DISSENY MODULAR

Part I

Albert Rubio Borja Valles

Última versió: setembre de 2009

1 Mòduls

L'objectiu del curs és produir programes fiables i fàcils d'entendre, modificar, mantenir i reusar. Totes aquestes propietats afecten molt significativament als costos associats al disseny i a la vida del programa. Per programes molt petits (de poques línies) això es pot aconseguir mitjançant l'ús d'una o unes poques funcions o accions de poques línies agrupades en una mateixa unitat monolítica. Ara bé, si el tamany del programa creix considerablement aquesta estructura monolítica ens pot fer perdre fàcilment totes les propietats que volem assolir. Per exemple, modificar o mantenir fiablement una unitat monolítica de milers de línies pot resultar impossible. A més, pensar en reusar parts d'aquesta unitat monolítica no té ni sentit. Finalment, la fiabilitat i la legibilitat d'un programa d'aquestes dimensions difícilment podrà ser garantida.

La forma natural d'evitar tots aquest problemes és dividir el nostre programa en *mòduls* que junts realitzin la mateixa tasca que el programa monolític.

És obvi que trencar simplement el nostre programa en troços qualssevol de tamany petit, no és una política molt raonable per assolir els nostres objectius. És molt important realitzar una bona descomposició a fi de garantir al màxim les propietats del nostre producte final. En aquest sentit s'han de tenir en compte les següents qüestions:

- Els mòduls han de ser independents entre si, de manera que es puguin fer canvis en un mòdul sense que això obligui a modificar els altres mòduls. Aquesta qüestió és molt important per a la modificabilitat i el manteniment, així com en el cas de que hi hagi molta gent involucrada en el disseny del programa.

- Els mòduls han de tenir una entitat pròpia, com petits programes que realitzen unes determinades tasques, i han d'interactuar amb altres mòduls d'una forma simple i ben definida. Aquesta qüestió és especialment útil per aconseguir programes fàcils d'entendre, per potenciar la reusabilitat i per facilitar el treball en grup.

La forma d'aconseguir bones descomposicions modulars que considerarem en aquest curs està basada en l'*abstracció.* Quan fem una abstracció d'un problema, el que fem és oblidar-nos de certs detalls que ens permetin transformar el nostre problema en un de més simple.

Un cas ja conegut d'abstracció s'obté mitjançant l'ús de paràmetres en funcions i accions. L'*abstracció per parametrització* ens abstreu de les dades particulars substituint-les per paràmetres. Suposem que tenim un vector d'enters i volem saber quants cops apareix al vector l'enter que està en la posició `j` del mateix vector. En aquest cas, el veritable problema és veure quants cops apareix un determinat enter `x` en el vector, i el fet de que estigui en la posició `j` del vector és irrellevant. Per tant, ens podem abstreure del fet que l'element a considerar sigui del vector.

Un altre cas d'abstracció també emprat a les funcions i accions s'obté mitjançant l'especificació Pre/Post. Per exemple,

```
funcio pot(a,b : nat ) ret p : nat;
{Pre: a>0 }
{Post: p= a^b }
```

L'*abstracció per especificació* ens abstreu dels detalls d'implementació, i ens permet coneixer simplement *què* fa la funció o acció, sense necessitat de saber *cóm* ho fa. D'aquesta manera, els canvis en la implementació no han d'afectar l'ús de la funció o acció. En l'exemple, es veu que per

usar la funcio `pot` no ens cal saber cóm està dissenyada, només ens cal saber que han de satisfer les dades que li passem i que satisfaran els resultats quan la funció s'hagi executat.

Mitjançant l'abstracció per paràmetres i l'abstracció per especificació podem realitzar dos tipus de descomposició o abstracció.

Abstracció funcional. Aquest tipus d'abstracció és útil quan el problema es pot descomposar en unitats funcionals (també anomenades operacions) independents, descrites simplement amb els seus paràmetres i les seves especificacions (i deixant la seva implementació per a futurs refinaments). En aquest cas, l'abstracció s'assoleix creant i afegint noves operacions al nostre llenguatge de programació que ens permetin resoldre el nostre problema. Aquest tipus de descomposició ja apareix en problemes de tamany petit i ja s'ha usat en el curs anterior sense donar-li cap nom.

