Curs 20-21

Unitat 06.01 Proves

Entorns de desenvolupament

IES Jaume II El Just
Tavernes de la Valldigna

Contingut

Introducció

Hem vist que, al llarg del procés de codificació, hem de detectar els possibles errors de sintaxi i també la forma de solucionar-los. En eixe procés tenim l'ajuda de l'IDE, que ens avisa si posem una instrucció malament, ens deixem algun parèntesi o clau per tancar, si intentem utilitzar una variable declarada, etc. També ens avisa dels errors de compilació que no pot detectar en temps d'edició.

Ara bé, què passa amb els errors lògics que no van en contra de la sintaxi del llenguatge? Eixe tipus d'error no són detectats per l'IDE, ni tant sols en temps de compilació. Són sentències que sintàcticament són correctes, però els seus resultats no són els que nosaltres teníem en ment quan les hem introduït en l'aplicació. En la pràctica això provoca errors que no trenquen l'execució del programa, però sí que ens fan obtindre resultats erronis.

Com podem detectar eixos errors lògics, si no ho poden fer ni l'IDE, ni el debugger, ni el propi compilador? L'única opció es realitzar **proves de funcionament**.

Fase de proves

En el cicle de vida d'una aplicació és molt important que no ens saltem la fase de proves. És el moment en el que hem de comprovar que la nostra aplicació no només no dona error quan compila o executa, sinó que fa exactament allò que nosaltres, i l'usuari final al que va destinada l'aplicació, volem que faça.

En la fase de proves podem fer dos tipus d'actuacions per comprovar el funcionament lògic de l'aplicació:

- **Proves de verificació**
  Es tracta de comprovar que el programa fa tot allò que ens han demanat i no ens hem deixat res
- **Proves de validació**
  Comprovem que eixes tasques que li hem dit que faça, les fa bé segons les especificacions rebudes per part de l'usuari en la fase d'anàlisi

Tècniques per a la fase de proves.

En funció de la tècnica utilitzada per realitzar les proves, podem fer diferents classificacions. Una d'elles, tal vegada la més important, és dividir les proves en:

- **Proves de caixa negra**
- **Proves de caixa blanca**

La diferència entre unes i altres està, com ara veurem, en el detall amb el qual se porten a terme i en el punt de vista que adoptem quan les estem realitzant.

Proves de caixa negra

Les proves de caixa negra se diuen així perquè el mètode, funció o classe que estem provant no ens mostra el seu interior. És a dir, estem fent proves per a les quals no necessitem conèixer de quina manera està programat l'element testejat. No necessitem accedir al seu codi font per a fer este tipus de proves.

Podríem dir que són proves que pot fer l'usuari final, perquè només se tracta de provar els diferents components de l'aplicació provant diferents entrades per veure si les eixides que obtenim són les esperades segons els requeriments d'anàlisi i el disseny dels components. També aprofiten per comprovar si la interfície d'usuari funciona correctament, si l'accés a les bases de dades o altres estructures de dades és la correcta, etc.

Proves de caixa blanca

Les proves de caixa blanca són diferents, perquè se fan amb total coneixement de com està programat el component analitzat. Se fan, per tant, des del punt de vista del programador, almenys d'una persona que coneix el codi font de l'element que està provant. Perquè és important fer este tipus de proves? Perquè en conèixer el codi font, estem en condicions de **provar tots els camins possibles** que pot agafar l'execució del component, cosa que no passa amb les proves de caixa negra.

Una altra classificació de les proves.

Podem fer una altra classificació de les proves una miqueta més tècnica, atenent a què és el que volem provar en cada cas.

Proves de càrrega, de resistència o d'estrès

Se solen fer en aplicacions multiusuari. La idea és provar què passa amb el rendiment i funcionament de l'aplicació quan hi ha molts usuaris treballant al mateix temps o amb els mateixos mòduls de l'aplicació. Serveixen per detectar problemes de concurrència a les dades, i també problemes de rendiment quan la càrrega de treball és molt alta.

Proves estructurals

Les proves estructurals coincidirien amb les que abans hem anomenat "proves de caixa blanca". El seu objectiu és comprovar si tots els camins possibles que pot seguir l'execució d'un component porten a un funcionament correcte.

