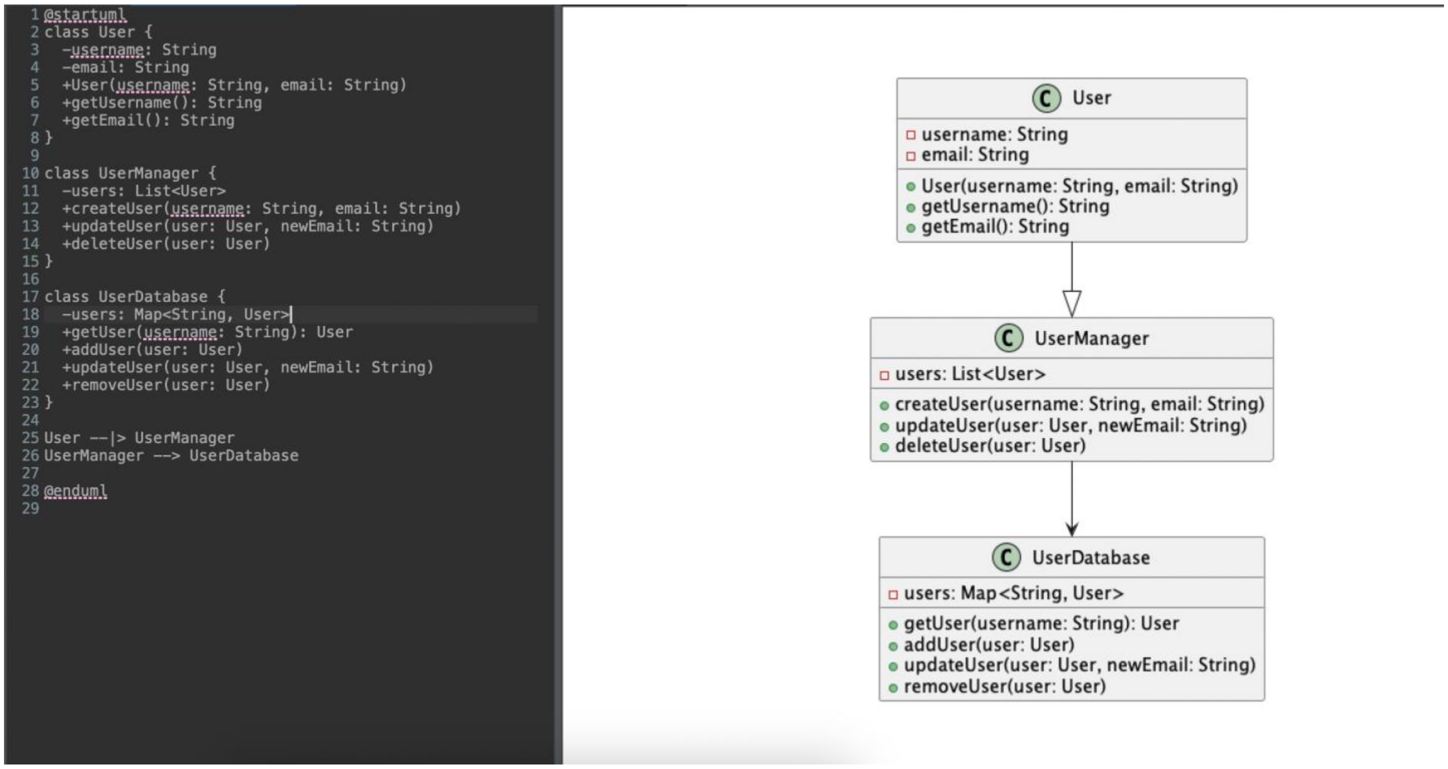