Abstracció de dades. En problemes de més entitat resulta normalment més útil pensar en crear i afegir nous tipus de dades al nostre llenguatge. Cóm podem descriure aquests nous tipus de manera que representin realment una ampliació del lleguatge? En qualsevol dels tipus de dades coneguts el que tenim es

- Un nom pel tipus, que l'identifica.
- Una sèrie d'operacions que ens permeten construir, modificar o consultar elements del tipus.

Per aconseguir que l'ús dels nous tipus sigui el mateix que amb els tipus existents hem d'assegurar que aquest ús és *independent* de les possibles implementacions particulars de les dades i les operacions que es puguin realitzar. Per exemple, fins ara hem realitzat una gran quantitat d'algorismes sobre els enters, sense necessitat de saber ni com es representa un enter dins la màquina ni com estan dissenyades les operacions de suma, resta, multiplicació, etc.

Finalment, s'ha de tenir en compte que l'abstracció de dades requereix també una abstracció funcional per identificar les operacions.

Així doncs en aquest curs considerarem dos tipus de mòduls:

- *Mòduls funcionals.* Són aquells obtinguts només per abstracció funcional i que contindran un conjunt d'operacions noves necessàries per resoldre algun problema o subproblema.
- *Mòduls de dades.* Són aquells obtinguts per abstracció de dades i que contindran un nou tipus i les seves operacions. Aquest serà el tipus de mòdul més usat en els nostres dissenys.

2 Especificació de mòduls

Considerem ara com especificar mòduls de dades (noteu que els mòduls funcionals en són un cas particular) de manera que garantim la independència de la implementació. Per exemple, suposem que per a gestionar les notes d'una assignatura volem crear un nou tipus de dades per enmagatzemar la informació dels estudiants. Especificariem aquest mòdul de dades de la següent manera:

```
Mòdul Estudiant;

Especificació

{ Tipus de mòdul: dades }

Tipus Estudiant;
```

```
{ Descripció general: Conte el DNI d'un estudiant, que és un enter positiu,
                          i pot tenir nota que és un real}

Operacions

funció crear_estudiant(dni: enter) ret est:Estudiant;
{Pre: dni >= 0 }

{Post: ens retorna a  est un estudiant amb DNI dni i sense nota}


funció afegir_nota(est:Estudiant; not:real) ret sest:Estudiant;
{Pre: est és un estudiant sense nota i not >= 0}

{Post: sest  és un estudiant amb el DNI de est i amb la nota not}


funció consultar_dni(est:Estudiant) ret dni: enter;
{Pre: --- }

{Post: dni  és el DNI de l'estudiant  est }


funció consultar_nota(est:Estudiant) ret not: real;
{Pre: est és un estudiant amb nota}

{Post: not  és la nota de l'estudiant  est}


acció modificar_nota(e/s est: Estudiant; ent not:real)
{Pre: est és un estudiant amb nota i  not >= 0}

{Post: la nota de  est  passa a ser  not}


funció te_nota(est:Estudiant) ret b: bool;
{Pre: --- }

{Post: b  ens diu si l'estudiant  est té nota o no}

FEspecificació
```

Comentaris:

1. encara que no s'afegeixi a les precondicions, s'assumeix que tots els paràmetres de tipus Estudiant contenen elements definits, és a dir, obtinguts amb operacions sobre el resultat de `crear_estudiant`.

2. la funció `afegir_nota` podria haver-se definit com una acció

   ```
   acció afegir_nota(e/s est:Estudiant; ent not:real);
   {Pre: est és un estudiant sense nota i  not >= 0}
   ```

```
{Post: modifica est posant-li com a nota  not}
```

En la versió que hem inclòs anteriorment en l'especificació, es considera que la funció crea un nou estudiant amb el mateix DNI però amb la nota definida. En canvi, en la versió que hem presentat ara, considerem que el resultat de l'acció és el mateix estudiant amb les seves dades modificades (amb la nota).

La conveniència d'una o altra versió és una decisió del dissenyador, i, fins i tot, podria ser raonable tenir les dues versions.

3. analogament (però al contrari) l'acció `modificar_nota` podria haver-se definit com una funció.

4. La raó per distingir `afegir_nota` de `modificar_nota`, és conceptual: encara que tenir només `modificar_nota` podria ser suficient (si li relaxem la precondició), és clar que ambdues operacions estan pensades per realitzar tasques diferents, i per tant considerar-les diferents aumenta tant la fiabilitat en l'ús del tipus com la legibilitat (i per tant la modificabilitat i la mantenibilitat).

5. la necessitat de definir `te_nota` apareix pel fet de que hi ha operacions que tenen com a requisit que l'estudiant tingui o no tingui nota.