Proves funcionals

Per la seua banda, les proves funcionals coincidirien amb les que hem anomenat com a "proves de caixa negra", perquè no cal conèixer el codi font que estem provant. Això sí, hem de saber

com funciona el component que estem provant a efectes de mòdul encapsulat: quina informació li hem de passar, i quina informació ens ha de tornar. Bàsicament consisteixen en provar les entrades i les eixides de cada mòdul, funció o mètode.

Proves funcionals

Les proves funcionals són, com hem comentat, proves de caixa negra. Se fan des del punt de vista de l'usuari, més que del programador. Hem de comprovar, principalment, si el funcionament o la eixida que dóna el programa (o funció / mètode) a qualsevol conjunt de dades d'entrada és correcta.

En este tipus de proves no comprovem la velocitat, ni l'eficiència, només si les respostes a cada conjunt de dades d'entrada són les correctes.

Per tal de realitzar les proves funcionals intentant no deixar-nos cap possibilitat que puga ser important, hi ha diferents tècniques. Unes tècniques que, principalment, se centren en intentar que els jocs de dades amb els quals provem el component tinguen la màxima amplitud i varietat, de manera que tinguem més possibilitats de trobar errors.

Totes les tècniques tenen en comú que hem de seleccionar un joc de dades per fer les proves. És a dir, hem de definir **casos de prova**. Un cas de prova és un conjunt d'entrades, condicions d'execució i resultats esperats. La idea és que, per cada joc d'entrades, hem de saber almenys quin hauria de ser el resultat correcte. Si no, no podríem provar res.

Les principals estratègies per a triar els valors d'entrada amb els quals provar el component que volem testejar són la utilització de **classes d'equivalència**, de **valors límit**, i de **valors aleatoris**.

Classes d'equivalència

Amb les classes d'equivalència s'intenta crear un conjunt de dades de prova que siguen equivalents a efectes del funcionament del component. És a dir, que el funcionament del mòdul, i de vegades el resultat final, siga el mateix per a totes les dades que formen part de la classe.

Valors límit

És una estratègia que va lligada a la utilització de classes d'equivalència. La idea és que si una classe té molts valors, i no podem provar-los tots un per un, intentar provar com a mínim els que estan en el límit de la classe, que són els que poden donar problemes una vegada comprovat un conjunt més o menys significatiu de la resta de dades que formen la classe.

Valors aleatoris

Si no hi ha uns valors que, en principi, tinguen més opcions de causar problemes que altres, o tenim només una classe d'equivalència, podem optar per l'estratègia de utilitzar valors generats aleatòriament per fer les proves.

Classes d'equivalència

Com podem definir quines serien les classes d'equivalència òptimes per provar un component d'una aplicació? En general es tracta de definir subconjunts de possibles dades d'entrada el comportament del qual, podem suposar, serà el mateix. Ho veurem millor amb alguns exemples típics:

- Si un paràmetre d'entrada ha d'estar dins d'un interval determinat, les possibles classes d'equivalència podrien ser:
  - Dades que estan en l'interval
  - Dades que estan fora de l'interval per dalt
  - Dades que estan fora de l'interval per baix
- Si un paràmetre d'entrada ha de pertànyer a un conjunt determinat de dades, o a un tipus de dades específics, com a mínim tindríem dues classes d'equivalència:
  - Dades que pertanyen al conjunt
  - Dades que no pertanyen
- Si una entrada ha de ser booleana (true or false), ho tenim fàcil: només hi ha dues classes d'equivalència:
  - Condició d'entrada que és verdadera (true)
  - Condició d'entrada que és falsa (false)

Exemple pràctic de prova funcional.

Imaginem un mòdul que rep com a paràmetre les dades d'un usuari i ha de prendre una decisió en funció de la seua edat. No sabem internament com està programat el mòdul, però sí que sabem que farà una cosa diferent en cadascun dels següents casos:

- Si l'usuari és menor d'edat
- Si l'usuari és major d'edat però té menys de 66 anys
- Si l'usuari té 66 anys o més

Quines serien les classes d'equivalència? Bé, sabem que l'eixida del mòdul serà la mateixa o molt similar en cadascun dels tres casos que hem vist. Per tant, hauríem de provar valors que pertanyen a cadascun dels tres casos. Les classes d'equivalència per a les dades d'entrada amb les quals provarem el mòdul serien:

- Números enters entre 0 i 17
- Números enters entre 18 i 65
- Números enters majors que 65

I una vegada establertes les classes d'equivalència, quins valors hem de provar? Tots els de cada classe?

