Turram Toussaint
2143 OOP
Prof Griffin

Part A: Conceptual Questions
DRY (Don’t Repeat Yourself)
Definition:
The DRY principle emphasizes that each piece of knowledge or logic should exist in a single place in your code. The goal is to reduce repetition, making the code more maintainable and less error-prone.

Example of Repeated Code:

```
class User {
   public:
      void printUserDetails() {
         cout << "Name: " << name << endl;
         cout << "Email: " << email << endl;
      }

      void printAdminDetails() {
         cout << "Name: " << name << endl;
         cout << "Email: " << email << endl;
         cout << "Role: Admin" << endl;
      }
};
```

Refactor to Adhere to DRY:

```
class User {
   public:
      void printDetails(bool isAdmin = false) {
         cout << "Name: " << name << endl;
         cout << "Email: " << email << endl;
         if (isAdmin) {
            cout << "Role: Admin" << endl;
         }
      }
};
```

Now, the repeated logic is encapsulated in a single method that can handle both user and admin details.

KISS (Keep It Simple, Stupid)
Definition:
The KISS principle emphasizes simplicity in design and code. It encourages avoiding unnecessary complexity and keeping solutions as straightforward as possible. Simple code is easier to maintain, test, and understand.

Why It’s Crucial:
Simplifying code helps developers avoid confusion and reduces the chances of introducing bugs. It makes future modifications and debugging easier.

Drawback of Oversimplifying:
If code is overly simplistic, it may lack the necessary flexibility or functionality to handle all future use cases. For instance, a simple method for discount calculation might not be able to handle complex business rules in the future.

Introduction to SOLID Principles
Single Responsibility Principle (SRP):
A class should have only one reason to change, meaning it should have only one responsibility. This helps in creating classes that are focused and easier to maintain.

Open-Closed Principle (OCP):
Software entities (classes, modules, functions) should be open for extension but closed for modification. This promotes adding new functionality without altering existing code.

Why SOLID Principles Matter in Large Codebases:
SOLID principles help to reduce coupling, improve maintainability, and ensure that the system is flexible for future enhancements. They lead to more readable, extensible, and reusable code, which is crucial when managing large, complex projects.

Part B: Minimal Examples or Scenarios
DRY Violation & Fix
Scenario:
You have two functions for printing user and admin details.

Violation:

```
class User {
   public:
      void printUserDetails() {
         cout << "Name: " << name << endl;
         cout << "Email: " << email << endl;
      }

      void printAdminDetails() {
         cout << "Name: " << name << endl;
         cout << "Email: " << email << endl;
         cout << "Role: Admin" << endl;
      }
};
```

Refactor:

```
class User {
   public:
      void printDetails(bool isAdmin = false) {
```

```
        cout << "Name: " << name << endl;
        cout << "Email: " << email << endl;
        if (isAdmin) {
            cout << "Role: Admin" << endl;
        }
    }
};
```
This removes the duplicate code by using a parameter to differentiate between the two behaviors.

KISS Principle Example
Scenario:
A method calculating the discount for a product has overly complex conditional checks.

Overly Complex Method:

```
double calculateDiscount(Product p) {
    if (p.isMember() && p.purchaseAmount > 100) {
        if (p.purchaseAmount > 200) {
            return p.purchaseAmount * 0.2;  // 20% discount
        } else {
            return p.purchaseAmount * 0.1;  // 10% discount
        }
    } else if (p.isMember()) {
        return p.purchaseAmount * 0.05;  // 5% discount
    } else {
        return 0;  // No discount
    }
}
```
Simplified Method (KISS):

```
double calculateDiscount(Product p) {
    double discountRate = 0;
    if (p.isMember()) {
        if (p.purchaseAmount > 200) {
            discountRate = 0.2;  // 20% discount
        } else if (p.purchaseAmount > 100) {
            discountRate = 0.1;  // 10% discount
        } else {
            discountRate = 0.05;  // 5% discount
        }
    }
    return p.purchaseAmount * discountRate;
}
```
This simplifies the code, making it more readable and easier to modify.

SOLID Application
Scenario:

You have a Shape interface with draw() and computeArea() methods, and Circle and Rectangle implement draw() differently but share the same computeArea() method.

SOLID Principle: Interface Segregation Principle (ISP)

```
// Interface Segregation Principle: Separate interfaces for different behaviors
class Drawable {
   public:
      virtual void draw() = 0;
};

class AreaComputable {
   public:
      virtual double computeArea() = 0;
};

class Circle : public Drawable, public AreaComputable {
   public:
      void draw() { /* Circle-specific drawing code */ }
      double computeArea() { return 3.14 * radius * radius; }
};

class Rectangle : public Drawable, public AreaComputable {
   public:
      void draw() { /* Rectangle-specific drawing code */ }
      double computeArea() { return width * height; }
};
```

This adheres to the Interface Segregation Principle by splitting the responsibilities of drawing and computing area into separate interfaces.

Part C: Reflection & Short Discussion
Trade-Offs
Scenario:
You have two similar methods to handle user input in different parts of a program. Instead of creating a generic solution, repeating the code keeps it simple and readable.

Example:
In a small application, repeating input handling logic in two places (without trying to generalize it into one method) may make the code easier to understand and modify quickly. For instance, handling user login in two places may not need a generalized method if the login logic is simple and unlikely to change often.

Combining Principles
Example:
Adhering to both DRY and KISS can help guide design decisions by ensuring that code is neither unnecessarily complex nor repetitive. For example, creating a method to calculate the total price for a

product, where the logic is written in one place, will be simpler and more maintainable than having multiple places where similar logic is duplicated.

SOLID in Practice
In Small Projects:
In a small project or code snippet, it's not always necessary to strictly follow every SOLID principle. Early-stage or small codebases often evolve quickly, and applying all principles may introduce unnecessary overhead or be over-engineered for the project’s scope.

Why Not Always Apply SOLID Strictly:
In smaller projects, the code may be more straightforward, and adhering to every SOLID principle can be time-consuming and might complicate the design without offering substantial benefits. As the project grows, however, following these principles can significantly improve maintainability.