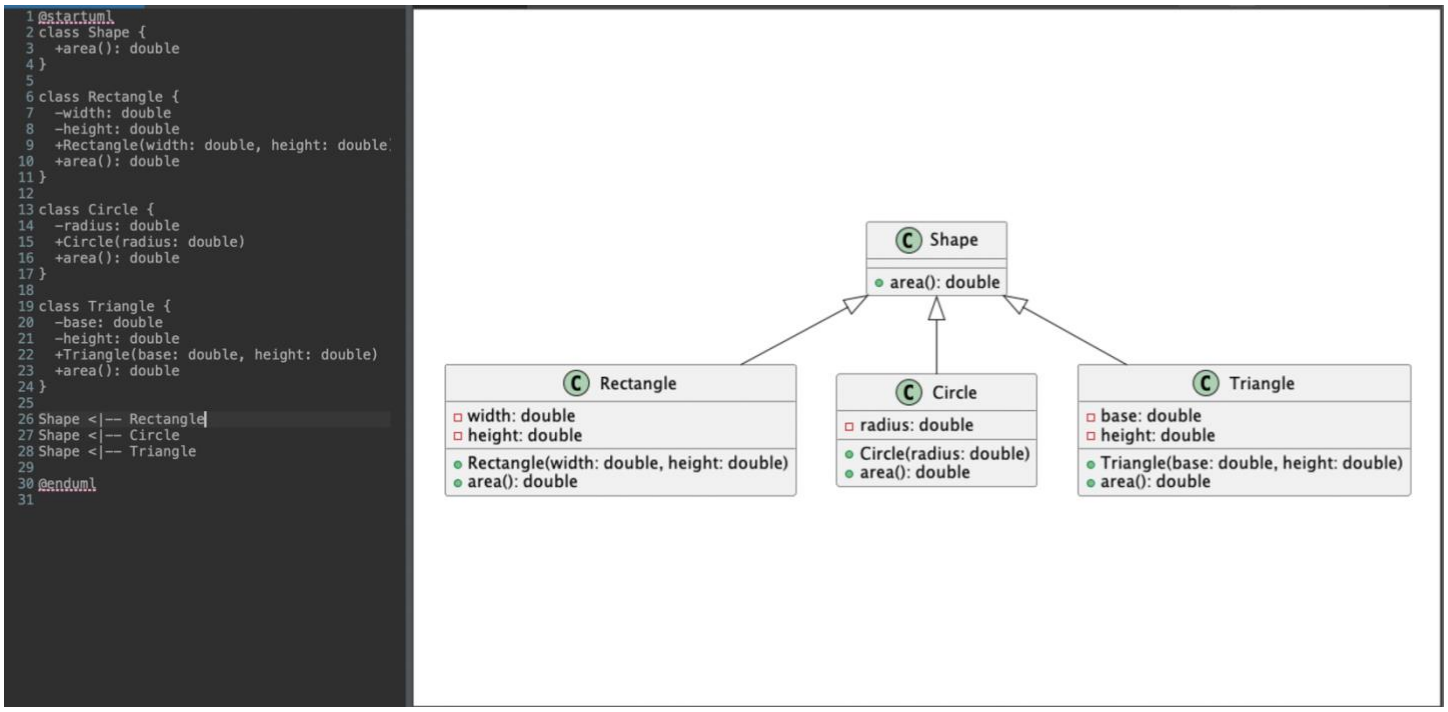
Exercise 1:

