Projecte de Programació

Curs 2023/24

GEINF

Programa: Be Water

Teoría

Universitat de Girona
Departament d'Informàtica, Matemàtica Aplicada i Estadística

Aleix Suriñach
Arnau Herrera

Índex

Elements sospitosos de “mal disseny”

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

- ## Element 1

Hi ha codi repetit en els mètodes *sortides(NodeAixeta node)* i *entrades(NodeAixeta node)*. L’única cosa que canvia entre els dos mètodes, és el mètode del node que es crida, la resta de la funció és idèntica a *entrades* fent que es repeteixi línies innecessàries.

```
public Iterator<Canonada> sortides(NodeAixeta node)
{
   Node nodeIterador = _xarxa.getNode(node.id());
   Iterator<Edge> edgeIterator = nodeIterador.leavingEdges().iterator();

   // Creem una llista per emmagatzemar les canonades
   List<Canonada> canonades = new ArrayList<>();

   // Convertim cada Edge a Canonada i l'afegim a la llista
   while (edgeIterator.hasNext()) {
       Edge aresta = edgeIterator.next();
       Canonada pipe = (Canonada) aresta.getAttribute("canonada");
       canonades.add(pipe);
   }

   return canonades.iterator();
}
```

- ## Element 2

En la funció *teCicles*, es requereix un subgraf el qual pertany el node, per poder mirar si el graf pertinent té cicles. El mètode *crearSubgraf* és un mètode recursiu que recórrer el graf que a la vegada conté un while. Si sumem aquest recorregut al què també ha de realitzar per comprovar si el graf té cicles, pot tenir un cost molt elevat sobretot per aquells grafs de gran mida. Hi hauria un problema d’eficiència.

```
public static boolean teCicles(Xarxa x, Origen nodeOrigen)
{
   Graph subGraf = x.crearSubGraf(nodeOrigen);
   Node primerNode = subGraf.getNode(nodeOrigen.id());

   Set<Node> visitat = new HashSet<>();
   return dfsCicles(subGraf, primerNode, null, visitat);
}
```

```
private static boolean dfsCicles(Graph graf, Node node, Node pare, Set<Node>
visitats)
{
   visitats.add(node);
   Iterator<Edge> iterator = node.edges().iterator();
   while (iterator.hasNext()) {
       Node adjacent = iterator.next().getOpposite(node);
       if (!visitats.contains(adjacent)) {
           if (dfsCicles(graf, adjacent, node, visitats)) {
               return true;
           }
       } else if (!adjacent.equals(pare)) {
           return true;
       }
   }
   return false;
}
```

- Element 3

Un altre cop, per intentar millorar en termes d'eficiència, es podria fer dos if's, perquè hi ha una condició que implica mirar el cabal del node i que sigui més gran que 0, donant prioritat a saber si és un node *Terminal* quan aquest és només una operació i no implica un recorregut.

```
public static float cabalMinim(Xarxa x, Origen nodeOrigen, float
percentatgeDemandaSatisfet)
{
   float cabalMin = 0;

   Graph subGraf = x.crearSubGraf(nodeOrigen);
   for(Node node : subGraf.nodes().toList()){
       NodeAixeta nodeAixeta = x.node(node.getId());
        //si terminal oberta i li arriba cabal(no hi ha aixetes tancades per
sobre)
        if(nodeAixeta.aixetaOberta() && x.cabal(nodeAixeta)>0 && nodeAixeta
instanceof Terminal) {
                                    cabalMin += x.demanda(nodeAixeta) *
(percentatgeDemandaSatisfet/100);
       }
   }

   return cabalMin;
}
```

- Element 4

Aquí es pot veure que la següent funció conté *fragilitat*, perquè la condició de si el node *n* és de classe origen (atribut donat per la realització del dibuix), no hauria d'importar segons com estar dissenyada la *Xarxa*, però afecta el resultat del càlcul del cabal.

```
public float cabal(NodeAixeta node) {
    if (_xarxa.getNode(node.id()) == null) {
        throw new NoSuchElementException("Aquest node no pertany a la xarxa");
    }

    if(node instanceof Origen origen) {
        Node n = _xarxa.getNode(node.id());
        if(n.getAttribute("ui.class").equals("origen")) {
            if (origen.cabal() <= demanda(origen)) {
                return origen.cabal();
            } else {
                return demanda(origen);
            }
        }
    }

    float cabalTotal = 0;
    Iterator<Canonada> pipes = entrades(node);
    while (pipes.hasNext()) {
        Canonada pipe = pipes.next();
        cabalTotal += calculCabalCanonada(pipe);
    }

    return cabalTotal;
}
```

