Persecució als sòtans de palau
o Soterranis, Tropes, Guarnicions i Onades

Pràctica de PRAP

Daniel Clemente Laboreo (grup 33)
DNI 45786256-H
www.danielclemente.com

26 Abril 2004

Seccions

1. Enunciat....1
   1.1 Exemple....1
   1.2 Es demana....1
2. Descripció general....2
3. Esquema modular....3
4. Especificació dels mòduls....3
   4.1 Especificació de Soterrani....3
   4.2 Especificació de Guarnició....4
   4.3 Especificació de Onades....5
   4.4 Especificació de Tropa....6
5. Programa principal....6
6. Implementació dels mòduls....8
   6.1 Implementació de Soterrani....8
      6.1.1 Implementació de simula_sortida....8
      6.1.2 Implementació de simula_sortida_rec....9
      6.1.3 Implementació de nou_soterrani....10
   6.2 Implementació de Guarnició....10
      6.2.1 Implementació de nova_guarnició....11
      6.2.2 Implementació de bàndol....11
      6.2.3 Implementació de qualitat....12
      6.2.4 Implementació de quants....12
      6.2.5 Implementació de modificar....13
      6.2.6 Implementació de modifica_un....14
   6.3 Implementació de Onades....15
      6.3.1 Implementació de nova_onades....17
      6.3.2 Implementació de afegir_onada....17
      6.3.3 Implementació de afegir_onada_individual....17
      6.3.4 Implementació de suma_dist_a_tots....18
      6.3.5 Implementació de quantes....18
      6.3.6 Implementació de tropa....18
      6.3.7 Implementació de distància....19
      6.3.8 Implementació de fusiona....19
   6.4 Implementació de Tropa....21
      6.4.1 Implementació de nova_tropa....21
      6.4.2 Implementació de lluita....22
      6.4.3 Implementació de suma_dist....23
      6.4.4 Implementació de distància....23
7. Possibles millores....24
   7.1 Classe Soterrani no necessària....24
   7.2 Variables temporals....24
   7.3 Constructores....24
   7.4 Funcions de còpia....25

1. Enunciat

Al segle XVII, els sòtans del palau de Versalles eren el lloc on els mosqueters del Rei i la guàrdia del Cardenal lluitaven i es perseguien en una xarxa de sales i passadissos.

Considerarem *N* sales inicials on hi ha com a màxim *N* soldats: 1 per sala, mosqueter o guàrdia, i amb un nivell d'experiència característic. Han de dirigir-se cap a la sortida avançant per uns túnels en estructura d'arbre binari: a cada sala -excepte les inicials- hi arriben dos túnels de la mateixa longitud. Els soldats no poden tornar enrera, així que només avançaran dirigint-se cap a l'arrel de l'arbre, la sortida. Tots van a la mateixa velocitat, però alguns túnels són més llargs.

Si dos soldats o més arriben a la mateixa sala a la vegada després de recórrer els túnels (de la mateixa distància) es pot produir una lluita. Guanyarà el soldat o grup que sumi més experiència en total, sense perdre cap soldat. Si els dos grups eren del mateix bàndol, continuen junts. Una lluita no compta com a temps perdut, ni es solapa amb les altres.

S'utilitzarà la paraula 'guarnició' per referir-se a tots els soldats del soterrani, i 'onada' per parlar dels soldats que arriben a la mateixa sala a la vegada.

1.1 Exemple

A les fulles el número és un identificador de la sala inicial; als altres nodes el número indica la longitud dels dos passadissos que hi arriben a la sala.

```
1   4
 \ /
  1   3   2   5
   \ /     \ /
    1       1
     \     /
        1
    (sortida)
```

Suposem que hi ha mosqueters a la sala 1 i 3, i guàrdies a la 2 i la 5, tots de qualitat 1. A la 4 no hi ha ningú.

La simulació és aquesta: els dos guàrdies de la 2 i la 5 es troben i avancen junts, i quan es troben amb el mosqueter de la sala 3 (just abans de sortir) li guanyen i surten. L'altre mosqueter (el de la sala 1) no es troba amb ningú i surt sol. Per tant, primer surten dos guàrdies i després un mosqueter.

1.2 Es demana

Fer un programa modular i eficient, amb especificació, implementació i derivació informal (excepte de les classes d'entrada/sortida) i després codificar-ho en JAVA, utilitzant si cal les classes que hem estudiat durant el curs (*Pila*, *InOut*, *Arbre*, *Lavadora*, *Estudiant*, ...).

- Ha de permetre simular una sortida com a l'exemple, només imprimint quants soldats surten, de quin tipus, i en quin ordre.
- Ha de tenir una opció de modificar guarnició on se'n preguntin dues, es comprovi si són

compatibles (han de tenir el mateix nombre de soldats i a les mateixes sales) i es retorni una altra on s'agafi el millor soldat de cada sala, sense importar el bàndol (bé, en cas d'empat, guanyen els mosqueters).
- Ha de tenir l'opció de modificar l'estructura de soterrani introduiïda.
- Ha de validar correctament els jocs de proves donats: una certa entrada per al programa ha de donar una certa sortida, sense que coneguem tots aquests jocs de proves.

2. Descripció general

No fa falta simular en temps real què està passant a cada segon ni fer avançar els soldats pas a pas, ja que el que volem és només saber l'ordre en què sortiran alguns dels soldats, o sigui, una llista de l'estil “2 guàrdies, 1 mosqueter i 3 guárdies més”.

Tampoc fa falta guardar informació individual sobre cada soldat, ja que quan es junta un de qualitat *q1* amb un de *q2*, a partir d'ara es comporten com un sol de qualitat *q1+q2*. Per tant, no s'ha de parlar de 'soldats' sinó de 'grups de soldats', o, al meu programa, 'tropes'.

Seria mala idea fer que l'estructura del soterrani sigui un arbre de sales on cada sala té soldats, ja que la majoria de sales estan buides i hi estaríem consumint memòria innecesàriament. L'únic útil que cal gravar a cada sala és la longitud dels dos passadissos que hi arriben a ella (que, per sort, són iguals). En el cas de les sales inicials no fa falta aquesta dada, així que es gravarà un número identificador.

Per últim, la disposició inicial dels soldats a les sales inicials no la podem gravar a dins de l'arbre, així que cal un objecte més que guardi la posició inicial de cada soldat.

Utilitzaré les següents classes:
- *Soterrani*: arbre d'enters que guarda l'estructura de sales i passadissos
- *Guarnició*: vector que relaciona cada soldat amb el número de sala inicial on estava
- *Onades*: llista ordenada de les tropes que arribaran a la sortida
- *Tropa*: per posar informació sobre un grup de soldats del mateix bàndol

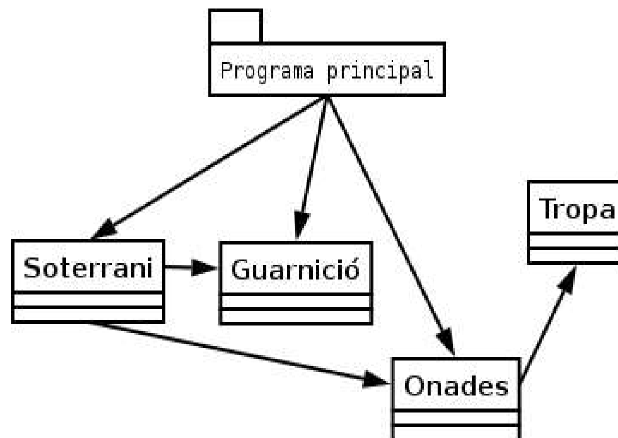
L'estratègia serà recórrer recursivament l'arbre començant per l'arrel fins arribar a les fulles (sales inicials). Quan es trobin, aquells soldats s'afegeixen com una onada que en principi arribarà a la sortida.