Single Responsibility Principle(SRP)



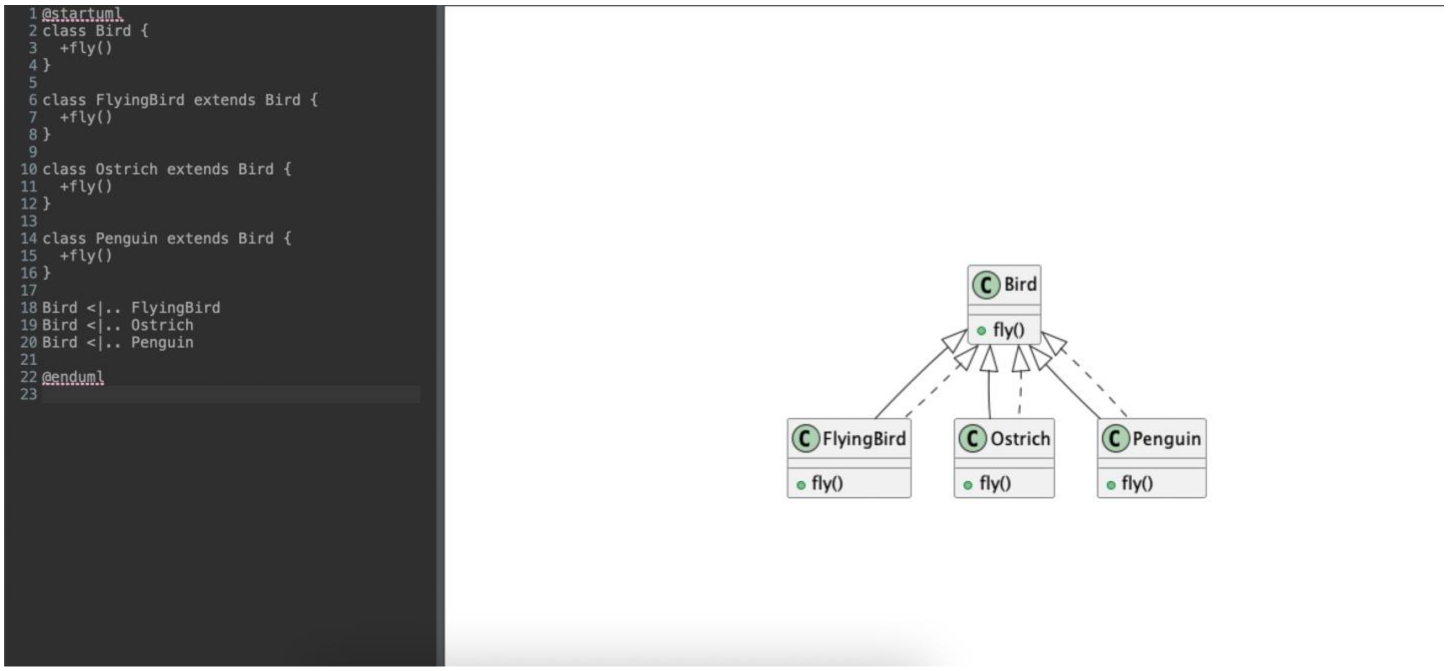
```
@startuml
class User {
  -username: String
  -email: String
  +User(username: String, email: String)
  +getUsername(): String
  +getEmail(): String
}

class UserManager {
  -users: List<User>
  +createUser(username: String, email: String)
  +updateUser(user: User, newEmail: String)
  +deleteUser(user: User)
}

class UserDatabase {
  -users: Map<String, User>
  +getUser(username: String): User
  +addUser(user: User)
  +updateUser(user: User, newEmail: String)
  +removeUser(user: User)
}

User --|> UserManager
UserManager --> UserDatabase

@enduml
```

Open/Closed Principle(OCP)



