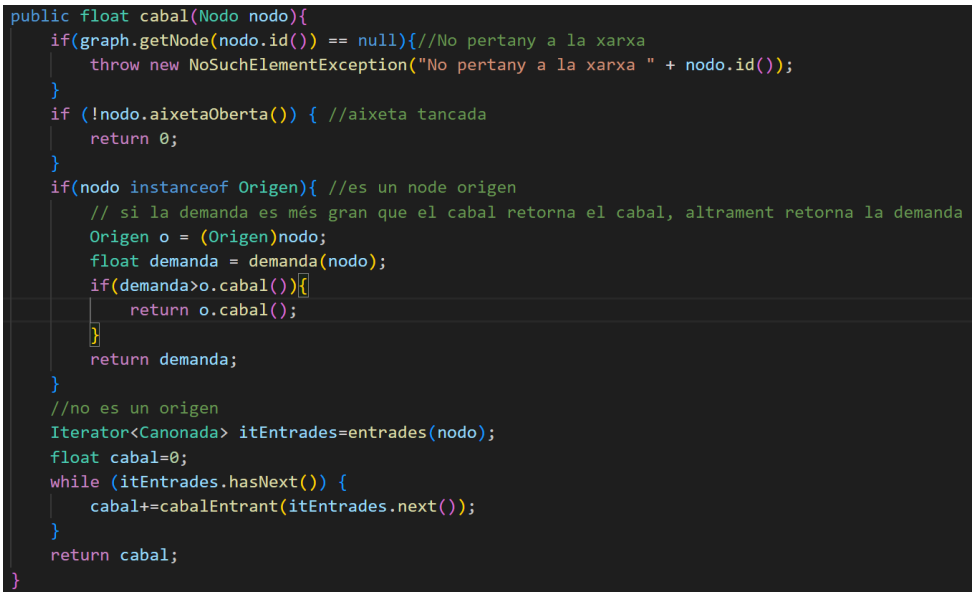
<u>Concretament, dels apunts del bloc Principis SOLID:</u>

Detecteu 5 elements del vostre codi que sospitosos de “mal disseny”. Heu de posar-hi el codi, o una part, i explicar perquè.

1)

Hi ha una part del codi la qual podríem categoritzar de repetició innecessària o immobilitat, ja que es troba en gran part a la classe Xarxa i a la classe GestorXarxes.

Codi Xarxa.java:



```java
public float cabal(Nodo nodo){
    if(graph.getNode(nodo.id()) == null){//No pertany a la xarxa
        throw new NoSuchElementException("No pertany a la xarxa " + nodo.id());
    }
    if (!nodo.aixetaOberta()) { //aixeta tancada
        return 0;
    }
    if(nodo instanceof Origen){ //es un node origen
        // si la demanda es més gran que el cabal retorna el cabal, altrament retorna la demanda
        Origen o = (Origen)nodo;
        float demanda = demanda(nodo);
        if(demanda>o.cabal()){
            return o.cabal();
        }
        return demanda;
    }
    //no es un origen
    Iterator<Canonada> itEntrades=entrades(nodo);
    float cabal=0;
    while (itEntrades.hasNext()) {
        cabal+=cabalEntrant(itEntrades.next());
    }
    return cabal;
}
```

Codi GestorXarxes.java

```java
public static float cabal(Nodo nodo, Graph graph, Xarxa x){
    if(graph.getNode(nodo.id()) == null){//No pertany a la xarxa
        throw new NoSuchElementException("No pertany a la xarxa " + nodo.id());
    }
    if (!nodo.aixetaOberta()) { //aixeta tancada
        return 0;
    }
    if(nodo instanceof Origen){ //es un node origen
        Origen o = ((Origen)nodo);
        float demanda = demanda(nodo, graph,x);
        if(demanda>o.cabal()){
            return o.cabal();
        }
        else{
            return demanda;
        }
    }
    Node graphNode=graph.getNode(nodo.id());
    Stream<Edge> entrades=graphNode.enteringEdges();
    Iterable<Edge> iteradorEntrades = entrades::iterator;
    float cabal=0;
    for (Edge edge : iteradorEntrades) {
        cabal+=cabalEntrant(edge, graph, x);
    }
    return cabal;
}
```

Com es pot veure és un codi molt semblant que podríem mirar d'evitar la repetició d'alguns trossos. El motiu pel qual ho hem fet així és perquè el mètode d'excesCabal del GestorXarxes no li serveix utilitzar els mètodes demanda i cabal de la classe Xarxa. Ja que en els mètodes de xarxa es té en compte la capacitat de les canonades a l'hora de propagar la demanda, en canvi, en el mètode excesCabal de GestorXarxes no s'ha de tenir en compte.