A mesura que es va retornant de la recursivitat, es va augmentant el comptador de la distància que separa a cada soldat de la sortida, i si es veu que dues onades estan a la mateixa distància de la sala actual, se'n elimina una ja que hi ha una lluita.

Anomeno 'lluita constructiva' a la trobada entre soldats del mateix bàndol: s'ajunten i continuen com un, i 'lluita destructiva' és quan són enemics i només sobreviuen els millors.

D'aquesta manera el programa es resumeix en cridar a aquesta funció recursiva passant-li l'arrel del soterrani i la guarnició, i com a resultat dóna la llista d'onades que aconseguiran sortir.

3. Esquema modular



A més, *Soterrani* depén d'*Arbre*, i el *Programa Principal* depén d'*IO* (operacions d'entrada/sortida) que també es relaciona amb moltes altres classes, com *Soterrani*, *Guarnició*, *Onades*, *Tropa*, *Arbre*, *ArbreIO* i *InOut*. No estan documentades en aquesta pràctica, i a més moltes ja les teníem fetes.

4. Especificació dels mòduls

El llenguatge algorísmic posat aquí no està adaptat a cap llenguatge en concret, i en traduir-ho poden canviar aspectes com el pas de paràmetres (implícits o no) i les operacions de creació/còpia/destrucció d'objectes. A més s'haurien d'eliminar els accents i canviar ç per c.

4.1 Especificació de Soterrani

El soterrani guarda l'estructura de les sales, la longitud dels passadissos, i el número identificador de cada sala inicial. Té l'algorisme més interessant de la pràctica, *simula_sortida*.

```
Mòdul Soterrani;


Especificació:

Sobre Arbre, Guarnició, Onades;

{ Tipus de mòdul: dades }

tipus Soterrani;

{ Descripció general: guarda l'estructura d'un soterrani, on les sales inicials
estan numerades i les altres contenen la longitud dels passadissos que hi
arriben }


Operacions:
```

```
funció simula_sortida (s: Soterrani, g: Guarnició) retorna o: Onades;
{ Pre: s i g contenen un soterrani i guarnició vàlides }
{ Post: o conté la llista d'onades que es produeixen al soterrani d'arrel a,
ordenades per distància }

funció nou_soterrani () retorna s: Soterrani;
{ Pre: - }
{ Post: s és un soterrani sense sales }
```

4.2 Especificació de Guarnició

Aquí hi ha l'altre algorisme important: modificar guarnició agafant el millor soldat de cada sala. Aquest mòdul no depén de *Tropa*, com explico a la implementació.

```
Mòdul Guarnició;


Especificació:

{ Tipus de mòdul: dades }

tipus Guarnició;

{ Descripció general: conté las característiques de tots els soldats inicials i
la sala a on es trobaven en començar }


Operacions:

funció nova_guarnició () retorna g: Guarnició;
{ Pre: - }
{ Post: g és una guarnició sense soldats }

funció bàndol (g: Guarnició, n: enter) retorna b: enter;
{ Pre: n<nombre_de_soldats_a_g }
{ Post: b conté el número de bàndol a què pertany el soldat de la sala n;
        si no n'hi ha cap, b=0 }

funció qualitat (g: Guarnició, n: enter) retorna q: enter;
{ Pre: n<nombre_de_soldats_a_g, la sala inicial n no està buida }
{ Post: q conté la qualitat del soldat de la sala n }

funció quants (g: Guarnició) retorna n: enter;
{ Pre: - }
{ Post: n és el número de soldats que hi ha a la guarnició g }

acció modificar (e[s g: Guarnició, e g1: Guarnició, e g2: Guarnició, s compat:
enter);
{ Pre: - }
{ Post: compat=1 => g té la unió de g1 amb g2,
                    agafant el millor soldat de cada sala;
        compat=0 => el valor de g no ha canviat, g1 i g2 són incompatibles }

acció modifica_un (e/s g: Guarnició, e n: enter, e bàndol: enter, e qualitat:
enter);
```

```
{ Pre: n<nombre_de_soldats_a_g, la sala inicial n no està buida }
{ Post: el soldat inicial número n s'ha modificat amb els valors demanats }
```

4.3 Especificació de Onades

Trobar la llista d'onades a la sortida és l'objectiu del programa. Aquesta llista s'anirà construint a mesura que processem el soterrani+guarnició des de les fulles fins l'arrel.

```
Mòdul Onades;


Especificació:

Sobre Tropa;

{ Tipus de mòdul: dades }

tipus Onades;

{ Descripció general: guarda la llista de tropes que arribaran a la sortida i
l'ordre en què ho faran }


Operacions:

funció nova_onades () retorna o: Onades;
{ Pre: - }
{ Post: o és una llista d'onades buida }

acció afegir_onada (e/s o: Onades, e t: Tropa);
{ Pre: a o encara hi cap una onada més }
{ Post: o conté un registre més amb la tropa t }

acció afegir_onada_individual (e/s o: Onades, e b: enter, e q: qualitat);
{ Pre: a o encara hi cap una onada més }
{ Post: o conté un registre més amb la onada d'un soldat de bàndol b i qualitat
q }

acció suma_dist_a_tots (e/s o: Onades, d: enter);
{ Pre: - }
{ Post: totes les tropes de o estan d passos més lluny de la sortida }

funció quantes (o: Onades) retorna n: enter;
{ Pre: - }
{ Post: n és el nombre de tropes de la llista o }

funció tropa (o: Onades, n: enter) retorna t: Tropa;
{ Pre: n<nombre_de_tropes_a_o }
{ Post: t és la tropa que hi ha a la posició n de la llista de tropes amb èxit }

funció distància (o: Onades, n: enter) retorna d: enter;
{ Pre: n<nombre_de_tropes_a_o }
{ Post: d és la distància a la que la tropa número n es troba de la sortida }

funció fusiona (o1: Onades, o2: Onades) retorna o: Onades;
```

```
{ Pre: o1 i o2 ordenades per distància }
{ Post: o té les onades de o1 i de o2, i ha processat les lluites on les
distàncies de les tropes coïncidien, i o està ordenada per distància }
```

