SED

TDD Process: first create the tests for all the behaviors we want, add the minimum amount of code required to make us pass the tests
Coupling: Two parts of code are coupled when they must change together. High coupling causes classes to be immobile
Afferent coupling: A class's afferent coupling is a measure of how many other classes use this class, measures **responsibility**
Efferent coupling: A class's efferent coupling is a measure of how many different classes are used by the specific class, a measure of **independence**
Mock objects: a false instance of a class we own. The main purpose of this is to test how a system interacts with its dependencies without having to rely on the actual implementations (we mock interfaces)

```
import org.jmock.Expectations; import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.junit.Test;
public class TestHeadChef {
    @Rule public JUnitRuleMockery context = new JUnitRuleMockery();
    Order ROAST_CHICKEN = new Order(“roast chicken”);
    Order APPLE_TART = new Order(“apple tart”);
    Chef pastryChef = context.mock(Chef.class);
    HeadChef headChef = new HeadChef(pastryChef);
    @Test public void delegatesPuddingsToPastryChef() {
        context.checking(new Expectations() {{
            exactly(1).of(pastryChef).order(APPLE_TART);
        }});
    headChef.order(ROAST_CHICKEN, APPLE_TART);
    }
}
```

take notice to the Expectations and the @Rule
tell dont ask: only pass the data but dont expect a certain output
Template Pattern: defines the steps of the algorithm as separate methods, allowing subclasses to provide their own implementations for some or all the steps, the advantage is it allows you to define the overall structure of an algorithm while customising steps as needed

```
abstract class foo {
    public void templateMethod(){
        step1();
        step2();
    }
    protected void step1();
    protected void step2();
}
class bar extends foo {
    @Override
    protected void step2() {
        ...
    }
}
```

Strategy Pattern:we can create many instance of different Strategies, then we pass the entire strategy into the whole program

```
abstract class strategy {
    abstract void op()
}
class startegy1 extends strategy {
    void op() {
        ...
    }
}
class startegy2 extends strategy {
    void op() {
        ...
    }
}
class Prog {
    public something(Strategy stra) {
        ...
    }
}
```

Law of Demeter: Bad design = rigid, immobile, fragile. An object should only communicate with its immediate neighbours and have limited knowledge about tother units
Factory object/methods/Pattern: can return a new object as init

```
public interface A {
    void op();
}
public class B implements A {
    void op(){return 1;}
}
public class C implements A {
    void op(){return 1;}
}
public factory {
    public A factoryMethod(Stirng name){
        if (name.equals("B")) return new B()
        else if (name.equals("C")) return new C()
        else throw new IllegalArgumentException()
    }
}
```

Builder Pattern: defines the object so we can gather all the information by consecutive method calling the build the object in the end, this makes the construction process clearer

```
public ObjBuilder {
    private double field1 = 0.0;
    private double field2 = 0.0;
    private ObjBuilder() {} // empty constructor, should never be called
    public static aObjBuilder() {return new ObjBuilder()}
    public Obj build() {Obj obj = new Obj(field1,field2), return obj}
    public Obj withField1(double field1){this.field1 = field1;return this}
    public Obj withField2(double field1){this.field2 = field2;return this}
}
```

Singleton Pattern: this ensures there is only one instance of the class, ensuring a global access point, must be synchronised in order to avoid race conditions, and this can cause tight coupling between different classes, for example, if a singleton use class A then there is a tight coupling between A and the singleton, so the singleton is hard to change to another implementation

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton{} // private constructor, never be called outside
    public static Singleton getInstance(){return instance;}
}
```

ABC Metrics: Assignment: an explicit transfer of data to a variable, Branches: an explicit forward program branch out of scope, Condition: a logical/Boolean test
Interactive Applications: example:

```
import javax.swing.*;
public class App{
    public static void main(String[] args) {
        new App.display();
    }
    public void display() {
        JFrame frame = new JFrame("Example");
        frame.setSize(400,300)
        JPanel panel = new JPanel();
        JTextField tf = new JTextField(10);
        JButton b = new JButton("Press");
        panel.add(tf);
        panel.add(b);
        frame.add(panel)
        frame.setVisible(true);
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE)
    }
}
```

Observer Pattern: the subject maintains a list of observers, and when a change happens, it notifies the observers to act accordingly(consider nextjs useEffect and the dependency list), in Java swing, we would expect

```
b.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent actionEvent){...}
})
```

for example, if attached to JButton, then it will trigger when the button is pressed
MVC(Model-View-Controller) Pattern: Model = application and business logic, View = presentation of the data, Controller = input handler, the view contains a reference to the controller, the controller detects input and pass it to the model, the model computes everything and stores data

PAC(Presentation-Abstraction-Control) Pattern A hierarchy of MVC where in each module, P -> V, A -> M, C -> C (consider the components of Vue or React, each component maintains its MVC)
Event Bus: a separate object to allow distant components to communicate with each other, but we lose the hierarchical nature the tree structure gives us
Adapter Pattern: allow the use of incompatible classes by provide interfaces compatible with the client