6. Noteu que, per usar el mòdul Estudiant, no cal saber ni com s'ha implementat el tipus Estudiant ni les seves operacions. En particular, no ens cal saber com s'ha gestionat internament la questió de tenir nota o no (amb un booleà o usant negatius a la nota, ...).

Un altre exemple més abstracte és el de la definició del tipus pila, com a estructura que enmagatzema elements i que permet obtenir i eliminar només l'últim element afegit. Considerarem l'especificació de les piles d'enters.

```
Mòdul Pilaent;

Especificació

{ Tipus de mòdul: dades }

Tipus pilaent;

{ Descripció general: és una estructura que conté enters
      que permet consultar i eliminar nomes l'últim element afegit}

Operacions

funció pila_buida() ret p: pilaent;
{Pre: --- }

{Post: p és una pila sense elements (buida)}


funció empilar(x: enter; p: pilaent) ret q: pilaent;
{Pre: --- }

{Post: q  és una pila com p però amb x afegit com a últim element}
```

```
funció desempilar(p: pilaent) ret q: pilaent;
{Pre: p no és buida }

{Post: q  és una pila com p però sense l'últim element
                                        que se li havia afegit}

funció cim (p: pilaent) ret x : enter;
{Pre: p no és buida }

{Post: x  és l'últim element que s'ha afegit a p}


funció es_buida (p: pilaent) ret b: bool;
{Pre: --- }

{Post: b  ens diu si  p  és buida o no}

FEspecificació
```

En aquesta especificació s'apliquen també els comentaris fets en l'exemple anterior sobre la conveniència d'usar funcións o accions amb paràmetres d'entrada/sortida en les operacions `empilar` i `desempilar`.

2.1 Especificacions genèriques

En l'últim exemple hem introduït les piles d'enters. Ara bé, si ens fixem en l'especificació de totes les operacions, el fet de que la pila sigui d'enters o de caràcters és irrellevant. De fet, si fessim l'especificació de les piles de caràcters l'unic que hauriem de canviar (a part del nom del tipus) és el tipus enter per caràcter en les operacions `empilar` i `cim`. En general, podem veure que podem abstreure'ns del fet de si la pila conté enters, caràcters o booleans, i considerar que en general tenim elements de tipus `elem`, on `elem` funcióna per l'especificació com un paràmetre formal (en el mateix sentit que els paràmetres en les accions i funcións). Així, quan volguem definir una pila haurem de dir quin és el paràmetre real, per exemple `pila de nat`, o `pila de car`, etc.

Aquestes especificacions parametritzades, s'anomenen *genèriques* o *polimòrfiques.* Encara que en aquest curs no s'hauran de dissenyar mòduls de dades genèric, si que se'n fara ús. S'ha de tenir en compte que l'ús de tipus parametritzats no és nou d'aquest curs, ja que, per exemple, en el tipus `vector` (o `taula`) també tenim parametritzat el tipus d'elements que conté (i l'hem de particularitzar en cada ús).

2.1.1 Piles genèriques

Un cop parametritzada l'especificació de piles ens queda de la següent manera.

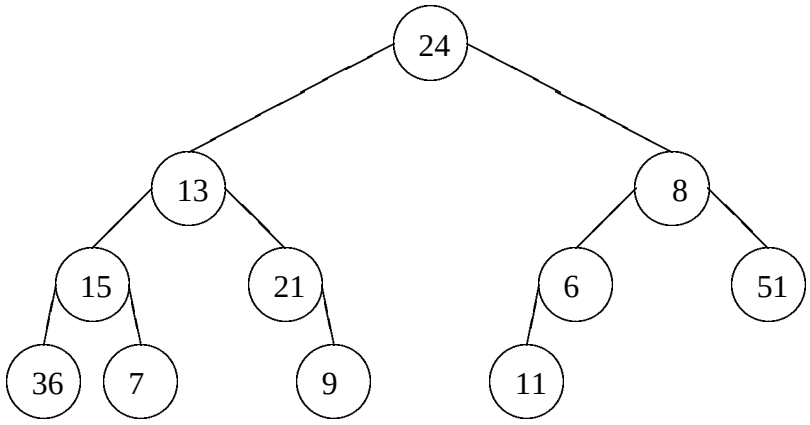
```
Mòdul Pila;

Especificació
```

```
{ Tipus de mòdul: dades }

Parametre formal elem;

Tipus pila;

{ Descripció general: és una estructura que conté elements de tipus elem
     i que permet consultar i eliminar nomes l'últim element afegit}

Operacions

funció pila_buida() ret p: pila;
{Pre: --- }

{Post: p  és una pila sense elements (buida)}


funció empilar(x: elem; p: pila) ret q: pila;
{Pre: --- }

{Post: q  és una pila com p però amb x afegit com a últim element}


funció desempilar(p: pila) ret q: pila;
{Pre: p no és buida }

{Post: q  és una pila com p però sense l'últim element que se li havia afegit}

funció cim (p: pila) ret x : elem;
{Pre: p no és buida }

{Post: x  és l'últim element que s'ha afegit a p}


funció es_buida (p: pila) ret b: bool;
{Pre: --- }

{Post: b  ens diu si  p  és buida o no}


FEspecificació
```