4.4 Especificació de Tropa

Una tropa és un grup de soldats (1 o més) amb les seves característiques com a grup. A més, aquí és on es grava com de lluny està cada grup de la sortida del soterrani, i també és en aquest mòdul on es programaran les condicions de les lluites.

```
Mòdul Tropa;


Especificació:

{ Tipus de mòdul: dades }

tipus Tropa;

{ Descripció general: guarda informació sobre un soldat o grup de soldats }


Operacions:

funció nova_tropa (bàndol: enter, qualitat: enter) retorna t: Tropa;
{ Pre: - }
{ Post: t és la tropa composta pel soldat de bàndol i qualitat seleccionats }

funció lluita (t1: Tropa, t2: tropa) retorna t: Tropa;
{ Pre: les dues tropes estaven a la mateixa distància }
{ Post: t és el resultat de la trobada entre t1 i t2
        (constructiva si són amics o destructiva si enemics) }

funció suma_dist (t: Tropa, d: enter) retorna t2: Tropa;
{ Pre: la tropa t té soldats }
{ Post: t2 él la tropa resultant d'incrementar la distància de la tropa }

funció distància (t: Tropa) retorna d: enter;
{ Pre: la tropa t té soldats }
{ Post: d és la distància a la que la tropa t es troba de la sala actual }
```

5. Programa principal

La implementació del programa és molt senzilla i similar a les que ja hem fet a classe. És aquí on s'utilitzen totes les funcions d'entrada/sortida, que suposarem que funcionen bé i que no fan buffer overflows com el *InOut* de la FIB. El programa és:

```
Programa principal;

Sobre Soterrani, Guarnició, Onades, IO;

// Aquestes funcions/accions són a la classe IO:
// llegir_soterrani, llegir_guarnició, llegir_enter
```

```
// mostrar_onades, mostrar_enter


{ Descripció general: presenta un menú amb les opcions disponibles }


var
  op: enter;
  s: Soterrani;
  g: Guarnició;
  o: Onades;

  g1: Guarnició; // per a l'opció de modificar
  g2: Guarnició; // per a l'opció de modificar
  compatibles: enter;

fvar

s := nou_soterrani();
g := nova_guarnició();

llegir_soterrani(s);
llegir_guarnició(g);

op := llegir_enter();

mentre op != -15 fer

  si op = -10 --> o := simula_sortida(s,g);
                  mostrar_onades(o);

  [] op = -11 --> llegir_guarnició(g1);
                  llegir_guarnició(g2);
                  modifica(g,g1,g2,compatibles);
                  mostrar_enter(compatibles);

  [] op = -12 --> llegir_soterrani(s);

  fsi;

  op := llegir_enter();

fmentre;
```

Abans de mostrar el menú es llegeix l'estructura del soterrani i la guarnició. Després, s'entra en un cicle fins que es surt amb l'opció -15, i llavors el programa acaba. Les opcions són tres, molt senzilles:

- **-10**: persecució amb l'estructura i guarnició actuals
- **-11**: modificar guarnició a partir de dues donades
- **-12**: modificar estructura del soterrani actual

Qualsevol cosa diferent de -10, -11, -12 o -15 s'ignora.

La crida a *simular_sortida* es fa passant un *Soterrani* i una *Guarnició*, però després la rutina

recursiva no treballa amb soterranis sinó amb arbres directament, com es veurà a la implementació. S'ha fet això per tenir més modularitat i no haver de passar a la funció un *Arbre* des del programa principal.

6. Implementació dels mòduls

Aquí es presenta no només el pseudocodi de les funcions anteriors sinó la seva justificació informal. Primer surten els detalls del tipus corresponent a la classe, i després l'explicació de cada funció, en el mateix ordre que a l'especificació.

La constant *N* estarà definida a cada mòdul on sigui necessària. Es va proposar *N = 20*.

6.1 Implementació de Soterrani

Utilitzo un arbre on cada node té un enter. Aquest número identifica a cada sala inicial si els nodes són les fulles de l'arbre, o guarda la longitud dels dos passadissos que arriben a la sala si és no inicial.

```
Mòdul Soterrani;


Implementació:

Sobre Arbre, Guarnició, Onades;

// Aquestes funcions/accions són a la classe Arbre
// és_nul, fill_esq, fill_dret, arrel, arbre_nul

tipus Soterrani =
  estructura: Arbre d'enter;


Operacions:
```

6.1.1 Implementació de simula_sortida

Aquesta funció només encapsula una crida a la rutina recursiva, que treballa passant-se arbres com a paràmetre (i no soterranis). Es podria fer tot amb objectes *Soterrani*, però el codi seria més complex.

Complim la precondició ja que l'arbre del soterrani i la guarnició són vàlids. Encara que sigui privada, la podem cridar perquè estem a la mateixa classe.

```
funció simula_sortida (s: Soterrani, g: Guarnició) retorna o: Onades;
{ Pre: s i g contenen un soterrani i guarnició vàlides }

  retorna simula_sortida_rec(s.estructura, g);

{ Post: o conté la llista d'onades que es produeixen al soterrani d'arrel a,
ordenades per distància }
ffunció;
```

6.1.2 Implementació de simula_sortida_rec

Aquest és l'algorisme important del programa, una funció recursiva múltiple que crea i retorna una llista d'onades a cada crida a partir de l'arbre i guarnició passats. Aquesta llista la podem crear de dos maneres diferents:

- **Cas directe**: la sala és inicial perquè els seus dos fills són l'arbre nul (de fet, si un d'ells és nul ja sabem que serà inicial). En aquest cas no cal endinsar-se més en el laberint perquè no hi ha sales més llunyanes. A aquesta sala pot haver-hi un o cap soldat; ho hem de consultar a la guarnició i, si hi ha algú, l'afegim a la llista d'onades ja que -en principi- sortirà del soterrani.
- **Cas recursiu**: la sala no és inicial, així que la disposició de soldats que hi arribin depén de les sales més internes. Haurem de calcular quines tropes vinen per l'esquerra i quines per la dreta i fusionar-les, o sigui, fer que lluitin les que anaven a coincidir. A totes elles se les suma la distància que acaben de recórrer per arribar a la sala, que és el valor que teníem guardat a l'arbre d'enters. També podria primer fusionar i després sumar la distància, que és una mica més eficient però no és l'ordre lògic (avancen, es troben i lluiten).

En ambdós casos acabem tornant una llista d'onades, amb un element si només hem trobat un soldat a la sala inicial, amb cap si aquesta estava buida, o amb les tropes amb èxit que hi havia a les sales anteriors.

Al cas recursiu continuem complint la precondició ja que si la sala no és inicial, sabem que tant el fill esquerre com el dret no són nuls i tenen una estructura vàlida. En descartar cada vegada l'arrel de l'arbre que s'ha passat, ens estem apropant cada vegada més cap a les sales inicials, que és el cas directe on no es fa cap crida més. La funció de cota és l'altura de l'arbre, que decreix fins arribar a 1.

