Part A: Conceptual Questions

1. DRY (Don't Repeat Yourself)
    - Definition: DRY emphasizes reducing code repetition by consolidating repeated logic into reusable structures (functions, classes, etc.), ensuring maintainability and scalability.
    - Example of Violation:

```
void printUserInfo1() {
   cout << "User: Zeke" << endl << "Age: 20" << endl;
}

void printUserInfo2() {
   cout << "User: Zeke" << endl << "Age: 20" << endl << Location: Texas" << endl;
}
```

    - Refactor to Adhere to DRY:

```
void printUserInfo(bool includeLocation = false) {
   cout << "User: Zeke << endl << Age: 20" << endl;
   if (includeLocation) {
      cout << "Location: Texas" << endl;
   }
}
```

2. KISS (Keep It Simple, Stupid)
    - Definition: KISS promotes simplicity in design and implementation to enhance readability, maintainability, and reduce potential errors. The idea is to aim for the simplest solution without unnecessary complexity.
    - KISS is crucial because simple code is easier to read, debug, and update—an essential quality for collaboration and long-term projects.
    - Drawback of Oversimplifying: If code is too simple, it may lack flexibility or modularity, leading to challenges in handling future requirements or edge cases.

3. Introduction to SOLID
    - Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should serve a single responsibility.
    - Liskov Substitution Principle (LSP): Objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program.
    - SOLID Principles Matter because they ensure that codebases remain flexible, maintainable, and scalable as they grow. In large codebases, adhering to these principles helps avoid tight coupling and makes individual modules more testable.

Part B: Minimal Examples or Scenarios

1. DRY Violation & Fix
    - Scenario: printing user details with slight variations.
    - Before Refactor:

```
void printName() {
   cout << "Name: Zeke" << endl;
}

void printAge() {
   cout << "Age: 25" << endl;
}
```

    - Refactored:

```
void printUserDetails(const string& name, int age) {
   cout << "Name: " << name << endl;
   cout << "Age: " << age << endl;
}
```

2. KISS Principle Example
    - Scenario: Overly complex discount calculation.
    - Before Simplification:

```
Function calculateDiscount(price, userType)
   If userType is "Student" AND price > 50
      Return price * 0.2
   Else If userType is "Senior" AND price > 30
      Return price * 0.15
   Else If price > 100
      Return price * 0.1
   End If
   Return 0
End Function
```

    - Simplified:

```
Function calculateDiscount(price, discountRate)
   Return price * discountRate
End Function
```

3. SOLID Application
    - Scenario: Shape Interface and its Implementation

- Applying Single Responsibility Principle (SRP):

```
Class Shape
   Method draw() is abstract
   Method computeArea() is abstract
End Class

Class Circle inherits from Shape
   Method draw()
      Output "Drawing Circle"
   End Method
   Method computeArea()
      Return 3.14 * radius * radius
   End Method
End Class

Class Rectangle inherits from Shape
   Method draw()
      Output "Drawing Rectangle"
   End Method
   Method computeArea()
      Return width * height
   End Method
End Class
```

Part C: Reflection & Short Discussion

1. Trade-Offs
    - Sometimes repeating code improves readability. For example:

```
if (userType == "Admin") {
   // Admin-specific logic
} else if (userType == "Guest") {
   // Guest-specific logic
}
```

    - Trying to condense this into a single function or structure might make the code harder to follow, defeating the purpose of DRY.

2. Combining DRY and KISS
    - Adhering to both ensures reusable, readable code.
    - Example: Use a helper function to centralize discount calculations, but keep the helper simple and readable.

3. SOLID in Practice
    - In small projects, strict adherence might introduce unnecessary complexity.
    - Early-stage projects often prioritize quick iteration over long-term structure. As the codebase grows, these principles gradually become essential for maintenance and scalability.