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

1. A GestorXarxes hi ha una acció anomenada omplirMapDemanda que el seu problema és que té masses variables d’entrada. Aquest fet es dona ja que necessito totes aquelles dades per a executar l’acció.

```
private static void omplirMapDemanda(Xarxa x, Node node, Map<Node,
Boolean> visitats, Map<Canonada, Float> demandaCanonades, Map<Node,
Float> demandaNodes, Map<Node, Float> cabalNodes, Map<Canonada, Float>
cabalCanonades){
    Iterator<Canonada> itCanonades = x.sortides(node);

    while (itCanonades.hasNext()) {
        Canonada canonada = itCanonades.next();

        if (visitats.containsKey(canonada.node2())) {
            float demandaNode = demandaNodes.get(node);
            float cabalNode = cabalNodes.get(node);
            float demandaPerCabal = demandaNode / cabalNode;
            float cabalCanonada = demandaCanonades.get(canonada) /
demandaPerCabal;
            cabalCanonades.put(canonada, cabalCanonada);
        }
    }
}
```

2. A l’acció de GestorXarxes calcularCabal es pot veure que hi ha dos fors un dins l’altre i entremig hi ha un if que si no es compleix no passa dins de l’altre for i dins del segon for hi ha un altre condicional. Aquest fet provoca que alhora d’entendre el codi sigui més complicat. El faig així ja que s’han de complir totes les condicions per poder entrar a l’últim condicional i calcular el cabal del node entry.

```
private static void calcularCabal(Xarxa x, Map<Node, Boolean> visitats,
Map<Canonada, Float> cabalCanonades, Map<Node, Float> cabalNodes,
Map<Canonada, Float> demandaCanonades, Map<Node, Float> demandaNodes) {
    // Primer tractem els nodes que ja estan marcats com a visitats
    for (Map.Entry<Node, Boolean> entry : visitats.entrySet()) {
        Node node = entry.getKey();
        Boolean visitat = entry.getValue();
```

```
        if (visitat) {
            omplirMapDemanda(x, node, visitats, demandaCanonades,
demandaNodes, cabalNodes, cabalCanonades);
        }
    }

    // Ara tractem la resta de nodes
    boolean totsVisitats = false;

    while (!totsVisitats) {
        totsVisitats =
muntarCabal(x,visitats,cabalCanonades,cabalNodes,demandaCanonades,demand
aNodes);
    }
}

private static void omplirMapDemanda(Xarxa x, Node node, Map<Node,
Boolean> visitats, Map<Canonada, Float> demandaCanonades, Map<Node,
Float> demandaNodes, Map<Node, Float> cabalNodes, Map<Canonada, Float>
cabalCanonades){
    Iterator<Canonada> itCanonades = x.sortides(node);

    while (itCanonades.hasNext()) {
        Canonada canonada = itCanonades.next();

        if (visitats.containsKey(canonada.node2())) {
            float demandaNode = demandaNodes.get(node);
            float cabalNode = cabalNodes.get(node);
            float demandaPerCabal = demandaNode / cabalNode;
            float cabalCanonada = demandaCanonades.get(canonada) /
demandaPerCabal;
            cabalCanonades.put(canonada, cabalCanonada);
        }
    }
}
```

3. En actualitzarFlux, que està a la classe de GestorXarxes, es dona el cas de que maneja múltiples responsabilitats. Ha de trobar el flux mínim i actualitzar els valors de flux tant de fluxEntrades com fluxSortides. Ho faig d'aquesta manera ja que l'acció ocupa menys de 30 línies i si hagués d'agafar una de les