La guarnició no té sentit modificar-la ja que ens parla de la situació inicial dels soldats, que no canviarà en tota la simulació.

```
funció privada simula_sortida_rec (a: Arbre, g: Guarnició) retorna o: Onades;
{ Pre: a i g contenen estructures de soterrani i guarnició vàlides }

  var valor: enter; fvar

  nova_onades(o);

  valor := arrel(a);

  si és_nul(fill_esq(a)) o és_nul(fill_dret(a)) --> // és sala inicial

    si bàndol(g, valor) != 0 --> // si hi ha un soldat a aquesta sala
      afegir_onada_individual(o, bàndol(g, valor), qualitat(g, valor));
    fsi;

  sinó --> // mesclar els dos fills
    var o1, o2: Onades; fvar

    o1 := simula_sortida_rec(fill_esq(a), g);
    o2 := simula_sortida_rec(fill_dret(a), g);
```

```
    suma_dist_a_tots(o, valor);

    o := fusiona(o1, o2);

  fsi;

  retorna o;

{ Post: o conté la llista d'onades que es produeixen al soterrani d'arrel a,
ordenades per distància }
ffunció;
```

6.1.3 Implementació de nou_soterrani

Ens retorna un arbre buit i que no és vàlid perquè no té cap sala. Per utilitzar-ho al programa haurem de cridar a una funció d'entrada/sortida d'arbres, com *llegir_soterrani*, que és a la classe *IO*.

```
funció nou_soterrani () retorna s: Soterrani;
{ Pre: - }

  s.estructura := arbre_nul();

  retorna s;

{ Post: s és un soterrani sense sales }
ffunció;
```

6.2 Implementació de Guarnició

El vector que guarda les dades de cada soldat no és un vector de *Tropa* sinó de una tupla *SoldatInicial*, que és privada i no s'ha d'utilitzar des de l'exterior. Utilitzo una tupla en comptes de *Tropa* perquè aquestes estructures serveixen per guardar dades diferents.

Sabem que com a màxim hi ha *N* soldats, així que el vector és de *N* posicions. De totes maneres, no sempre hi seran els *N* (l'usuari pot posar-hi menys), i hem de gravar aquest número com un enter, que no canviarà durant la simulació.

Els elements estan a la matriu al mateix ordre amb què els ha posat l'usuari.

```
Mòdul Guarnició;


Implementació:

tipus Guarnició = tupla
                    soldats: vector [0..N-1] de SoldatInicial;
                    númSoldats: enter;
                  ftupla;

tipus privat SoldatInicial = tupla
                               salaInicial: enter;
                               bàndol: enter;
                               qualitat: enter;
```

```
                              ftupla;


Operacions:
```

6.2.1 Implementació de nova_guarnició

Retorna una guarnició amb cap soldat (no vàlida per al programa). S'haurà de cridar a *llegir_guarnició* de la classe *IO*, o a *modificar* (d'aquesta classe) per donar-li un valor coherent.

```
funció nova_guarnició () retorna g: Guarnició;
{ Pre: - }

  g.númSoldats := 0;

  retorna g;

{ Post: g és una guarnició sense soldats }
ffunció;
```

6.2.2 Implementació de bàndol

Volem saber el número de bàndol del soldat de la sala *n*, però per fer-ho hem de buscar aquesta sala a la llista. Si no la trobem o si el bàndol és 0, és que la sala està buida (i es retorna 0).

- **Inici**: el primer soldat que volem mirar és el de la posició 0 de la matriu, si existeix. En principi no l'hem trobat.
- **Invariant**: hem mirat els soldats de 0 a *i*-1, $b \neq 0 \longrightarrow$ *soldats[i]* és de la sala *n* i té bàndol *b*, *i* < *número_de_soldats_a_g*
- **Condició d'acabament**: podem trobar un valor per *b* (hem trobat al mosqueter o guàrdia de la sala *n*), podem trobar *b*=0 (l'usuari ha introduït un soldat de tipus 0), o podem acabar amb *i*=*número_de_soldats_a_g* i *b*=0 si no sortia a la llista. El que volem retornar és *b*.
- **Cos del bucle**: si és aquesta la posició on estan les dades del soldat de la sala *n*, agafa el valor de bàndol. Incrementa *i*.
- **Funció de cota**: *número_de_soldats_a_g – i*

```
funció bàndol (g: Guarnició, n: enter) retorna b: enter;
{ Pre: n<nombre_de_soldats_a_g }

  var i: enter; fvar

  i := 0;
  b := 0; // En principi no l'ha trobat a la llista

  { I: b != 0 ==> s'ha trobat el soldat de la sala n,
                  i b conté el bàndol a què pertany,
       b = 0 ==> als elements [0..i-1] de la matriu no hi és }
  mentre b=0 i i < g.númSoldats fer
    si g.soldats[i].salaInicial = n --> b := g.soldats[i].bàndol;
    fsi;
```

```
    i := i + 1;
  fmentre;

  retorna b;

{ Post: b conté el número de bàndol a què pertany el soldat de la sala n;
        si no n'hi ha cap, b=0 }
ffunció;
```

6.2.3 Implementació de qualitat

Aquesta funció resol un problema similar a l'anterior, però ara només acceptarem donar la qualitat d'un soldat que estigui a una sala no buida, ja que els soldats de les sales buides ni tan sols existeixen.

Per tant, podem fer una cerca amb garantia d'èxit: saltem uns quants elements fins trobar el bo.

- **Inici**: comencem mirant el primer element
- **Invariant**: hem mirat de 0 a *i*-1 i no l'hem trobat, *i<número_de_soldats_a_g* sempre
- **Condició d'acabament**: sabem que aquesta sala té un soldat, per tant surt a la llista i en algun moment trobarem la sala inicial *n*. Ja estem situats a la posició bona, ara només cal retornar la qualitat del soldat.
- **Cos del bucle**: simplement avançar; el bucle és només per posicionar-se en el registre buscat.
- **Funció de cota**: *número_de_soldats_a_g - i*

```
funció qualitat (g: Guarnició, n: enter) retorna q: enter;
{ Pre: n<nombre_de_soldats_a_g, la sala inicial n no està buida }

  var i: enter; fvar

  i := 0;

  { I: el soldat de la sala n no es troba a soldats[0..i-1] }
  mentre g.soldats[i].salaInicial != n fer
    i := i + 1;
  fmentre;

  retorna g.soldats[i].qualitat;

{ Post: q conté la qualitat del soldat de la sala n }
ffunció;
```

6.2.4 Implementació de quants

Retorna quants elements omplerts té el vector. Pot estar entre 1 i *N*.

```
funció quants (g: Guarnició) retorna n: enter;
{ Pre: - }

  retorna g.númSoldats;
```

```
{ Post: n és el número de soldats que hi ha a la guarnició g }
ffunció;
```

6.2.5 Implementació de modificar

Aquesta és la segona rutina important del programa: li passem dues guarnicions del mateix soterrani, i hem de comprovar si són compatibles (han de tenir el mateix número de soldats i a les mateixes sales). Si ho són, hem de tornar una guarnició nova on s'agafi el millor (en qualitat) de cada soldat inicial.

Com a paràmetres de sortida tinc la nova guarnició i una variable que diu si són compatibles o no (serà la que després es mostrarà per pantalla). Ho podia haver fet tornant només una tupla amb l'enter i la guarnició, però llavors hauria de declarar la tupla a molts llocs i crec que no val la pena només per relacionar aquests dos objectes.

No he fet cap funció per comprovar la compatibilitat, ja que ho he vist més eficient fet en la mateixa rutina de mescla de guarnicions. Comença a mesclarles, i si en algun moment es troba un signe d'incompatibilitat, s'atura i no modifica la *g*. Les dues guarnicions, *g1* i *g2*, es van mesclant guardant el resultat en *g2* (així s'evita perdre *g* si hi ha errors). En cas de que tot hagi funcionat bé, s'ha de copiar *g2* a *g*, perquè és *g* el paràmetre de sortida. Crec que és més eficient això que passar els objectes a una altra funció.

