Answer no . 1

1st Code Smell:

a) The 1st code smell I have identified is in the calculator.java file and the code smell is **long class.**
b) It occurs because the class calculator is handling too many operations all together, so it becomes heavier. Which is a code smell
c) I have done the extract class here. So the new class is BiOperations. This class will handle the Bi operations of add, multiply, minus, divide, power etc things in a new class now.

2nd Code Smell:

a) The 2nd code smell is **divergent change** which also in the calculator.java class. It means when we need to change many things in a single class.
b) The calculator.java class has so many operations that changing one thing will result in change many things within the same class.
c) So I have extracted a subclass as MonoOperations which extends the super class Calculator.

3rd Code Smell:

a) The 3rd code smell is long parameter list which is BioOperatorsmode ,
b) which occurs because the method in calculator.java class is really filled with too many parameters.
c) I have created a class as BiOperatorModes and those variables are now there, we can call them by the class.

Answer No. 2:

a) In the calculator.java class, there is a huge if else conditions for the bi operations. This is a bad design practice because introducing something new will bring another if else statement.

b) So I have implemented state pattern in this class to handle the bi operations of add, minus, multiply, divide, power. First of all there is an interface as Operations. This interface contains the method calculatebiimpl(). This method is the key method for the other subclasses.

c) Now I have created 5 subclasses **as addOperation, minusOperation, multiplyOperation, divideOperation and powerOperation**. Each of the class implements the Operations interface and implement the preferable operation. Now I have declared the class s**tateHandler** which is the context class.

Now each of the operations of if else will be held in separated class and the operation will be choosen according to the state of the **statehandler**. This is how I have implemented the **State Pattern** in the calculator. Java class to remove the if else conditions of the bioperations of plus, minus, multiply, power, divide etc.

In my code, I have commented the things I have done in the code file.

The code files are :

calculator.java file:

1. Interface : Operations

2. subclass implementing the interface (Concrete States) : addOperation, minusOperation, multiplyOperation, divideOperation, powerOperation

3. Concrete class StateHandler();

Answer No. 3

New feature can be implemented by applying the **Decorator design pattern**. This pattern ensures we can implement new feature without touching or changing the existing feature.

For instance, if we want to implement the matrix calculation in out calculator, we can add a decorator class. This class wraps the existing features. And then we can implement our desired features upon that decorator class. It ensures that the Open Closed Principal is also maintained.

In the java class calculator.java, I have added an interface component as cacl with a method void addFetaure. And a concrete component class feature which implemts the interface. Now a decorator class decorator which implements calc.