Noteu que gràcies a la genèricitat ara podem definir piles de qualsevol tipus de elements, en particula podem definir piles de vectors, piles de piles d'enters, etc.

2.1.2 Arbres binaris genèrics

Per ampliar el repertori de tipus de dades genèrics (que podrem usar per enmagatzemar les nostres dades i que podrem combinar amb tots els altres tipus coneguts) presentarem els *arbres binaris.* Gràficament, l'estructura que volem representar seria com un arbre invertit amb l'arrel a dalt i les fulles a baix. El següent exemple mostra un arbre de números naturals.



La propietat principal d'un arbre no buit és que podem visitar la seva *arrel* i els dos subarbres que hi tenim penjats i que anomenarem *fills*. Noteu que tots dos subarbres d'un arbre són també arbres i que tot arbre no buit té dos subarbres, encara que algun pot ser buit. En particular, les *fulles* són arbres no buits que tenen dos subarbres buits. Per altra banda, per afegir elements s'han d'ajuntar dos arbres amb una nova arrel. Aquesta operació l'anomenarem `plantar`. Per exemple, l'arbre que conté només el número `9` és el resultat de plantar `9` sobre dos arbres buits. Els elements d'un arbre els anomenarem *nodes*.

Finalment, noteu que, de la mateixa manera que amb les piles, el tractament que fem d'un arbre, és independent del tipus d'elements que contingui, i per tant podem presentar-lo en una especificació parametritzada.

```
Mòdul Arbre;

Especificació

{ Tipus de mòdul: dades }

Parametre formal elem;

Tipus arbre;

{ Descripció general: és una estructura que conté elements de tipus
     elem i que permet estructurar la informació de forma
     arborescent (amb dos i nomes dos descendents) i consultar
     nomes l'últim element afegit i un dels dos descendents }

Operacions

funció arbre_buit() ret a: arbre;
{Pre: --- }

{Post: a  és un arbre sense elements (buit)}


funció plantar(x: elem; a,b: arbre) ret c: arbre;
{Pre: --- }
```

```
{Post: c  és un arbre que té x com a últim element afegit,
       a  com a descendent (fill) esquerra i  b  com a fill dret}


funció fill_esq(a: arbre) ret b: arbre;
{Pre: a no és buit }

{Post: b  és l'arbre descendent (fill) de l'esquerra de  a }


funció fill_dre(a: arbre) ret b: arbre;
{Pre: a no és buit }

{Post: b  és l'arbre descendent (fill) de la dreta de  a }


funció arrel (a: arbre) ret x : elem;
{Pre: a no és buit }

{Post: x  és l'últim element que s'ha afegit a  a }


funció es_buit (a: arbre) ret b: bool;
{Pre: --- }

{Post: b  ens diu si  a  és buit o no}


FEspecificació
```

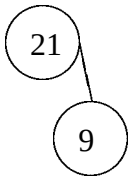
Per assignar a la variable b (declarada de tipus arbre d'enters) l'arbre que conté només el número 9 faríem

```
b := plantar( 9 , arbre_buit(), arbre_buit() );
```

A continuació podriem fer

```
c := plantar( 21 , arbre_buit(), b );
```

i aconseguiriem tenir a c l'arbre



Com hem dit tot arbre no buit té dos descendents. En l'exemple, el fill esquerra és un arbre buit (que gràficament no apareix) i el fill dret és l'arbre que conté el número 9 i té els dos descendent buits.