Sobre el bucle:

- **Inici**: si, d'entrada, les dues guarnicions no tenen el mateix número de soldats, ja sabem que són incompatibles. Si no ho són haurem de recórrer el vector de 0 fins a *número_de_soldats_a_g* – 1 (o *número_de_soldats_a_g2* – 1), per tant inicialitzem *i* a 0
- **Invariant**: *compat* = 0 ⟶ s'ha detectat una incompatibilitat; *compat* = 1 ⟶ s'han mirat els soldats de [0..*i*-1], i de moment són compatibles, i *g2* conté la fusió de les guarnicions en [0..*i*-1]
- **Condició d'acabament**: si *compat* es posa a fals (0), sortim del bucle i després no hem de fer res més ja que no s'ha modificat *g*. Si *compat* = 1 tota l'estona, segur que sortim perquè ja s'han recorregut tots els registres, en aquest cas *g2* ja té allò que volem retornar, i ho hem de copiar a *g*.
- **Cos del bucle**: les guarnicions deixen de ser compatibles si, per a una mateixa sala *n*, a *g1* està buida però a *g2* no, o viceversa (això vol dir que els soldats, encara ser els mateixos en nombre, no estan a les mateixes sales inicials). Sinó, s'ha d'agafar el millor, que és qui tingui més qualitat o els mosqueters en cas d'empat. Es modifica un registre de *g2* amb una funció externa més simple.
- **Funció de cota**: *número_de_soldats_a_g - i*

```
acció modificar (e/s g: Guarnició, e g1: Guarnició, e g2: Guarnició, s compat:
enter);
{ Pre: g1 i g2 són guarnicions vàlides }

  // Anirem modificant g2, i si cal, després es copia a g

  var
    i: enter;
```

```
    b1, b2, q1, q2: enter;
  fvar


  si g1.númSoldats = g2.númSoldats --> compat := 1;
  [] g1.númSoldats != g2.númSoldats --> compat := 0;
  fsi;

  i := 0;

  { I: compat = 1 ==> s'han mirat els soldats de [0..i-1],
       i de moment són compatibles,
       g2 conté la fusió de les guarnicions en [0..i-1] }
  mentre compat = 1 i i < g1.númSoldat fer

    b1 := bàndol(g1,i);
    b2 := bàndol(g2,i);

    si (b1=0) xor (b2=0) --> compat := 0; // una sala buida, l'altra no
    sinó --> // agafar el millor

      q1 := qualitat(g1,i);
      q2 := qualitat(g2,i);

      // Si g1 guanya a g2, modifica g2
      si q1 > q2 o (q1 = q2 i b1 = 1) --> modifica_un(g2, i, b1, q1);
      fsi;

    fsi;

    i := i + 1;

  fmentre;

  si compat = 1 --> g := g2; // copia g2 a g
  [] compat = 0 --> // g no s'ha modificat
  fsi;

{ Post: compat=1 => g té la unió de g1 amb g2,
        agafant el millor soldat de cada sala;
        compat=0 => el valor de g no ha canviat, g1 i g2 són incompatibles }
facció;
```

6.2.6 Implementació de modifica_un

L'algorisme és semblant a *qualitat*, però ara no busquem per llegir *bàndol* o *qualitat* sinó per escriure'ls. Ccerca amb garantia d'èxit: saltem uns quants elements fins trobar el bo.

- **Inici**: comencem mirant el primer element
- **Invariant**: hem mirat de 0 a *i*-1 i no l'hem trobat, *i*<*número_de_soldats_a_g* sempre
- **Condició d'acabament**: sabem que aquesta sala té un soldat, per tant surt a la llista i en algun moment trobarem la sala inicial *n*. Ja estem situats a la posició bona, i queda modificar els valors demanats.
- **Cos del bucle**: simplement avançar; el bucle és només per posicionar-se en el registre

buscat.

- **Funció de cota**: *número_de_soldats_a_g - i*

```
acció modifica_un (e/s g: Guarnició, e n: enter, e bàndol: enter, e qualitat:
enter);
{ Pre: n<nombre_de_soldats_a_g, la sala inicial n no està buida }

  var i: enter; fvar

  i := 0;

  { I: el soldat de la sala n no es troba a soldats[0..i-1] }
  mentre g.soldats[i].salaInicial != n fer
    i := i + 1;
  fmentre;

  g.soldats[i].bàndol := bàndol;
  g.soldats[i].qualitat := qualitat;

{ Post: el soldat inicial número n s'ha modificat amb els valors demanats }
facció;
```

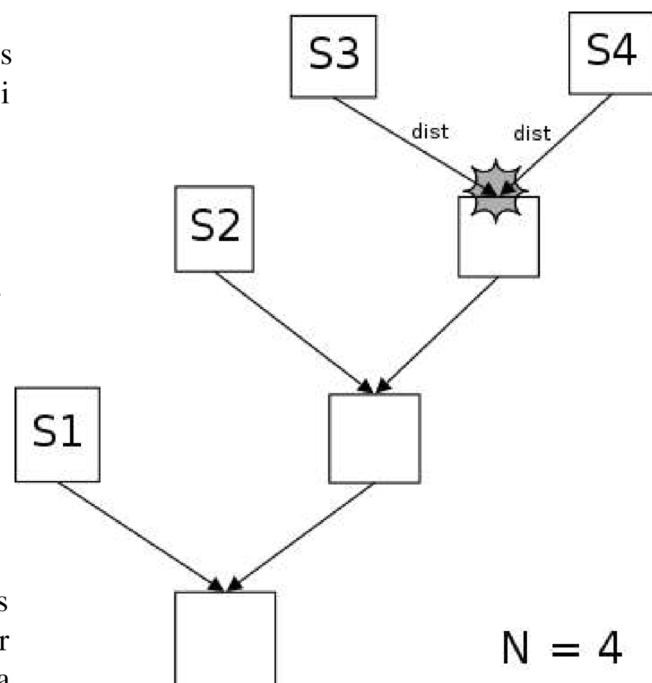
6.3 Implementació de Onades

Una onada és cada una de les tropes que aconseguiran arribar a la sortida. Hem de guardar informació sobre a quina distància estan de la sortida; això està al mòdul *Tropa*.

El més interessant d'aquest mòdul és que el vector de tropes està ordenat per la distància a la sortida de les tropes que hi ha dins.

Suposem que hi ha *N* soldats a les *N* sales inicials. En un principi sembla que el vector hagi de ser de *N* posicions ja que si cada soldat sortís pel seu compte, sense lluitar, es poden provocar fins a *N* onades diferents.

Però encara es pot millorar: el pitjor cas que he trobat per provocar el màxim d'onades és el del dibuix, perquè implica el número mínim de lluites (una) si suposem passadissos de la mateixa longitud.



De totes maneres, SEMPRE es produirà una lluita a dues sales inicials (*S3* i *S4* a l'exemple); és inevitable ja que la longitud dels passadissos recorreguts és la mateixa. Una de les tropes desapareixerà (fusionada o derrotada). Per tant, mai es produiran *N* onades, sinó *N*-1 (com a màxim).