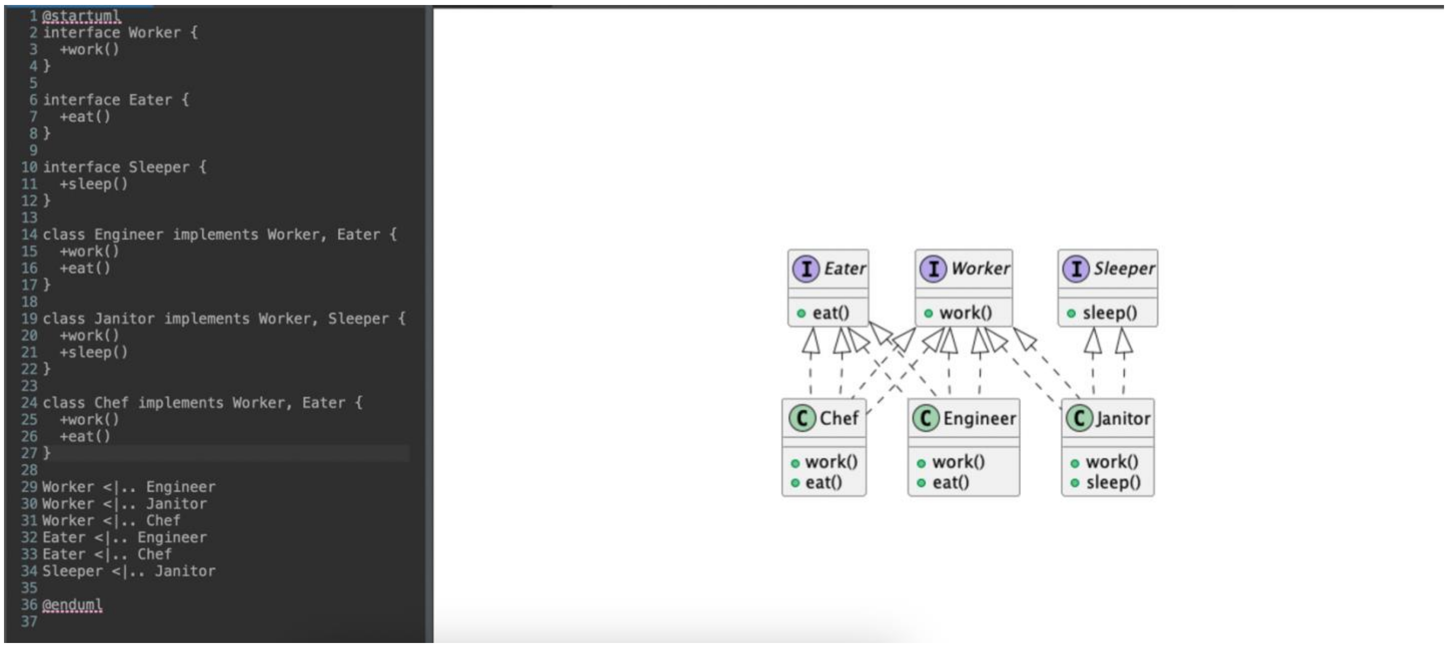
```
@startuml
class Shape {
  +area(): double
}

class Rectangle {
  -width: double
  -height: double
  +Rectangle(width: double, height: double
  +area(): double
}

class Circle {
  -radius: double
  +Circle(radius: double)
  +area(): double
}

class Triangle {
  -base: double
  -height: double
  +Triangle(base: double, height: double)
  +area(): double
}

Shape <|-- Rectangle
Shape <|-- Circle
Shape <|-- Triangle

@enduml
```

Liskov Substitution Principle (LSP)