dues parts i posar-la en una altre acció o accions, les variables d'entrada d'aquestes accions serien moltes igual que a actualitzarFlux i he preferit fer-ho tot en una sola acció.

```
private static void actualitzarFlux(List<Node> cami, Map<Node, Map<Node,
Float>> fluxSortides, Map<Node, Map<Node, Float>> fluxEntrades) {
    // Trobar el flux mínim en el camí
    float fluxMinim = Float.MAX_VALUE;
    for (int i = 0; i < cami.size() - 1; i++) {
        Node nodeActual = cami.get(i);
        Node nodeSeguent = cami.get(i + 1);
        float flux;
        if(fluxSortides.get(nodeActual).containsKey(nodeSeguent)) flux =
fluxSortides.get(nodeActual).get(nodeSeguent);
        else flux = fluxEntrades.get(nodeActual).get(nodeSeguent);

        if (flux < fluxMinim) fluxMinim = flux;
    }

    // Actualitzar fluxSortides i fluxEntrades
    for (int i = 0; i < cami.size() - 1; i++) {
        Node nodeActual = cami.get(i);
        Node nodeSeguent = cami.get(i + 1);

        if (fluxSortides.get(nodeActual).containsKey(nodeSeguent)) {
            // El nodeSeguent és una sortida del nodeActual
            float fluxSortida =
fluxSortides.get(nodeActual).get(nodeSeguent);
            fluxSortides.get(nodeActual).put(nodeSeguent, fluxSortida -
fluxMinim);
            float fluxEntrada =
fluxEntrades.get(nodeSeguent).get(nodeActual);
            fluxEntrades.get(nodeSeguent).put(nodeActual, fluxEntrada +
fluxMinim);
        }
        else if (fluxEntrades.get(nodeActual).containsKey(nodeSeguent)) {
            // El nodeSeguent és una entrada del nodeActual
            float fluxEntrada =
fluxEntrades.get(nodeActual).get(nodeSeguent);
            fluxEntrades.get(nodeActual).put(nodeSeguent, fluxEntrada -
fluxMinim);
            float fluxSortida =
fluxSortides.get(nodeSeguent).get(nodeActual);
```

```
            fluxSortides.get(nodeSeguent).put(nodeActual, fluxSortida +
fluxMinim);
        }
    }
}
```

4. En aquest cas veurem la funció Node de Xarxa que retorna un node

```
public Node node(String id) {

    if(graph.getNode(id) != null) {
        if (graph.getNode(id).hasAttribute( s: "origen")) {
            return (Node) graph.getNode(id).getAttribute( s: "origen");
        }
        if (graph.getNode(id).hasAttribute( s: "terminal")) {
            return (Node) graph.getNode(id).getAttribute( s: "terminal");
        }
        if (graph.getNode(id).hasAttribute( s: "connexio")) {
            return (Node) graph.getNode(id).getAttribute( s: "connexio");
        }
    }
    return null;
}
```

Violació del principi de responsabilitat única (SRP) → Aquest mètode node realitza diverses tasques , verifica l'existència del node , comprova múltiples atributs i retorna el node adequat. Això podria separar-se en diversos mètodes ,es petits, cadascun amb una única realitat.

Violació del Principi de Obert/Tancat (OCP) → En el cas de que s'afegeixen un nou tipus de node el mètode aquest hauria d'estar modificat per a que funcioni, la qual cosa va en contra del principi de obert/tancat, aquest suggereix que las classes han d'estar obertes a l'extensió però tancades a la modificació.

Solució al problema

```java
public Node getNode(String id) {
    Node node = graph.getNode(id);
    if (node != null) {
        return (Node) getNodeAttribute(node);
    }
    return null;
}

private Object getNodeAttribute(Node node) {
    if (node.hasAttribute("origen")) {
        return node.getAttribute("origen");
    }
    if (node.hasAttribute("terminal")) {
        return node.getAttribute("terminal");
    }
    if (node.hasAttribute("connexio")) {
        return node.getAttribute("connexio");
    }
    return null;
}
```

5. En aquest cas podem veure també uns errors de mal disseny en connectarAmbCanonada accio de Xarxa →

```java
public void connectarAmbCanonada(Node node1, Node node2, float c) {
    // Pre: node1 i node2 pertanyen a la xarxa, no estan connectats, i node1 no és un node terminal
    // Post: S'han connectat els nodes amb una canonada de capacitat c, amb sentit de l'aigua de node1 a node2
    // Excepcions: NoSuchElementException node1 o node2 no pertanyen a la xarxa
    // IllegalArgumentException els nodes ja estan connectats o node1 és un node terminal

    if (graph.getEdge( s: node1.id() + "-" + node2.id()) != null || graph.getEdge( s: node2.id() + "-" + node1.id()) != null) {
        throw new IllegalArgumentException("Ja existeix una connexio entre aquests nodes");
    }
    if (graph.getNode(node1.id()) == null || graph.getNode(node2.id()) == null) {
        throw new NoSuchElementException("Un o els dos nodes no pertanyen a la xarxa");
    }
    if (node1 instanceof Terminal) {
        throw new IllegalArgumentException("El node1 es tracta de un node terminal");
    }

    Canonada canonada = new Canonada(node1, node2, c);

    if (node2 instanceof Origen) {
        graph.getNode(node2.id()).setAttribute( s: "connexio", new Connexio(node2.id(), node2.coordenades()));
        actualizarConexiones(node2, (Connexio) graph.getNode(node2.id()).getAttribute( s: "connexio"));
        graph.getNode(node2.id()).removeAttribute( s: "origen");
        graph.getNode(node2.id()).setAttribute( s: "ui.label", node2.id());
        graph.getNode(node2.id()).setAttribute( s: "ui.style",  ...objects: "fill-color: red; size: 25px; text-size: 15;");
        actualizarEtiqueta((Connexio) graph.getNode(node2.id()).getAttribute( s: "connexio"));
        canonada = new Canonada(node1, (Connexio) graph.getNode(node2.id()).getAttribute( s: "connexio"), c);
    }
        graph.addEdge( id: node1.id() + "-" + node2.id(), node1.id(), node2.id(),  directed: true).setAttribute( s: "Canonada", canonada);
        graph.getEdge( s: node1.id() + "-" + node2.id()).setAttribute( s: "ui.style",  ...objects: "fill-color: blue; size: 3px;");
}
```