Ens estalviem un objecte *Tropa* per crida recursiva... ja va bé, la memòria és cara aquest estiu. És important optimitzar aquest mòdul perquè és amb el que treballa la funció recursiva

simular_sortida.

Hi ha un detall a tenir en compte: el cas *N*=1. Si només hi ha una sala, aquesta és inicial (també és la sortida), i poden passar dues coses: que hi hagi un soldat o no. Si està buida, es produiran 0 onades, però si hi ha un soldat, ja hi ha una onada (individual i amb distància 0, però compta com a onada ja que és una tropa que aconsegueix sortir del soterrani). Si faig el vector de *N*-1 element i *N*=1, un vector de 0 elements no serveix per res. Per tant, necessito aquest número de posicions:

S1

N = 1

N = 1 --> 1 pos.	*N* = 2 --> 1 pos.	*N* = 3 --> 2 pos.	*N* = 4 --> 3 pos.	etc.

La fórmula que dóna això és *(N-1) mod N*, i com que *N* és constant, aquesta expressió també ho és i la podem utilitzar aquí com a tamany del vector (he definit una constant per què quedi més clar).

```
Mòdul Onades;


Implementació:

Sobre Tropa;

constant N = 20 // o el valor apropiat
constant MAX_ONADES = ( (N-1) mod N )

tipus Onades = tupla
                 tropes: vector [0..MAX_ONADES-1] de Tropa;
                 númOnades: enter;
               ftupla;


Operacions:
```

6.3.1 Implementació de nova_onades

Retorna una llista buida, amb 0 onades.

```
funció nova_onades () retorna o: Onades;
{ Pre: - }

  o.númOnades := 0;

  retorna o;

{ Post: o és una llista d'onades buida }
ffunció;
```

6.3.2 Implementació de afegir_onada

S'afegeix una nova onada a partir de la tropa que s'ha passat. A aquesta acció mai se la crida quan el vector ja està ple; abans de que passi això s'acaben de processar tots els soldats.

```
acció afegir_onada (e/s o: Onades, e t: Tropa);
{ Pre: a o encara hi cap una onada més }

  o.tropes[númOnades] := t;
  o.númOnades := o.númOnades + 1;

{ Post: o conté un registre més amb la tropa t }
facció;
```

6.3.3 Implementació de afegir_onada_individual

Com l'acció anterior, però aquest és un cas particular en què el número de soldats és 1 i la distància recorreguda (pel soldat) és 0.

```
acció afegir_onada_individual (e/s o: Onades, e b: enter, e q: qualitat);
{ Pre: a o encara hi cap una onada més }

  var temp: Tropa; fvar

  temp := nova_tropa(b, q);
  o.tropes[númOnades] := temp;
  o.númOnades := o.númOnades + 1;

{ Post: o conté un registre més amb la onada d'un soldat de bàndol b i qualitat
q }
facció;
```

6.3.4 Implementació de suma_dist_a_tots

Incrementa el valor de distància recorreguda a totes les tropes del vector.

- **Inici**: toca examinar el primer element
- **Invariant**: als *i*-1 primers elements ja se'ls ha sumat la distància, *i*<*número_onades_a_o*
- **Condició d'acabament**: quan s'arriba a l'última onada s'acaba
- **Cos del bucle**: sumem la distància a una tropa individual
- **Funció de cota**: *número_onades_a_o - i*

```
acció suma_dist_a_tots (e/s o: Onades, d: enter);
{ Pre: - }

  var i: enter; fvar

  i := 0;

  { I: ja s'ha sumat la distància als i-1 primers elements }
  mentre i < o.númOnades fer
    o.tropes[i] := suma_dist(o.tropes[i], d);
```

```
    i := i + 1;
  fmentre;

{ Post: totes les tropes de o estan d passos més lluny de la sortida }
facció;
```

6.3.5 Implementació de quantes

Retorna el número d'elements del vector usats.

```
funció quantes (o: Onades) retorna n: enter;
{ Pre: - }

  retorna o.númOnades;

{ Post: n és el nombre de tropes de la llista o }
ffunció;
```

6.3.6 Implementació de tropa

Retorna l'objecte *Tropa* que hi ha a la posició *n* del vector (no a la sala *n*, aquelles funcions eren de *Guarnició*).

```
funció tropa (o: Onades, n: enter) retorna t: Tropa;
{ Pre: n<nombre_de_tropes_a_o }

  retorna o.tropes[n];

{ Post: t és la tropa que hi ha a la posició n de la llista de tropes amb èxit }
ffunció;
```

6.3.7 Implementació de distància

Retorna la distància a la que la tropa número *n* es troba de la sortida de la sala actual (o de la sortida del laberint si la sala actual és l'arrel)

```
funció distància (o: Onades, n: enter) retorna d: enter;
{ Pre: n<nombre_de_tropes_a_o }

  retorna distància(o.tropes[n]);

{ Post: d és la distància a la que la tropa número n es troba de la sortida }
ffunció;
```

6.3.8 Implementació de fusiona

Aquesta és una funció de fusió de llistes ordenades una mica adaptada al problema. Tenim *o1* i *o2*, que són les llistes d'onades que han sortit de processar els fills de la sala actual. Ambdues estan ordenades per distància, ja que o bé consten d'un sol element o bé han estat generades per aquesta mateixa funció.

L'objectiu és generar una altra llista, *o*, que contingui tots els elements de *o1* junt amb els de *o2* ordenats per distància, però no ha de permetre que dues tropes tinguin la mateixa distància perquè en aquest cas es produeix una lluita i només queda una (tant si s'ajunten com si una guanya l'altra).

No es donarà el cas de què la tropa resultant d'una lluita entri en conflicte amb altra tropa, perquè si hi existís aquesta tropa ja l'hauríem descobert abans i l'hauríem fet lluitar.

A la funció utilitzo dos punters, *i* i *j*, que avancen per *o1* i *o2* amb molta cura, ja que sempre han d'afegir l'element amb distància menor (o, en cas de lluita, el resultat). Quan una llista s'acaba, l'altra s'acaba de copiar per complet.

