Part A: Conceptual Questions

1. **Dry (Don't Repeat Yourself)**
   - DRY is a principle that aims to reduce code duplication by ensuring that a piece of logic is written only once.
   - Example of code that violates DRY:

```
int main() {
    int width1 = 5, height1 = 10;
    cout << "Area of rectangle 1: " << width1 * height1 << endl;

    // Repeated code for finding area of rectangle
    int width2 = 8, height2 = 12;
    cout << "Area of rectangle 2: " << width2 * height2 << endl;

    return 0;
}
```

   - To refactor the code so that it adheres to DRY, you could encapsulate the repeated logic into a reusable function.
2. **KISS (Keep It Simple, Stupid)**
   - KISS is a principle that emphasizes simplicity in software by writing clear, concise, and minimal code. KISS is crucial for maintainable code as it reduces code complexity, making it easier to debug and update.
   - One drawback of oversimplifying code is that it can sacrifice scalability and flexibility due to unhandled edge cases, complex workflows, etc.
3. **Introduction to SOLID (High-Level)**
   - The Single Responsibility Principle (SRP) states that every class should have only one reason to change, meaning it should have a single, focused responsibility.
   - The Open-Closed Principle (OCP) states that entities like classes or functions should be open for extension but closed for modification, allowing new functionality without altering existing code.
   - SOLID principles matter in large codebases because they promote scalable, maintainable, and flexible software design through improved readability, ease of maintenance, encouraged reusability, etc.

Part B: Minimal Examples or Scenarios

1. **Dry Violation & Fix**

```
class User {
public:
   void printUserDetails(string firstName, string lastName, int id = -1) {
      if (id != -1) {
         cout << "ID: " << id << ", ";
      }
      cout << "Full Name: " << firstName << " " << lastName << endl;
   }
};
```

2. **KISS Principle Example**

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

3. **SOLID Application**

```
//Follow ISP Principle
class Drawable {
public:
   virtual void draw() = 0;
};

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

class Circle : public Drawable, public AreaComputable {
   void draw() override { /* Circle-specific drawing */ }
   double computeArea() { /* Compute Circle area */ }
};

class Rectangle : public Drawable, public AreaComputable {
   void draw() override { /* Rectangle-specific drawing */ }
   double computeArea() { /* Compute Rectangle area */ }
};
```

Part C: Reflection & Short Discussion

1. **Trade-Offs**

```
void logStartup() {
cout << "System is starting up..." << endl;
cout << "Initialization complete." << endl;
}

void logShutdown() {
cout << "System is shutting down..." << endl;
cout << "Cleanup complete." << endl;
}
```

   - While this code may appear like it violates DRY, combining this logic into one function may be overly abstract or complex.

2. **Combining Principles**
   - Adhering to both DRY and KISS together ensures simplicity while avoiding redundancy.
   - Example: If you're developing an online store's discount system, follow DRY by using a generic function for calculating discounts based on user type and purchase amount, and follow KISS by keeping the conditions clear and avoiding overly clever abstractions or nested loops.
3. **SOLID in Practice**
   - Strict adherence to SOLID principles isn't always necessary for small projects and code snippets.
   - Simple designs and fewer dependencies make strict SOLID implementation unnecessary.