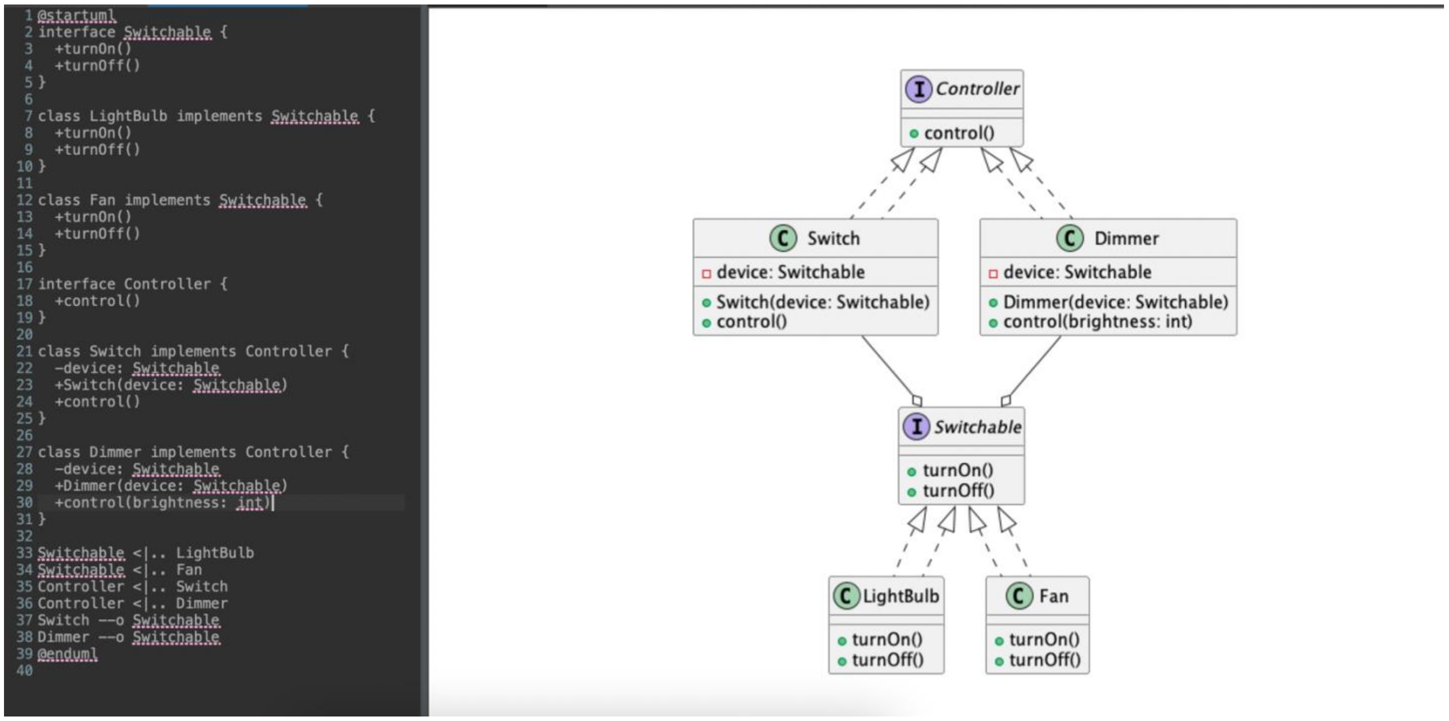
```
@startuml
class Bird {
  +fly()
}

class FlyingBird extends Bird {
  +fly()
}

class Ostrich extends Bird {
  +fly()
}

class Penguin extends Bird {
  +fly()
}

Bird <|.. FlyingBird
Bird <|.. Ostrich
Bird <|.. Penguin

@enduml
```

Interface Segregation Principle (ISP)



```
@startuml
interface Worker {
  +work()
}

interface Eater {
  +eat()
}

interface Sleeper {
  +sleep()
}

class Engineer implements Worker, Eater {
  +work()
  +eat()
}

class Janitor implements Worker, Sleeper {
  +work()
  +sleep()
}

class Chef implements Worker, Eater {
  +work()
  +eat()
}

Worker <|.. Engineer
Worker <|.. Janitor
Worker <|.. Chef
Eater <|.. Engineer
Eater <|.. Chef
Sleeper <|.. Janitor

@enduml
```

Dependency Inversion Principle (DIP)



```
@startuml
interface Switchable {
  +turnOn()
  +turnOff()
}

class LightBulb implements Switchable {
  +turnOn()
  +turnOff()
}

class Fan implements Switchable {
  +turnOn()
  +turnOff()
}

interface Controller {
  +control()
}

class Switch implements Controller {
  -device: Switchable
  +Switch(device: Switchable)
  +control()
}

class Dimmer implements Controller {
  -device: Switchable
  +Dimmer(device: Switchable)
  +control(brightness: int)
}

Switchable <|.. LightBulb
Switchable <|.. Fan
Controller <|.. Switch
Controller <|.. Dimmer
Switch --o Switchable
Dimmer --o Switchable
@enduml
```