```
// Target
interface interface Bird { void quack(); } // Existing class
class Duck implements Bird { @Override public void quack() { System.out.println("Quack!"); } }
// Adaptee
class Sparrow { public void chirp() { System.out.println("Chirp!"); } }
// Adapter
class SparrowAdapter implements Bird { private Sparrow sparrow; public SparrowAdapter(Sparrow sparrow) { this.sparrow = sparrow; } @Override public void quack() { sparrow.chirp(); } }
```

it just calls chirp in quack, making the classes compatible
Decorator Pattern: allows to add new behavior to an object

```
// Component
interface interface Coffee { String getDescription(); double cost(); }
// Concrete component
class SimpleCoffee implements Coffee { @Override public String getDescription() { return "Simple Coffee"; } @Override public double cost() { return 2.0; } }
// Base decorator
abstract class CoffeeDecorator implements Coffee { protected Coffee decoratedCoffee; public CoffeeDecorator(Coffee coffee) { this.decoratedCoffee = coffee; } @Override public String getDescription() { return decoratedCoffee.getDescription(); }
@Override public double cost() { return decoratedCoffee.cost(); } }
```

the CoffeeDecorator wraps around the Coffee object so that we can change the behaviour of getDescription and cost
The Facade pattern The facade pattern is prociding a simple interface or decouple a client from a subsystem, for example, you may want to encapsulate all the stuff in a HomeTheater class while having a subsystem of devices, in this way, the user do not need to interact with the device directly, their interact with the simple interface of the theater
Simplicator Pattern this is similar but it only simplifies the output, for example, we have a complex calculator that does something complex, so we wrap around the class and in a method we design a simpleOperation that just call the complexOperation, so the user only have to deal with the simpleOperation
Proxy Pattern: when we want to call a method of a certain class, we call the method of a wrapper class that then calls the method of the target class, so that we may do operations in the intermediate class without altering the user class or the target class
Caching: in the proxy pattern, you can store items in the intermediate class to avoid calling the same function over and over again
Hexagonal Architecture(ports and adapters): having a core (think of an ip address), and several adapters in to interact with third party services, so when we need to switch a 3rd party service, we dont change the core part
Unit test: test the individual elements of the core, Any external services can be mocked at the adapter level
Integration tests: To test the adapters, for example, MySQL, we write many database operations and test if everything works fine
Legacy system: first discover the dependencies, only refactor the things we have to and understand, keep the stuff that works and dont change too much
Seam a place where you can alter behavior of the program without editing in that place, and has a enabling point, for example, A class call a Emailsender to send email, we can change the behaviour of the place where email is send in the initialisation where we create the emailsender, maybe another class of the same interface
HTTP and REST(Representational State Transfer): Resources are defined by URIs, and we can use xml or json to represent them, use the HTTP requests to request data, still stateless
RMM (Richardson Maturity Model): Level 0(protocols): use http protocol but do not take advantage of the URIs to identify resources **SOAP(simple object access protocol)** wraps a xml document of a request; Level 1(URIs): more URIs, but typically do not take advantage of all the available HTTP methods, sometimes do not support; Level2: Level1 but uses HTTP methods to update the state of resource; Level3: Level2 but can include hyperlinks to other resources
Waterfall development model: sets out a number of phases, if one phase is done, we are complete done with it and never rework, too rigid or can be wrong
Agile Development model: works in short cycles (sprints) we design, build, release small sets of features. discover value from the releases, shortens the time between idea and implementation, the first release happens sooner, means the produce can be sold earlier
CI/CD(Continuous Integration/Development)
future: a placeholder for the result of a task that has not yet been completed (TODOs, python pass or NotImplementedError)
Latch: a concurrent utility that allow one of more threads to wait (similar to semaphore)
Commander pattern:

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
public class Execution {
    public static void main(String[]
args){
    Executor executor = Executors.
    newFixedThreadPool(2);
    executor.execute(new C("A",10));
    executor.execute(new C("B",10));
    executor.execute(new C("C",10));
    System.out.println(“Done.");
}
```

queue up everything with encapsulation and executes in the end
dependency inversion: High-level modules should not depend on low-level modules. Both should depend on abstractions. so to apply this, for example A use C inside, then we use B to wrap C and pass B to A upon construction

```
// Low-level module
class LightBulb {
    public void turnOn() { System.out.println("Light on"); }
    public void turnOff() { System.out.println("Light off"); }
}

// High-level module (directly depends on LightBulb)
class Switch {
    private LightBulb bulb = new LightBulb(); // Tight coupling ❌

    public void press() {
        bulb.turnOn();
    }
}
```

with dependency inversion:

```
// Abstraction (interface)
interface Device {
    void turnOn();
    void turnOff();
}

// Low-level module (implements the abstraction)
class LightBulb implements Device {
    public void turnOn() { System.out.println("Light on"); }
    public void turnOff() { System.out.println("Light off"); }
}

// Another low-level module (e.g., a fan)
class Fan implements Device {
    public void turnOn() { System.out.println("Fan on"); }
    public void turnOff() { System.out.println("Fan off"); }
}

// High-level module (depends on the abstraction)
class Switch {
    private Device device; // Depends on interface ✅

    public Switch(Device device) {
        this.device = device; // Dependency injected
    }

    public void press() {
        device.turnOn();
    }
}
```