En realitat seria ideal provar-los tots, però moltes vegades no serà possible. En eixe moment és quan acudim a l'estratègia dels **valors límit**. Recordeu que els valors límit són els que estan en el límit entre dues classes d'equivalència. En el nostre cas serien els següents: 0, 17, 18, 65 i 66.

Perquè és important provar els valors límit?

Anem a veure-ho fent una ullada al possible codi font, encara que, recordem, estem fent proves de caixa negra i no tenim accés al codi. Però precisament per això se'ns poden passar errors com el que anem a veure. Imaginem que el codi font hauria de dir:

```
if(edat >=0 && edat <18) { ... // què fer amb els menors d'edat  }
else if(edat >=18 && edat <=65) { // què fer amb els majors d'edat menors de 66 }
else { // què fer amb els majors de 65 anys }
```

però en realitat el codi diu:

```
if(edat >=0 && edat <=18) { ... // què fer amb els menors d'edat  }
else if(edat >=18 && edat <=65) { // què fer amb els majors d'edat menors de 66 }
else { // què fer amb els majors de 65 anys }
```

Detecteu l'error?

Efectivament, si posem en la primera condició `edat <=18` el que estem fent, sense voler, és tractant a la gent que té 18 anys com si foren menors d'edat. I si no tenim accés al codi font, com podem detectar eixe fallo? Bé, només hi ha una manera, i es provant què passa quan posem el valor 18.

Per això és important provar els valors que estan en el límit d'una classe, perquè solen ser els que s'utilitzen en les condicions i, per tant, els que si no se posen bé els operadors de comparació o els valors, poden provocar una eixida incorrecta. Fixeu-vos que si provem per exemple els valors 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75 80 estem fent una bona quantitat de proves i, tot i això, no detectaríem l'error perquè no estem provant el valor 18.

Quins valors més hauríem de provar? Una vegada provats els valors vàlids (números enters positius), una bona estratègia és provar valors que no estan dins d'eixe subconjunt de dades. Per exemple:

- Què passa si pose un número negatiu?
- Què passa si pose lletres en lloc de números en un possible formulari d'entrada?
- Què passa si pose números amb decimals? Me deixa? Té influència en el resultat?

Proves estructurals

Són proves de caixa blanca, per tant hem de conèixer el funcionament intern de l'aplicació. Se pretén, com ja hem comentat, que cada possible camí que pot agafar l'execució de l'aplicació siga comprovat almenys una vegada. Se comprova també que no hi ha codi que no s'utilitza, camins que mai se recorren, etc.

Hauríem de seguir certes indicacions per abarcar totes les opcions possibles, cosa que tenim fàcil perquè, recordem, tenim accés al codi font.

- **Cobertura de sentències**: cada instrucció del mòdul que estem provant s'ha d'executar almenys una vegada.

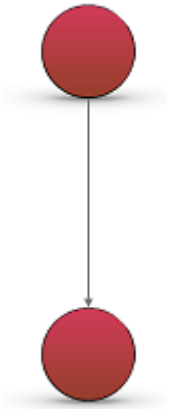
- **Cobertura de decisions i condicions**: en les condicions avaluades i les decisions que se prenen en funció de les condicions s'han de provar totes les opcions (condició verdadera, condició falsa, etc.)
- **Cobertura de camins**: ens hem d'assegurar que tots els possibles camins que pot seguir l'execució, des del punt d'entrada de les dades fins el punt d'eixida del mòdul, se recorren almenys una vegada.

