout << "Name: " << u.name << endl;
  cout << "Email: " << u.email << endl;
  cout << "Phone: " << u.phone << endl;
}
	void printUser(User u, bool detailed = false) {
	  cout << "Name: " << u.name << endl;
	  cout << "Email: " << u.email << endl;
	  if (detailed) cout << "Phone: " << u.phone << endl;
	}

2. KISS Principle Example

Before	After

double getDiscount(double price, string category) {
  if (category == "electronics" && price > 1000 || category == "furniture" && price > 500) {
  return price * 0.10;
  } else if (category == "clothing" && price > 100) {
  return price * 0.05;
  } else {
  return 0;
  }
}
	double getDiscount(double price, double threshold, double rate) {
	  return (price > threshold) ? price * rate : 0;
	}

3. SOLID Application

Problem: Shape has draw() and computeArea() but draw() varies while computeArea() is the same.

.cpp

```
interface Drawable {
   void draw();
}

interface Measurable {
   double computeArea();
}

class Circle : public Drawable, public Measurable {
   void draw() override { /* draw circle */ }
   double computeArea() override { /* same for all shapes */ }
}

class Rectangle : public Drawable, public Measurable {
   void draw() override { /* draw rectangle */ }
   double computeArea() override { /* same for all shapes */ }
}
```

SRP:Separate interfaces for distinct responsibilities.

ISP (Interface Segregation Principle): Clients use only what they need (e.g., some may only care about area, not drawing).

PART C: Reflection & Short Discussion

1. Trade-Offs

When dealing with explicit error warnings for distinct input fields in a form, repeating code may be more understandable than using a smart DRY solution.

.code

```
if (username.empty()) {
   cout << "Username is required." << endl;
}
if (email.empty()) {
   cout << "Email is required." << endl;
}
if (password.empty()) {
   cout << "Password is required." << endl;
```

}

While looping through fields and messages with maps or arrays might save duplication, it may make the code more difficult to read and debug—especially for others. In this case, repetition increases clarity and makes each condition more understandable.

2. Combining Principles

Following both DRY and KISS guidelines allows you to develop code that avoids unnecessary repetition while not becoming unnecessarily abstract or difficult. They work together to ensure that you only extract reusable logic when it simplifies and clarifies the design.

```
double calculateTax(double amount, double rate = 0.07) {
   return amount * rate;
}
```

3. SOLID in Practice

A tiny project or code snippet does not always require rigorous adherence to every SOLID guideline. These ideas are especially beneficial in bigger, growing codebases where maintainability, scalability, and team collaboration are critical.

In early-stage or tiny codebases, rigid adherence might result in overloading, extra complexity, and slower development. At that level, simplicity, clarity, and iteration speed are frequently prioritized. It is usually preferable to begin simple then revise toward SOLID as the project progresses and patterns emerge that warrant more structured design.