Exercise 2:

Single Responsibility Principle

```
// User represents user data.
class User {
   private String username; // private Instant variables.
   private String email;

   public User(String username, String email) { // Constructor functions
      this.username = username;
      this.email = email;
   }

   // Getters and setters for username and email to make it accessible
}

class UserManager {
   private List<User> users = new ArrayList<>();

   public void createUser(String username, String email) {
      User user = new User(username, email);
      users.add(user);
   }
```

```
    public void updateUser(User user, String newEmail) {
        user.setEmail(newEmail);
    }

    public void deleteUser(User user) {
        users.remove(user);
    }
}

class UserDatabase { // Manages user data storage.
    private Map<String, User> users = new HashMap<>();

    public User getUser(String username) {
        return users.get(username);
    }

    public void addUser(User user) {
        users.put(user.getUsername(), user);
    }

    public void updateUser(User user, String newEmail) { // Update users’ email
        user.setEmail(newEmail);
    }

    public void removeUser(User user) { // Remove user database
        users.remove(user.getUsername());
    }
}
```

Open/Close Principle(OCP)

```
// Abstract Shape class with area calculation
abstract class Shape {
    public abstract double area();
}
//Rectangle extends Shape
class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }
}
// Circle extends Shape for area of circle.
```

```
class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

Liskov Substition Principle(LSP)

```
// Bird with a fly method.
class Bird {
    public void fly() {
        System.out.println("Bird is flying.");
    }
}

// Flying bird extends Bird
class FlyingBird extends Bird {
    @Override
    public void fly() {
        System.out.println("Flying bird is flying.");
    }
}

class Ostrich extends Bird {
    // Ostrich doesn't fly, therefore no method implemented.
}

class Penguin extends Bird {
    // Penguin doesn't fly, therefore no method implemented.
}
```

Interface Segregation Principle (ISP)

```
//Worker Interface.
interface Worker {
    void work();
}

//Engineer implements Worker with work methods.
class Engineer implements Worker {
    @Override
    public void work() {
        System.out.println("Engineer is working.");
    }
}

class Janitor implements Worker {
    @Override
    public void work() {
        System.out.println("Janitor is working.");
    }
}

// Chef class implements Worker.
class Chef implements Worker {
    @Override
    public void work() {
        System.out.println("Chef is working.");
    }
}
```

Dependency Inversion Principle (DIP)

```
// Swichable interface turnOn

interface Switchable {
    void turnOn();

    void turnOff();
}

// LightBulb implements Switchable
class LightBulb implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("LightBulb is turned on.");
    }

    @Override
    public void turnOff() {
        System.out.println("LightBulb is turned off.");
    }
}

class Fan implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("Fan is turned on.");
    }

    @Override
    public void turnOff() {
        System.out.println("Fan is turned off.");
    }
}
```

```
interface Controller {
   void control();
}

class Switch implements Controller {
   private Switchable device;

   public Switch(Switchable device) {
      this.device = device;
   }

   @Override
   public void control() {
      device.turnOn();
   }
}

// Dimmer implements Controller and Switchable.
class Dimmer implements Controller {
   private Switchable device;