Complexitat ciclomàtica

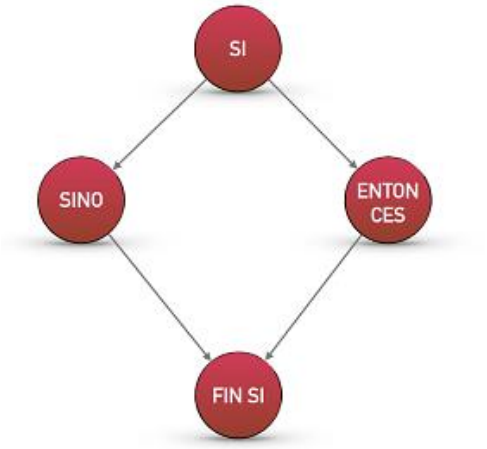
En mòduls extensos o complicats és important tindre una mida de la seua complexitat per definir els possibles camins que pot seguir la seua execució sense saltar-ne cap.

Per obtindre la complexitat ciclomàtica d'un mòdul utilitzem **grafs**.

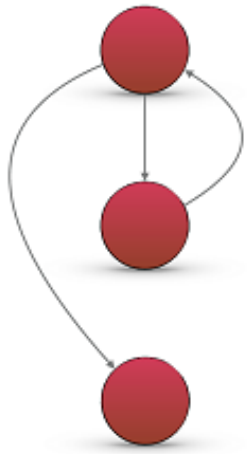
Per exemple, açò seria un graf seqüencial que representa un camí directe entre una instrucció i la següent.



I este seria un graf condicional que representa que, en un moment donat, el camí se pot bifurcar en funció d'una o vàries condicions.



I el següent seria un graf que representa un bucle. És a dir, que el camí seguit per l'aplicació, en un moment donat, pot tornar a un punt anterior i repetir-se, o fins i tot agafar un altre camí diferent.



Cada cercle és un node que representa una o més sentències. Les fletxes representen el flux del programa, unint uns nodes amb altres.

La **complexitat ciclomàtica d'un mòdul** ens diu la quantitat de camins independents que pot seguir el mòdul i, per tant, també ens dirà la quantitat mínima de casos de prova que necessitarem per cobrir totes les opcions.

La complexitat ciclomàtica se pot calcular de vàries formes:

- Quantitat de regions delimitades per les fletxes i els nodes (s'inclou la regió exterior)
- Quantitat de regions tancades per fletxes i nodes + 1
- Fletxes – nodes + 2
- Nodes dels quals ix o arriba més d'una fletxa (perquè tenen una condició) + 1

El resultat hauria de ser el mateix en cada cas.

En funció del valor de la complexitat ciclomàtica que obtenim, podem classificar una aplicació o un mòdul dins d'uns paràmetres que avaluen el risc d'error:

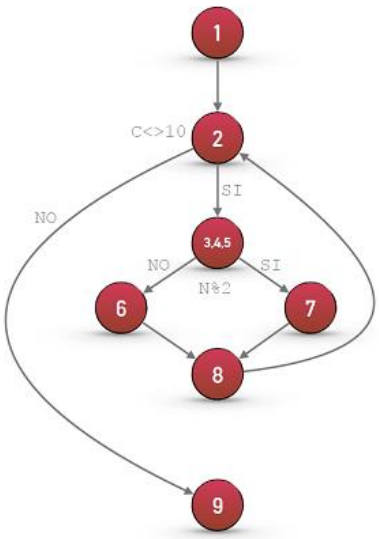
Complexitat ciclomàtica	Avaluació del risc
Entre 1 i 10	Programa o mètode molt senzill, no hi ha pràcticament risc
Entre 11 i 20	Programa o mètode una miqueta més complex, amb risc moderat
Entre 21 i 50	Programa o mètode complex, amb risc alt
Més de 50	Programa o mètode pràcticament molt complicat de testejar, risc molt alt

Exemple de càlcul de la complexitat ciclomàtica

Suposem que tenim una aplicació que demana 10 números enters i, al final, ens diu quants hi ha impars i quants hi ha pars. L'aplicació, en llenguatge Java, podria ser la següent:

```
public static void main(String[] args) {
        int comptador=0;
        int pars=0;
        int impars=0;
        int n;
        Scanner entrada=new Scanner(System.in);
        while(comptador!=10) {
            System.out.print("Tecleja un número enter: ");
            n=entrada.nextInt();
            if(n%2==0) {
                pars++;
                System.out.println("El número "+n+" és par");
            }
            else {
                impars++;
                System.out.println("El número "+n+" és impar");
            }
            comptador++;
        }
        System.out.println("Has introduit "+comptador+" números dels quals hi ha "+
                pars+" que són pars i "+impars+" que són impars");
    }
```