2)

Es podria considerar que treure els espais del fitxer d'entrada és una complicació innecessària. Ho hem fet així per no perdre molt temps buscant errors en el codi, quan eren errors de l'entrada molt difícil de veure, com en aquest cas que eren espais en blanc.

```java
private String llegir(BufferedReader br){
    String linea="";
    try{
        linea=br.readLine();
    }
    catch(IOException e) {
        e.printStackTrace();
    }
    if(linea==null){
        return null;
    }
    return linea.replaceAll(regex:"\\s", replacement:"");//Eliminem tots els espais per evitar errors d'entrada
}
```

3)

Una complicació innecessària és la funció omplirEntrades(), que emparella cada node amb els nodes que li entren i s'utilitza per comprovar si una xarxa té cicles.

```
private static void omplirEntrades(Map<String,ArrayList<String>> entrades,Xarxa x, Origen nodeOrigen) {
    Graph subGfraf =x.componentConexa(nodeOrigen);
    for (Node node : subGfraf){
        Nodo n = new Nodo(x.node(node.getId()));
        Iterator<Canonada> itE= x.entrades(n);
        String nom= node.getId();
        entrades.put(nom,new ArrayList<>());
        while(itE.hasNext()) {
            String id=itE.next().node1().id();
            entrades.get(nom).add(id);
        }
    }
}
```

Podríem considerar-la innecessària, ja que GraphStream ja proporciona una funció que, donat un node, ens permet saber quins nodes li entren i iterar directament sobre ells dins de la funció iteCicles.

```
private static boolean iteCicles(Map<String,ArrayList<String>> entrades, List<String> visitats, String seg){
    boolean trobat =false;
    if(visitats.contains(seg)){
        trobat=true;

    }
    else{
        visitats.add(seg);
        if(entrades.containsKey(seg)){
            List<String> conexs= new ArrayList<>(entrades.get(seg));
            if(entrades.get(seg)!=null){
                int i=0;
                while(i< conexs.size() && !trobat){
                    trobat=iteCicles(entrades, visitats, conexs.get(i));
                    visitats.remove(conexs.get(i));
                    i++;
                }
            }
        }
    }


    return trobat;
}
```

4)
Una altra complicació innecessària pot sorgir a l'hora de construir el graf residual de la funció fluxMaxim(). Connectem el node super origen amb les sortides dels orígens tenint en compte la seva capacitat, i si dos nodes origen van a una mateixa connexió, hem d'augmentar la capacitat de la canonada que va del super origen a aquella connexió. Podríem connectar el node super origen directament amb els orígens amb canonades de capacitat infinita, això faria que el codi sigués molt més llegible i no necessitaríem les funcions posarArestes, posarSortidesMaxFlow i posarEntradesMaxFlow.

5)
Hi ha una rigidesa en el codi relacionada amb el disseny a l'hora de dibuixar el Graph amb GraphStream. Si volem canviar els noms de les constants que fem servir per dibuixar el Graph, les quals es troben a la classe Xarxa, també ens veurem obligats a modificar els fitxers de disseny (.css).

Constants en el fitxer Xarxa.java:

```
private static final String atributGraficOrigen = "gorigen"; ///< Atribut que utilitzem en el gr
private static final String atributGraficTerminal = "gterminal"; ///< Atribut que utilitzem en e
private static final String atributGraficConnexio = "gconnexio"; ///< Atribut que utilitzem en e
private static final String atributGraficAixetaTancada = "tancada"; ///< Atribut que assignarem
```

Exemple de disseny (estil1.css):

```
node.tancada {
    size: 15px;
    fill-image: url('.\recurs
    fill-mode: image-scaled;

}

node.gorigen {
    size: 25px;
    fill-color: white;
}

node.gterminal {
    fill-color: white;
}

node.gconnexio {
    fill-color: black;
}
```

Expliqueu 3 elements dels Principis SOLID que heu aplicat o que, amb el que ara heu après, aplicaríeu i com..

