Domanda 1:

Il pattern Decorator è un pattern definito originariamente dalla Gang of Four, che consente di estendere il comportamento di un oggetto senza doverlo modificare direttamente. Questo pattern consente di avvolgere un oggetto all'interno di un altro oggetto, aggiungendo funzionalità extra dinamicamente.

Per comprendere meglio come funziona il pattern Decorator, immagina di avere una classe (non final)/classe astratta/interfaccia di base chiamata "Component" che rappresenta un oggetto di base con un certo comportamento. Successivamente, vuoi estendere il comportamento di questo oggetto di base aggiungendo funzionalità extra senza modificare direttamente la classe "Component".

Il pattern Decorator fa uso dell'ereditarietà e della composizione per ottenere questo risultato. Innanzitutto, crei una classe astratta chiamata "Decorator" che eredita dalla classe "Component" e ha un riferimento (instance field) a un oggetto "Component". Successivamente, crei delle classi concrete che ereditano dalla classe "Decorator" e implementano le funzionalità extra desiderate.

Il decorator si presenta come un'alternativa all'ereditarietà con due principali vantaggi:

- componibilità: poiché il decoratore stesso implementa la classe base (nell'esempio sopra Component), è possibile passare ad un decoratore un altro decoratore. Puoi continuare a creare più livelli di decoratori per estendere ulteriormente il comportamento. Ogni decoratore aggiuntivo avvolge l'oggetto precedente, creando così una catena di decoratori che possono essere impilati in modo flessibile.
- logiche a runtime: è possibile definire a runtime quanti decoratori applicare all'oggetto ed in che ordine, di fatto permettendo qualsiasi combinazione invece che definirla staticamente a compile time.

Di seguito un esempio di una implementazione del pattern Decorator

Data una classe astratta A e una sua implementazione A_I, un esempio di applicazione del pattern decorator è il seguente: creeremo una classe D (la nostra classe base per i decoratori) che eredita da A e una implementazione che chiamiamo D_I che prenderà in costruzione un oggetto di tipo A.
Il decoratore D_I eseguirà l'override dei metodi di A aggiungendo comportamento custom. Invocando il metodo M di D_I, il decoratore eseguirà il suo comportamento custom oltre ad invocare internamente lo stesso metodo M dell'oggetto di tipo A che ha preso in costruzione.

```
public abstract class A{

public abstract int somma(int a, int b);

}
```

```
public class A_I extends A {

@Override
public int somma(int a, int b)
        return a+b;
}

}



public abstract class D extends A {}

public class LoggerDecorator extends D {

private A original;

public  LoggerDecorator(A original){
        this.original = original;
}

@Override
public int somma(int a, int b)
        System.out.println(“a: %d b:%d”.formatted(a, b));
        return original.somma(a,b);
}

}
```

Nella JDK un esempio di decorator pattern utilizzato nel package java.io è il BufferedInputStream. La classe BufferedInputStream prende in input un oggetto di tipo InputStream, classe che lui stesso implementa. Questo è un tipico segnale di utilizzo di decorator pattern.

```
import java.io.*;

public class EsempioDecorator {
   public static void main(String[] args) {
      try {
         InputStream inputStream = new FileInputStream("file.txt");
         InputStream bufferedInputStream = new BufferedInputStream(inputStream);

         int data = bufferedInputStream.read();
         while (data != -1) {
            System.out.print((char) data);
            data = bufferedInputStream.read();
```

```
            }

            bufferedInputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Domanda 2:

Il pattern “Template Method” permette di definire un algoritmo con una logica predefinita, di cui è possibile personalizzare alcuni aspetti.
In particolare, data una classe con un “template method”, esso internamente farà riferimento ad altri metodi presenti in classe. Questi metodi possono essere di 3 tipi:

- metodi astratti (le sottoclassi si occuperanno di implementarli, personalizzando quindi l’esecuzione dell’algoritmo principale)
- metodi non final: questi metodi contengono le istruzioni predefinite da eseguire. La sottoclasse può decidere se sovrascrivere il metodo per cambiarlo o lasciarlo inalterato
- metodi final: questi metodi contengono le istruzioni da eseguire obbligatoriamente all’interno dell’algoritmo contenuto nel template method