Violació del Principi de Responsabilitat Única (SRP) → El mètode realitza diverses tasques: verifica l'existència de nodes i connexions, llança excepcions si és necessari, crea una nova connexió i actualitza atributs i estils. Hauria de descompondre's en diversos mètodes més petits.

Violació del Principi de Inversió de Dependències (DIP) → El mètode depèn directament de la implementació concreta de la classe Canonada i la manipulació dels seus atributs. Hauria de dependre d'abstraccions en lloc de concrecions.

Proposta de solució :

```
public void connectarAmbCanonada(Node node1, Node node2, float c) {
    validarNodes(node1, node2);
    Canonada canonada = crearCanonada(node1, node2, c);
    actualizarGrafo(node1, node2, canonada);
}

private void validarNodes(Node node1, Node node2) {
    if (graph.getEdge(node1.id() + "-" + node2.id()) != null || graph.getEdge(node2.id() + "-" + node1.id()) != null) {
        throw new IllegalArgumentException("Ja existeix una connexio entre aquests nodes");
    }
    if (graph.getNode(node1.id()) == null || graph.getNode(node2.id()) == null) {
        throw new NoSuchElementException("Un o els dos nodes no pertanyen a la xarxa");
    }
    if (node1 instanceof Terminal) {
        throw new IllegalArgumentException("El node1 es tracta de un node terminal");
    }
}

private Canonada crearCanonada(Node node1, Node node2, float c) {
    Canonada canonada = new Canonada(node1, node2, c);
    if (node2 instanceof Origen) {
        graph.getNode(node2.id()).setAttribute("connexio", new Connexio(node2.id(), node2.coordenades()));
        actualizarConexiones(node2, (Connexio) graph.getNode(node2.id()).getAttribute("connexio"));
        graph.getNode(node2.id()).removeAttribute("origen");
        canonada = new Canonada(node1, (Connexio) graph.getNode(node2.id()).getAttribute("connexio"), c);
    }
    return canonada;
}
```

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

Després d'examinar tot el codi del nostre programa, tres dels principis SOLID que apliquem son SRP, OCP i LSP.

1. Single-Responsibility Principle (SRP):

   En les classes de Coordenades, Node, Origen i Terminal es pot observar que aquestes només tenen una única responsabilitat. Coordenades gestiona només les coordenades d'un punt en un espai, Node gestiona tots els nodes de la xarxa de distribució d'aigua, la subclasse de Node anomenada Origen gestiona els nodes, però només els d'origen en la xarxa i Terminal, que també és una subclasse de Node, gestiona els nodes terminals.

2. Open/Closed Principle (OCP):

   Les classes han d'estar obertes per a l'extensió però tancades per a la modificació. Això significa que hauríem de poder afegir nova funcionalitat al codi existent sense modificar el codi ja existent.
   En el nostre programa, el principi SOLID està aplicat en la classe Node i les subclasses Origen i Terminal. La classe Node defineix el comportament dels nodes mentres que Terminal i Origen, que son les subclasses de Node juntament amb Connexió, li afegeixen funcionalitats als nodes d'origen i terminals sense modificar la classe Node.

3. Liskov Substitution Principle (LSP):

   Les subclasses Origen i Terminal poden substituir la classe Node en el codi client sense trencar el comportament esperat, seguint el principi LSP. El codi client seria la classe SimuladorModeText ja que utilitza els mètodes d'Origen, Terminal i Node per a modificar l'estat de les aixetes i processar els nodes per mostrar la seva informació.

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. Hi ha un getter a la funció camiMesCurt de la classe de GestorXarxes que a la penúltima línia hi ha un condicional de que si en la primera posició de la List<Node> camí no està l'Origen de la xarxa que retorni un null que significa que el camí no és vàlid. Aquest get, concretament getFirst(), ha d'estar segur ja que si no fes aquest get hauria de fer un for per mirar totes les posicions quan només necessito la primera.