- Element 5

El següent cas hi hauria un mal disseny d'*opacitat*, perquè la falta de documentació a "cada" línia, pot donar dificultat a l'hora d'entendre el codi i esdevindrà a ser més opac a mesura que envelleix.

```
public static Set<NodeAixeta> aixetesTancar(Xarxa x, Map<Terminal, Boolean>
aiguaArriba)
{

    Set<NodeAixeta> tancarAixetes = new HashSet<>();
    //Buscar aixetes trencades
     for (Map.Entry<Terminal, Boolean> ItAiguaArriba : aiguaArriba.entrySet())
{
        Terminal terminal = ItAiguaArriba.getKey();
        Boolean aiguaArribaValor = ItAiguaArriba.getValue();
```

```
        if(!aiguaArribaValor && terminal.aixetaOberta()) {//si no arriba aigua
            Iterator<Canonada> entradesT = x.entrades(terminal);
            while (entradesT.hasNext()) { //mirem aixetes superiors
                 tancarAixetes.add(entradesT.next().node1()); //guardem aixeta
trencada
            }
        }

    }
    //buscar nodes fills que no estan trencats
    Set<NodeAixeta> tancarAixetesFills = new HashSet<>();//set que serveix per
saber els nodes fills
    Iterator<NodeAixeta> iterator = tancarAixetes.iterator();
    while (iterator.hasNext()) {
        NodeAixeta aixeta = iterator.next();
        eliminarFills(tancarAixetes, aixeta, x,tancarAixetesFills);
    }
    //treiem els nodes fills
    tancarAixetes.removeAll(tancarAixetesFills);
    return tancarAixetes;
}
```

Elements dels Principis SOLID

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

- Element 1

Single Responsibility Principle (SRP) En el nostre codi actual, la classe Xarxa està manejant diverses responsabilitats, com ara la gestió de nodes, canonades, aixetes, etc. Per millorar això, podríem dividir aquesta classe en classes més petites, cadascuna amb una única responsabilitat. Això faria que el codi fos més fàcil de mantenir i entendre.

- Element 2

Liskov Substitution Principle (LSP) En el nostre codi estem utilitzant l'herència de manera adequada, ja que Origen, Terminal i Connexio són subclasses de NodeAixeta i poden ser substituïdes per NodeAixeta sense problemes. Això és un bon exemple de l'aplicació del principi de substitució de Liskov.

- Element 3

Open-Closed Principle (OCP): Aquest principi diu que el codi ha de ser obert per a l'extensió, però tancat per a la modificació. En el nostre codi, els mètodes com *afegir(Origen nodeOrigen)*, *afegir(Terminal nodeTerminal)* i *afegir(Connexio nodeConnexio)* podrien violar aquest principi, ja que si es vol afegir un nou tipus de node, s'hauria de modificar la classe Xarxa. Una possible solució seria utilitzar la sobrecàrrega de mètodes o el polimorfisme per a aquesta funcionalitat.

Exemples 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.

- Exemple 1

Aquest mètode actua com a getter per obtenir un node de la xarxa. No necessita cap modificació perquè està ben dissenyat: retorna null si el node no existeix, el que és una pràctica comuna en Java.

```
public NodeAixeta node(String id) {
   Node n = _xarxa.getNode(id);

   return (n != null) ? (NodeAixeta) n.getAttribute("aixeta") : null;
}
```

- Exemple 2

Aquest mètode actua com a setter per afegir un client a la xarxa. Es podria modificar el return perquè retorni un booleà més informatiu a l'hora de saber si el client ja estava abonat a la xarxa o la crida d'una funció externa perquè faci aquesta feina.

```
public boolean abonar(String idClient, Terminal nodeTerminal) {
   if(_xarxa.getNode(nodeTerminal.id()) == null)
   {
       throw new NoSuchElementException("El node no pertany a la xarxa");
   }

   //Igual a null, no existia client, si diferent retornaria el nodeTerminal
   return _clients.putIfAbsent(idClient, nodeTerminal) == null;
}
```

- Exemple 3

Aquest mètode actua com a setter per afegir un node origen a la xarxa. No necessita cap modificació, ja que fa una bona comprovació per assegurar-se que no s'afegeix un node amb un identificador que ja existeix a la xarxa. A més, estableix correctament els atributs del node.

```
public void afegir(Origen nodeOrigen) {
   String id = nodeOrigen.id();
   if(_xarxa.getNode(id) != null)
   {
       throw new IllegalArgumentException("L'origen amb id " + id + " ja
existeix");
   }

   Node n = _xarxa.addNode(nodeOrigen.id());
   n.setAttribute("ui.label", nodeOrigen.id());
   n.setAttribute("aixeta", nodeOrigen);
   n.setAttribute("ui.class", "origen");
}
```