   public Dimmer(Switchable device) {
      this.device = device;
   }

   @Override
   public void control() {
      device.turnOn();
   }
}
```

Exercise 3: Use Cases and Why Apply the SOLID Principles

Single Responsibility Principle (SRP) - User Authentication System

SRP can be used in user authentication systems.
The User Class: The User class is this case uses a single responsibility in representing user data as shown the diagram. This class focuses on storing and providing access to user-related information, which are retrieved from the Database. Users can authenticate and sign-in to the program.

UserManager Class: The UserManager class has the responsibility of managing user operations across each of the domains. These operations include creating, updating, and deleting user records from the database. Separating user management from the user data representation follows an SRP, as such this makes the code more maintainable over time with ease without breaking the main components of the applications.

UserDatabase Class: The UserDatabase class is responsible for data storage and retrieval. As such it is not burdened with user management logic, such as user creation or deletion process. Keeping data storage and management separate aligns with SRP implementations and goals of the solid principle.

Why use the Single Responsibility Principle Over Others:
While the other SOLID principles play important roles, SRP is the most relevant in this case because it ensures that each class has a single reason to change and is responsible for one specific aspect of the system. Separating concerns leads to code that is easier to understand, maintain, and extend, as such making the code DRY. It makes it easier to integrate other applications and make updates and changes without pulling down and entire code base.

Open/Closed Principle (OCP) - Extensible Shape Calculation

In a geometric shape calculation system, OCP is a key principle in this kind or calculations:
Shape Class: The Shape class defines a common interface for shapes, allowing the addition of new shapes without modifying existing code. New shapes can extend this class and provide their own implementation of the area method in the sample code.

Rectangle and Circle Classes: The Rectangle and Circle classes extend the Shape class to provide their area calculation logic which is important across the methods. They are open for extension because they allow for new shapes to be added without altering their code base.

Why use OCP Over Others: In this case, OCP is the primary principle because it allows for the system to be easily extensible. Without modifying the existing code base, new shapes can be added, and the Shape class acts as an abstraction that enables this extensibility and flexibility.

Liskov Substitution Principle (LSP) - Bird Hierarchy.

In a system representing birds, LSP is vital for maintaining consistent behavior and functionality:
Bird Class: The base Bird class defines common behaviors for all birds, such as the `fly` method. Derived classes must adhere to these behaviors to ensure substitutability.

FlyingBird, Ostrich, and Penguin Classes: The FlyingBird class extends the Bird class and correctly overrides the fly method, indicating its ability to fly. In contrast, the Ostrich and Penguin classes also extend Bird but don't override the fly method, correctly reflecting their inability to fly as decribed.

Why the LSP Over Others: LSP is crucial in this case because it ensures that derived classes can be used interchangeably with their base class. It enforces a contract that all subclasses must adhere to, providing a consistent and predictable interface across different bird types.

Interface Segregation Principle (ISP) - Workers in a Company

In a system managing workers with various roles, ISP helps create focused interfaces:
Worker Interface: The Worker interface defines a work method, which all worker roles must implement. It allows each worker type to have a specific role-related implementation.

Engineer, Janitor, and Chef Classes: Each class implements the Worker interface and provides its own implementation of the work method, reflecting the unique responsibilities of each worker role.

Why the ISP Over Others: ISP is the most relevant principle in this context because it ensures that classes do not implement unnecessary methods. Each worker type implements only the methods relevant to their specific role, making the code cleaner and preventing dependencies on unused methods.

Dependency Inversion Principle (DIP) - Home Automation System

In a home automation system controlling various devices, DIP helps achieve flexibility:
Switchable Interface: The Switchable interface defines common methods (turnOn and turnOff) for controllable devices. This allows high-level controllers to depend on abstractions, not concrete implementations.

LightBulb and Fan Classes: Both classes implement the Switchable interface, allowing them to be controlled by high-level controllers without modifying the controllers' code.

Switch and Dimmer Classes: These high-level controller classes depend on the Switchable interface, adhering to DIP. They can control any device that implements Switchable, offering flexibility for future device additions. As such this method is crucial in these kinds of operations.

Why the DIP Over Others: DIP is highly applicable here because it allows for loose coupling and the ability to add new device types without altering existing high-level controller code base. By depending on abstractions, the system becomes highly extensible, and easier to make and deploy.