SOLID Principles Documentation:

1. Single Responsibility Principle (SRP):

The Single Responsibility Principle states that a class should have only one reason to change. Each class should have one responsibility or should encapsulate a single functionality.

- Observation:
    - The `Celeb` class represents a celebrity with attributes like `celebId` and `name`.
    - `Actor` and `Director` extend the `Celeb` class, adding specific functionalities related to actors and directors.
- Adherence:
    - `Celeb` class handles basic celebrity information.
    - `Actor` and `Director` classes handle specific details related to actors and directors, respectively.

```
package com.ilp.entity;
public class Celeb {
private String celebId;
private String name;
public Celeb(String celebId, String name) {
        super();
        this.celebId = celebId;
        this.name = name;
}
//getters and setters
}
```

2. Open/Closed Principle (OCP):

The Open/Closed Principle states that a class should be open for extension but closed for modification. Code should allow for future enhancements without modifying existing code.

- Observation:
  - The `Actor` and `Director` classes extend the `Celeb` class.
  - Interfaces `ActorRemuneration` and `Celebrity` are used for specific functionalities without modifying existing classes.
- Adherence:
  - New types of celebrities or functionalities can be added by extending existing classes or implementing new interfaces.

```java
//Celebs is open to extension instead of modifications

public class Director extends Celeb implements Celebrity {
        public Director(String celebId, String name, String
assistants) {
                super(celebId, name);
                this.setAssistants(assistants);
        }

        //getters and setters

}
```

3. Liskov Substitution Principle (LSP):

The Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

- Observation:
  - The `Celeb` reference is used to point to instances of both `Actor` and `Celeb` in the `Main` class.
- Adherence:
  - `Celeb`, `Actor`, and `Director` can be used interchangeably, demonstrating Liskov Substitution.

```
        Celeb celeb = new Actor("A1111", "Ramu"); // Liskov '
        substitution principle
```

4. Interface Segregation Principle (ISP):

The Interface Segregation Principle states that a class should not be forced to implement interfaces it does not use. It encourages creating smaller, specific interfaces.

- Observation:
    - `Actor` and `Director` implement specific interfaces (`ActorRemuneration` and `Celebrity`) to provide relevant functionalities.
- Adherence:
    - Each class implements only the interfaces that are relevant to its responsibilities.

```
public class Actor extends Celeb implements
ActorRemuneration,Celebrity {
  public Actor(String celebId, String name) {
          super(celebId, name);


      }
      private ArrayList<String> actedMovies;
          @Override
      public void displayRemuneration() {

           System.out.println("Actor Remuneration : 2 lakhs per
day");
       }
      @Override
      public void displayDateOfBirth() {
          System.out.println("Date Of Birth : 22/11/1987");
      }}
```

5. Dependency Inversion Principle (DIP):

The Dependency Inversion Principle states that high-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

- Observation:
  - The `Actor`, `Director` class depends on abstractions rather than concrete implementations.
- Adherence:
  - The code follows dependency inversion by relying on interfaces and abstract classes, allowing for flexibility and ease of maintenance.

```
//classes depend on a high level module-like interface rather
than concrete implementations of the methods, hence
dependency inversion principle is also implemented.
public class Director extends Celeb implements Celebrity {
	public Director(String celebId, String name, String
assistants) {
		super(celebId, name);
		this.setAssistants(assistants);
	}

	private String assistants;
	@Override
	public void displayDateOfBirth() {
		System.out.println("Date Of Birth : 12/12/2012");
	}

	}


}
```

Summary:

The given code adheres to SOLID principles by promoting maintainability, extensibility, and flexibility. It demonstrates good design practices by encapsulating responsibilities, allowing for easy extension, supporting polymorphism, and relying on abstractions.