Per al primer bucle:

- **Inici**: *i* i *j* es posen a 0 perquè és l'element que toca mirar
- **Invariant**: *o1*[0..*i*-1] i *o2*[0..*j*-1] s'han mesclat en *o*, processant lluites on calia, i *o* continua estant ordenada per distància
- **Condició d'acabament**: a cada iteració incrementem un o els dos comptadors *i* i *j*, així que arribarem a *i=quantes(o1)* o *j=quantes(o2)*, i llavors acabem perquè s'ha acabat una llista, i l'altra contindrà valors de distància més grans (en ordre, és clar). Queda acabar-les de copiar.
- **Cos del bucle**: afegim a la llista *o* el menor de *o1* i *o2*, segons toqui, i s'incrementa aquest mateix comptador ja que el següent element també pot ser menor que el de l'altra llista. En cas de ser iguals, no afegim les dues, ni només una, sinó que calculem la lluita de les tropes (dóna una altra tropa; una només) i l'afegim a *o*. Incrementem els dos comptadors perquè acabem d'afegir una dada que sortia a tots dos.
- **Funció de cota**: *quantes(o1) + quantes(o2) – (i+j)*

```
funció fusiona (o1: Onades, o2: Onades) retorna o: Onades;
{ Pre: o1 i o2 ordenades per distància }

  var i, j: enter; fvar

  i := 0; j := 0;

  { I: o1[0..i-1] i o2[0..j-1] s'han mesclat en o, processant lluites on calia,
       o continua estant ordenada per distància }
  mentre i < quantes(o1) i j < quantes(o2) fer

    si distància(o1, i) < distància(o2, j) -->
      afegir_onada(o, tropa(o1, i));
      i := i + 1;
    [] distància(o1, i) > distància(o2, j) -->
      afegir_onada(o, tropa(o2, j));
      j := j + 1;
    [] distància(o1, i) = distància(o2, j) -->
```

```
      afegir_onada(o, lluita( tropa(o1, i), tropa(o2, j) ));
      i := i + 1;
      j := j + 1;

    fsi;

  fmentre;
```

Ara una de les llistes s'ha acabat, però no sabem quina (a vegades serà *o1*, altres *o2*). A l'altra potser hi queden més valors, tots amb distància més gran que l'últim que acabo de ficar a *o*. Amb aquests dos bucles es termina de copiar el vector que falta. Només s'executarà un dels bucles, o potser cap si per casualitat les dues llistes *o1* i *o2* s'han acabat al mateix temps.

Per aquest segon bucle:

- **Inici**: *o1* o *o2* s'ha acabat de processar
- **Invariant**: els elements que queden a *o1* tenen una distància major que qualsevol dels que hi ha a *o*, *i<=número_de_tropes_a_o1*
- **Condició d'acabament**: sempre incrementem el comptador *i*, així que s'arribarà a *i = número_de_tropes_a_o1*
- **Cos del bucle**: afegim aquesta onada de la tropa *o1* (estem mantenint l'ordre) i s'incrementa *i*
- **Funció de cota**: *número_de_tropes_a_o1 – i*

```
  { I: o1 o o2 s'ha acabat de processar, els elements que queden a o1 tenen una
distància major que qualsevol dels que hi ha a o }
  mentre i < quantes(o1) fer
    afegir_onada(o, tropa(o1, i));
    i := i + 1;
  fmentre;
```

El tercer bucle és similar, però per copiar els elements que quedin a *o2*.

- **Inici**: *o1* s'ha acabat de processar sencera
- **Invariant**: els elements que queden a *o2* tenen una distància major que qualsevol dels que hi ha a *o*, *j<=número_de_tropes_a_o2*
- **Condició d'acabament**: sempre incrementem el comptador *j*, així que s'arribarà a *j = número_de_tropes_a_o2*
- **Cos del bucle**: afegim aquesta onada de la tropa *o2* (estem mantenint l'ordre) i s'incrementa *j*
- **Funció de cota**: *número_de_tropes_a_o2 – j*

```
  { I: o1 s'ha processat sencera, els elements que queden a o2 tenen una
distància major que qualsevol dels que hi ha a o }
  mentre j < quantes(o2) fer
    afegir_onada(o, tropa(o2, j));
    j := j + 1;
  fmentre;

  retorna o;

{ Post: o té les onades de o1 i de o2, i ha processat les lluites on les
distàncies de les tropes coïncidien, i o està ordenada per distància }
ffunció;
```

Amb aquest algorisme s'ha mantingut l'ordre de les onades; així ja està preparada la llista per mostrar-se per pantalla per ordre de sortida de les tropes.

6.4 Implementació de Tropa

Aquest mòdul només l'utilitza *Onades* al seu vector de tropes que arriben a la sortida. A aquesta classe es guarden las característiques importants de cada grup de soldats (les que infueixen en el càlcul de la sol·lució); la més important és el camp *distància*, que guarda la suma de les longituds dels passadissos que arriben fins a la sala que estem processant. Quan s'acabi el programa, sabrem la distància que separa cada tropa de la sortida.

```
Mòdul Tropa;


Implementació:

tipus Tropa = tupla
                bàndol: enter;
                númSoldats: enter;
                qualitat: enter;    // la suma de qualitats
                distància: enter;
              ftupla;


Operacions:
```

6.4.1 Implementació de nova_tropa

Retorna una tropa d'un soldat que es troba una sala inicial. El bàndol i la qualitat es passen, però ja sabem el número de soldats de la tropa (1) i que ha recorregut distància 0 (ja que encara no s'ha mogut de la sala inicial).

```
funció nova_tropa (bàndol: enter, qualitat: enter) retorna t: Tropa;
{ Pre: - }

  t.bàndol := bàndol;
  t.númSoldats := 1;
  t.qualitat := qualitat;
  t.distància := 0;

  retorna t;

{ Post: t és la tropa composta pel soldat de bàndol i qualitat seleccionats }
ffunció;
```

6.4.2 Implementació de lluita

Aquesta és una funció que ens retorna la tropa resultant de la trobada d'altres dos tropes. Si les tropes estan lluitant és perquè s'han trobat, o sigui, que estaven a la mateixa distància de la sala

actual.

Distingeixo dos casos diferents:

- **Lluita constructiva**: quan dos tropes amigues es troben, no es barallen, sinó que s'uneixen i continuen juntes. La tropa resultant té el mateix bàndol i distància i la suma del número de soldats i qualitats.
- **Lluita destructiva**: si no són del mateix bàndol, lluiten, i una de les tropes perd tots els soldats mentre que l'altra pot continuar sense perdre cap soldat. Ha de guanyar la tropa amb més qualitat, o els mosqueters en cas d'empat.

Per donar avantatge als mosqueters en cas d'empat, el que faig és tornar a desequilibrar les qualitats a favor dels mosqueters; llavors sempre hi ha una tropa millor que l'altra, i la comparació és molt senzilla.

En notació algorísmica, si guanya *t1* llavors *t := t1* i si guanya *t2* llavors *t := t2*, però he preferit deixar escrites les assignacions membre per membre perquè s'entengui millor.

```
funció lluita (t1: Tropa, t2: Tropa) retorna t: Tropa;
{ Pre: les dues tropes estaven a la mateixa distància }

  si t1.bàndol = t2.bàndol --> // Trobada constructiva, s'uneixen
                               t.bàndol := t1.bàndol;
                               t.númSoldats := t1.númSoldats + t2.númSoldats;
                               t.qualitat := t1.qualitat + t2.qualitat;
                               t.distància := t1.distància;

  [] t1.bàndol != t2.bàndol --> // Trobada destructiva, lluiten
    // Guanyen els de més qualitat, o els mosqueters en cas d'empat

    si t1.qualitat = t2.qualitat --> // LLavors els mosqueters tenen avantatge
      si t1.bàndol = 1 --> t1.qualitat := t1.qualitat + 1;
      [] t1.bàndol != 1 --> t2.qualitat := t2.qualitat + 1;
    fsi

    si t1.qualitat > t2.qualitat --> // Guanya t1
                                     t.bàndol := t1.bàndol;
                                     t.númSoldats := t1.númSoldats;
                                     t.qualitat := t1.qualitat;

    [] t1.qualitat < t2.qualitat --> // Guanya t2
                                     t.bàndol := t2.bàndol;
                                     t.númSoldats := t2.númSoldats;
                                     t.qualitat := t2.qualitat;
    fsi;

    t.distància := t1.distància;

  fsi;

  retorna t;

{ Post: t és el resultat de la trobada entre t1 i t2
        (constructiva si són amics o destructiva si enemics) }
ffunció;
```

6.4.3 Implementació de suma_dist

Retorna una tropa igual a la passada però amb la distància incrementada.

```
funció suma_dist (t: Tropa, d: enter) retorna t2: Tropa;
{ Pre: la tropa t té soldats }

  t2.bàndol := t.bàndol;
  t2.númSoldats := t.númSoldats;
  t2.qualitat := t.qualitat;
  t2.distància := t.distància + d;

  retorna t2;

{ Post: t2 él la tropa resultant d'incrementar la distància de la tropa }
ffunció;
```

6.4.4 Implementació de distància

Retorna el valor del membre privat *distància*.