El graf corresponent seria així:



Aplicant qualsevol de les fórmules podeu comprovar que la complexitat ciclomàtica del programa és 3. Això vol dir que hi ha tres possibles camins principals que pot seguir l'execució, i que per provar-los tots hem de triar tres casos de prova:

- Camí 1: node 1 → node 2 → node 9
- Camí 2: node 1 → node 2 → node 3,4,5 → node 6 → node 8 → node 2 → node 9
- Camí 3: node 1 → node 2 → node 3,4,5 → node 7 → node 8 → node 2 → node 9

Els casos de prova, per tant, serien:

Camí	Cas de prova	Resultat esperat
1	Quan acabem d'introduir els 10 números	Mostrarà quants hi ha pars i quants impars, i acabarà
2	Quan teclegem un número impar	Incrementarà el comptador dels números impars i mostrarà un missatge dient que el número és impar
3	Quan teclegem un número par	Incrementarà el comptador dels números pars i mostrarà un missatge dient que el número és par

En la pràctica, com hauríem de provar l'aplicació? Lògicament, posant números pars e impars. Si hi ha 2 camins possibles una vegada introduïm el número, si només teclegem números pars o només impars només provem un camí. Seria ideal, ja que són 10 números, teclejar-ne 5 pars i 5 impars. Encara que si funciona bé amb un par i un impar hauria de funcionar bé amb tots.

No oblidem que estem parlant de casos de prova amb valors vàlids, i que també **hauríem de provar valors no vàlids** o que estiguen fora de l'interval esperat (números enters positius, per exemple). És a dir:

- Què passaria si posem el 0? El compta com a par o impar?
- Què passaria si posem un número amb decimals?
- Què passaria si posem lletres o altres símbols no numèrics?

Eixes proves de valors no vàlids ens serviran per detectar on hem de capturar excepcions, o si hem de filtrar d'alguna manera els valors d'entrada.

Exemple de prova estructural

Imaginem que tenim el següent codi:

```
int prova (int x, int y) {
   int z=0;
   if(x>0 && y>0) {
       z=x;
   }
   return z;
}
```

Segons el que hem vist en els tipus de cobertura, hauríem de dissenyar un joc de proves de manera que:

- Comprovem que la funció se crida alguna vegada

- Cobertura de sentències: hauríem de fer almenys una prova amb valors de x i y que facen que s'execute el contingut de l'if. O siga, que els dos, x i y, siguen majors que 0.

- Cobertura de condicions i decisions: hauríem de provar:

    - Què passa si els dos paràmetres són majors que 0 (s'hauria d'executar z=x i per tant tornar el valor de x)

    - Què passa si només un dels dos paràmetres és major o igual que 0, primer x i després y (no s'hauria d'executar z=x i per tant hauria de tornar 0)

    - Què passa si els dos paràmetres són menors o iguals que 0 (no s'hauria d'executar z=x i per tant hauria de tornar 0)

Altres tipus de proves

Proves d'integració

Serveixen per comprovar la interacció entre els diferents mòduls. Podem provar cada mòdul per separat (amb les proves que hem vist anteriorment), i ara provar tota l'aplicació en conjunt.

Proves de regressió

En realitat no són un tipus de proves. Les **proves de regressió** són les que se fan, una vegada comprovada la correcta integració dels mòduls, quan se modifica un mòdul per veure si eixa modificació afecta a cap altre.

Es tracta simplement de repetir les proves que ja s'han fet en els mòduls que poden veure afectat el seu funcionament pels canvis que s'han produït en el mòdul modificat.

En les proves de regressió normalment només utilitzarem proves de caixa negra.

Proves de recuperació

Se força un error per veure si el sistema reacciona de manera adequada (tractant una excepció, desfent les transaccions, etc.) i si se recupera bé de l'error.

Proves de seguretat

Se verifica que l'aplicació està protegida contra accessos indeguts, que la identificació dels usuaris i la comprovació dels permisos funciona correctament, i en general tots els aspectes relacionats amb la seguretat externa.