1. Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should have only one job or responsibility.

Example:

```
class Book {
   private String title;
   private String author;
// Constructor, getters, and other book-related methods
}
class BookPrinter {
   public void printBook(Book book) {
      // Code to print the book details
   }
}
```

Explanation: The Book class is only responsible for book-related data, and the BookPrinter class is responsible for printing the book. This separation ensures that changes to printing logic do not affect the Book class.

2. Open/Closed Principle (OCP)

Definition: Software entities should be open for extension but closed for modification.

Example:

```
abstract class Shape {
   abstract void draw();
}
class Circle extends Shape {
   void draw() {
      // Drawing logic for circle
   }
}
class Rectangle extends Shape {
   void draw() {
      // Drawing logic for rectangle
   }
}
class Drawing {
   private List<Shape> shapes = new ArrayList<>();
public void addShape(Shape shape) {
      shapes.add(shape);
   }
public void drawAllShapes() {
      for (Shape shape : shapes) {
         shape.draw();
      }
   }
}
```

Explanation: The Shape class is open for extension (new shapes can be added) but closed for modification (existing shape classes do not need to be modified).

3. Liskov Substitution Principle (LSP)

Definition: Subtypes must be substitutable for their base types without affecting the correctness of the program.

Example:

```
class Bird {
    void fly() {
        // Flying logic
    }
}
class Sparrow extends Bird {
    void fly() {
        // Flying logic specific to sparrow
    }
}
class Ostrich extends Bird {
    void fly() {
        throw new UnsupportedOperationException("Ostrich can't fly");
    }
}
```

Explanation: Here, Ostrich violates LSP because it cannot be substituted for Bird without affecting the program's correctness. A better design would be to have a more general Bird class that does not assume all birds can fly.

4. Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on methods they do not use.

Example:

```
interface Worker {
    void work();
}
interface Eater {
    void eat();
}
class Human implements Worker, Eater {
    public void work() {
        // Working logic
    }
public void eat() {
        // Eating logic
    }
}
class Robot implements Worker {
    public void work() {
        // Working logic
    }
}
```

Explanation: By splitting the Worker and Eater interfaces, we ensure that Robot does not depend on an eat method it does not use.

5. Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions.

Example:

```
interface Database {
   void save(String data);
}
class MySQLDatabase implements Database {
   public void save(String data) {
      // Save data to MySQL database
   }
}
class MongoDBDatabase implements Database {
   public void save(String data) {
      // Save data to MongoDB database
   }
}
class DataManager {
   private Database database;
public DataManager(Database database) {
      this.database = database;
   }
public void saveData(String data) {
      database.save(data);
   }
}
```

Explanation: The DataManager class depends on the Database interface, not a specific database implementation. This allows easy switching of database implementations without modifying DataManager.