<u>El principi de substitució de Liskov:</u>

En el nostre programa tenim que Origen, Terminal i connexió són un subtipus de Nodo.

En aquesta classe podem trobar aquests mètodes:

```java
public class Nodo {
    public Nodo(String id, Coordenades c){
    }

    /**
     * @brief Retorna l'identificador
     * @pre cert
     * @post Retorna l'identificador del node
     */
    public String id(){
        return id;
    }

    /**
     * @brief Retorna les coordenades
     * @pre cert
     * @post Retorna les coordenades del node
     */
    public Coordenades coordenades(){
        return c;
    }

    /**
     * @brief Estat de l'aixeta
     * @pre cert
     * @post Retorna true si l'aixeta es oberta, altrament false
     */
    public boolean aixetaOberta(){
        return aixetaOberta;
    }
```

I com diu el principi de substitució de Liskov si reemplacem Nodo per qualsevol dels seus subtipus no hauria d'alterar el comportament del programa, com en aquest exemple d'una funció de la classe Xarxa on en un Origen utilitzem la funció id() com es mostra a continuació:

```java
public void afegir(Origen nodeOrigen){
    if(graph.getNode(nodeOrigen.id()) != null){//Ja existeix un node amb aquest id
        throw new IllegalArgumentException("ja existeix a les xarxes una aixeta amb nom " + nodeOrigen.id() + ".");
    }
    //No existeix un node amb aquest id
    Node n=graph.addNode(nodeOrigen.id());
    n.setAttribute(atributNode, nodeOrigen);
```

<u>El principi de "Single-Responsibility":</u>

Aquest principi esmenta que cada classe o funció ha de ser responsable de fer una sola cosa, en el nostre cas, violem aquest principi perquè tant en la classe Xarxa com en la de GestorXarxes hi ha mètodes per dibuixar:

dibuixa():

```java
public void dibuixar(Origen nodeOrigen){
    Graph subGraph = componentConexa(nodeOrigen); //creo un subGraph amb només la xarxa corresponent
    SpriteManager sman = new SpriteManager(subGraph);//utilitzat per poder mostrar més informació en el dibuix
    for (Node nodeGraph : subGraph){
        nodeGraph.setAttribute("ui.label", nodeGraph.getId()); //mostrem els id's de tots els nodes del graph
        Nodo nodo = node(nodeGraph.getId());
        String propietatsClass="";

        //a cada nodo afegeixo la informació de les coordeandes
        dibuixarCoordenades(sman, nodo);

        if(nodo instanceof Terminal){
            Terminal t=(Terminal)nodo;
            propietatsClass+=atributGraficTerminal;
            //afegeixo la informació a cada terminal, sobre la demanda actual i la demanda punta
            dibuixarTerminal(sman, t);
        }
        else if(nodo instanceof Origen){
            propietatsClass+=atributGraficOrigen;
        }
        else{
            propietatsClass+=atributGraficConnexio;
        }
        if(!nodo.aixetaOberta()){//aixeta tancada
            propietatsClass+=", " + atributGraficAixetaTancada;
        }

        //posicionar
        nodeGraph.setAttribute("x", nodo.coordenades().getLongitud());
        nodeGraph.setAttribute("y", nodo.coordenades().getLatitud());

        nodeGraph.setAttribute("ui.class", propietatsClass);

        for(Edge edge : nodeGraph){
            Canonada c = canonada(edge.getId());
            float cabal=cabalEntrant(c);
            edge.setAttribute("ui.label", Float.toString(cabal)+ " / " + Float.toString(c.capacitat()));
        }
    }
    subGraph.setAttribute("ui.stylesheet", "url('recursos\\estil1.css')");//associar a un stylesheet
    subGraph.display(false);//mostrar segons les coordenades
    //subGraph.display();
}
```