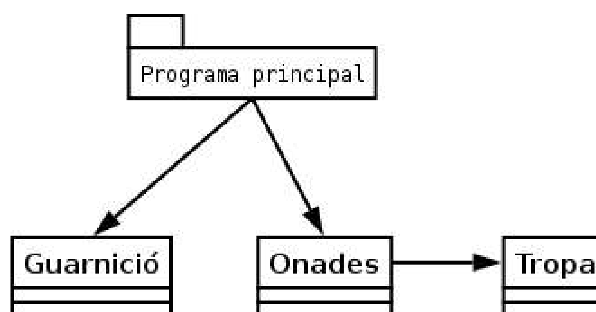
```
funció distància (t: Tropa) retorna d: enter;
{ Pre: la tropa t té soldats }

  retorna t.distància;

{ Post: d és la distància a la que la tropa t es troba de la sala actual }
ffunció;
```

7. Possibles millores

7.1 Classe Soterrani no necessària

La classe *Soterrani* no és del tot necessària, ja que només té un arbre d'enters, i és més fàcil accedir a un 'arbre d'enters' que a un 'arbre d'enters que està a dins d'una classe'. La he deixat per claredat -ja que l'estructura d'un soterrani no és trivial- i per no mesclar sales/passadissos amb els altres mòduls. D'altre mode haurien anat en un mòdul funcional (o a *Guarnició*, *Onades* o al *Programa Principal*, encara que és poc elegant).

Si no s'hagués utilitzat *Soterrani*, l'esquema modular hauria quedat molt més senzill:



7.2 Variables temporals

A molts llocs s'han utilitzat variables temporals només per poder posar cada instrucció en una línia separada, però moltes vegades les podem eliminar sense perjudicar l'eficiència del programa (sí la legibilitat). Per exemple, a *afegir_onada_individual*:

```
  var temp: Tropa; fvar

  temp := nova_tropa(b, q);
  o.tropes[númOnades] := temp;
  o.númOnades := o.númOnades + 1;
```

es pot escriure com:

```
  o.tropes[númOnades] := nova_tropa(b, q);
  o.númOnades := o.númOnades + 1;
```

però ja s'ha fet una crida a una funció i una assignació a la mateixa línia, i això costa més d'entendre.

Altres exemples: *b1*, *b2*, *q1*, *q2* a *modificar* (*Guarnició*), *valor* a *simular_sortida_rec* (*Soterrani*).

7.3 Constructores

Les funcions *nou_soterrani* i *nova_guarnició* no són del tot necessàries, ja que el codi següent:

```
s := nou_soterrani();
g := nova_guarnició();

llegir_soterrani(s);
llegir_guarnició(g);
```

també es podria escriure com:

```
llegir_soterrani(s);
llegir_guarnició(g);
```

i ja estaríem donant valors a *s* i *g*.

S'han deixat aquestes funcions per evitar haver de treballar amb objectes no inicialitzats. De la primera forma, si es crida a *nova_guarnició* ja sabem que “*g* té 0 soldats”, què és molt millor que poder dir només que “*g* no té definit cap valor per al número de soldats”.

7.4 Funcions de còpia

A *modificar* de la classe *Guarnició* trobem:

```
  si compat = 1 --> g := g2; // copia g2 a g
```

on *g* i *g2* són objectes de tipus *Guarnició*.

Quan això es tradueixi a llenguatges com JAVA, s'hauran d'afegir operacions de còpia que asegurin que els elements interns també es copiïn bé. Un exemple en pseudocodi per a l'acció de còpia de guarnicions és:

```
acció copiar (s g: Guarnició, e origen: Guarnició);
{ Pre: - }
```

```
  var i: enter; fvar

  g.númSoldats := origen.númSoldats;

  i := 0;
  { I: s'han copiat els soldats de [0..i-1] }
  mentre i < g.númSoldats fer
    g.soldats[i].salaInicial := origen.soldats[i].salaInicial;
    g.soldats[i].bàndol := origen.soldats[i].bàndol;
    g.soldats[i].qualitat := origen.soldats[i].qualitat;
  fmentre;

{ Post: g és una còpia de la guarnició origen }
facció;
```

ERRORES DETECTADOS DESPUÉS DE LA ENTREGA :-(

- Si la calidad es igual en la lucha, le suma 1 a los mosqueteros. No debería.

- En lo de modificar guarniciones hay dos problemas:

```
while (compat==1 && i<g1.numSoldats) {
   b1=g1.bandol(i); b2=g2.bandol(i);

   if ( (b1==0) ^ (b2==0) ) compat=0;
   else {          <<<<<<<<<<<============== ¡¡¡¡¡ ERROR !!!!! SI UNA SALA ESTÁ
VACÍA EN AMBAS GUARNICIONES, NO TENGO QUE COGER EL SOLDADO DE MEJOR CALIDAD.
     q1=g1.qualitat(i); q2=g2.qualitat(i);
     if(q1>q2 || (q1==q2 && b1==1) ) g2.modifica_un(i,b1,q1);
```

CAMBIA A:

```
   if ( (b1==0) ^ (b2==0) ) compat=0;
   else if (b1!=0 && b2!=0) {
     q1=g1.qualitat(i); q2=g2.qualitat(i);
```

En el mismo sitio:

- Necesito una función en Guarnición como llegeix_sala:

```
public int llegeix_sala (int n) throws Exception
{        return soldats[n].salaInicial;  }
```

No he de hacer referencia al *bando del soldado número 0*, sino al *bando del soldado de la sala correspondiente a la posición 0*, o sea, llegeix_sala(0)

TOTAL, QUE QUEDA ASÍ:

```
while (compat==1 && i<g1.numSoldats) {

int sala=g1.llegeix_sala(i);
b1=g1.bandol(sala); b2=g2.bandol(sala);

  if ( (b1==0) ^ (b2==0) ) compat=0;
  else if (b1!=0 && b2!=0) {
    q1=g1.qualitat(sala); q2=g2.qualitat(sala);
    if(q1>q2 || (q1==q2 && b1==1) ) g2.modifica_un(sala,b1,q1);
  }
  i++;
}
```

- Por lo demás, funciona todo bien.