fluxMaxim():

```java
public static void fluxMaxim(Xarxa x, Origen nodeOrigen){

    Graph graphMaxFlow=crearGraphMaxFlow(x, nodeOrigen);

    Node s=graphMaxFlow.getNode(SUPERORIGEN);
    Node t=graphMaxFlow.getNode(SUPERTERMINAL);
    List<String> nodesVisitats=new ArrayList<>();;
    while(hiHaCami(s, t, nodesVisitats)){
        Float flow = minimCapacitat(graphMaxFlow, nodesVisitats);
        posarFlux(graphMaxFlow, nodesVisitats, flow);
        nodesVisitats.clear();
    }

    s.setAttribute("ui.label", "S");
    t.setAttribute("ui.label", "T");
    Stream<Edge> arestes=graphMaxFlow.edges();
    Iterator<Edge> it=arestes.iterator();
    while(it.hasNext()){
        Edge edge=it.next();
        if(edge.hasAttribute(atributCapacitat)){
            Node source=edge.getSourceNode();
            Node target=edge.getTargetNode();
            Edge edge2=target.getEdgeToward(source);
            Float flux=(Float)edge2.getAttribute(atributFlux);
            Float capacitat=(Float)edge.getAttribute(atributCapacitat);
            edge.setAttribute("ui.label", Float.toString(flux) + "/" + Float.toString(capacitat+flux));
        }
        else{
            edge.setAttribute("ui.hide");
        }
    }
    graphMaxFlow.setAttribute("ui.stylesheet", "url('recursos\\estil2.css')");
    graphMaxFlow.display();
}
```

Si volguéssim implementar correctament l'SRP hi hauria d'haver una classe impressora que es dediqués a fer això.

El principi "Open/Closed"

El principi "Open/Closed" estableix que un mòdul hauria d'estar obert per a l'extensió però tancat per a la modificació. Això significa que hauríem de poder afegir noves funcionalitats o comportaments a un mòdul sense necessitat de canviar el seu codi font original.

En el context del nostre programa, podríem trobar aquest problema amb la classe GestorXarxes. Si en el futur volem utilitzar el gestor per a una xarxa no dirigida i la classe no implementa cap interfície, hauríem de modificar gairebé tot el codi d'aquesta classe. Això violaria completament el principi "Open/Closed".

Una solució per complir amb aquest principi seria refactoritzar GestorXarxes perquè implementi una interfície, per exemple, XarxaInterfície, que defineixi mètodes

genèrics per gestionar una xarxa. D'aquesta manera, podríem crear una nova implementació de XarxaInterfície per a una xarxa no dirigida sense modificar la classe original GestorXarxes, complint així amb el principi "Open/Closed".

Dels apunts del bloc Getters & Setters:

Preneu 3 exemples de getters/setters del vostre codi (si n'hi ha) i expliqueu per què seria convenient modificar-los (o no), i com ho faríeu.

1)

```
/**
 * @brief Retorna la latitud
 * @pre cert
 * @post Retorna el valor de la latitud
 */
public double getLatitud() {
    return a_latitud;
}

/**
 * @brief Retorna la longitud
 * @pre cert
 * @post Retorna el valor de la longitud
 */
public double getLongitud() {
    return a_longitud;
}
```

Aquí es pot veure com estic tractant Coordenades com un simple contenidor de dades. En aquest cas encara necessito obtenir aquesta informació de les coordenades, però és conceptualment incorrecte tenir mètodes que comencin amb get o set. Per tant, el que faria seria treure la cadena get dels mètodes.

2)

```java
/**
 * @brief Retorna l'atribut graph
 * @pre cert
 * @post Retorna l'atribut graph
 */
public Graph grafic(){
    return graph;
}
```

Aquest codi està a la classe Xarxa, i és el GestorXarxes qui l'utilitza en una ocasió. Sincerament amb els 3 arguments clàssics contra els getters & setters de les diapositives es veu clarament perquè no ho hauríem de fer així. Qualsevol consulta sobra la Xarxa que necessiti el GestorXarxes s'hauria de realitzar a través de mètodes de Xarxa. Clarament s'hauria de modificar. S'hauria de mirar si el GestorXarxes necessita més mètodes per consultar el graph, si ha d'afegir-los a Xarxa, i llavors canvia la part del codi de GestorXarxes per tal de no necessitar aquest get.

3)

```java
//calculem cabal mínim
for(Terminal t:terminals){
    float aux=t.demandaActual();
    if(aux==0){
        aux=t.demandaPunta();//si no té cap demanda actual interpretem la seva demanda punta
    }
    cabal+=aux*(percentatgeDemandaSatisfet/100);
}
return cabal;
```

De la mateixa manera que en el punt 2), en la funció de cabalMinim() de GestorXarses es demana la demandaActual d'un terminal per fer una comparació de si és igual a 0, per fer una correcta implementació caldria preguntar a terminal si la seva demandaActual és igual a 0. Per exemple, en el if podríem posar: (t.teDemandaActual()).