2.1.3 Cues genèriques

L'últim exemple de tipus genèric que introduirem són les *cues*. A una cua es poden emmagatzemar elements d'un en un i l'únic que es pot consultar o treure en un moment donat és el primer que va entrar dels que hi són en aquell moment. Com a conseqüencia d'això, un element que s'emmagatzema a una cua no es pot treure fins que s'hagin tret tots els que van entrar-hi abans que ell. Com podeu veure, un element d'aquest tipus representa fidelment una cua de la vida real, per exemple, la d'una finestreta o la d'una taquilla.

```
Mòdul Cua;

Especificació

{ Tipus de mòdul: dades }

Parametre formal elem;

Tipus cua;

{ Descripció general: és una estructura que conté elements de tipus elem
     i que permet consultar i eliminar nomes el primer element afegit}

Operacions

funció cua_buida() ret c: cua;
{Pre: --- }

{Post: c  és una cua sense elements (buida)}


funció demanar_torn(x: elem; c: cua) ret q: cua;
{Pre: --- }

{Post: q  és una cua com c però amb x afegit com a últim element}


funció avançar(c: cua) ret q: cua;
{Pre: c no és buida }

{Post: q  és una cua com c però sense el primer element
                                        que se li havia afegit}

funció primer (c: cua) ret x : elem;
{Pre: c no és buida }

{Post: x  és el primer element que s'ha afegit a c}


funció es_buida (c: cua) ret b: bool;
{Pre: --- }

{Post: b  ens diu si  c  és buida o no}
```

```
FEspecificació
```

Com al cas de les piles, les operacions `demanar_torn` i `avançar` poden ser declarades alternativament com accions.

3 Jerarquia de mòduls

Els nous mòduls funcionals o de dades es defineixen sobre tipus de dades ja existents, ja siguin predefinits o incorporats al llenguatge via mòduls de dades. Igualment, quan implementem les especificacions dels mòduls podem necessitar altres mòduls funcionals o de dades ja disponibles (o que suposem que existeixen i que posteriorment haurem de crear). Això ens defineix una jerarquia de mòduls que reflecteix les dependències entre els mòduls. En aquest curs considerarem un tipus de dependència molt bàsic.

S'ha de distingir entre les dependències visibles en l'especificació i les ocultes que apareixen en la implementació. Considerarem ara només les dependències d'especificació quan definim un nou tipus sobre un previament definit.

Suposeu que volem tenir operacions de lectura d'estudiants. Per això, considerem un nou tipus de dades "seqüència d'estudiants". A continuació teniu una possible especificació.

```
Mòdul Seq_Estudiant;

Especificació

Sobre Estudiant;
{ Aquest mòdul està definit sobre el mòdul Estudiant}

{ Tipus de mòdul: dades }

Tipus seq_estudiant;

{ Descripció general: és una estructura de lectura sequencial
                            que conté estudiants }


Operacions

acció obrir_seq_estudiant( e/s  s: seq_estudiant );
{ Pre: s  està tancada }
{ Post: s  està oberta i preparada per llegir el primer estudiant }

acció llegir_estudiant( e/s  s: seq_estudiant; sort e: Estudiant );
{ Pre: s  està oberta i queden estudiants per llegir }
{ Post: Ha llegit l'estudiant  e  i
         s  està preparada per llegir el seguent estudiant }

funció final_seq_estudiant( s: seq_estudiant ) ret b: bool;
```

```
{ Pre: s  està oberta }
{ Post: b = ‘‘queden estudiants per llegir’’ }

acció tancar_seq_estudiant( e/s  s: seq_estudiant );
{ Pre: s  està oberta }
{ Post: s  està tancada }

FEspecificació
```

Com hem pogut veure el mòdul `Seq_Estudiant` depèn del mòdul `Estudiant`.

3.1 Enriquiments

Els enriquiments serien mòduls funciónals que extenen un mòdul de dades o un tipus predefinit amb noves operacions. En aquest cas tenim també una dependència, però no definim cap tipus nou.

Per exemple suposeu que volem afegir a les piles de naturals una operació per calcular l'altura d'una pila i una operació d'igualtat a les piles. Fariem el següent

```
Mòdul opsPila;

Especificació

Sobre Pila;
{ Aquest mòdul està definit sobre el mòdul Pila}

{ Tipus de mòdul: funciónal }

Operacions

funció altura( p: Pila ) ret n: nat;
{ Pre: --- }

{ Post: n = ‘‘nombre d’elements que conté p’’ }

funció iguals( p, q: Pila ) ret b: bool;
{ Pre: --- }

{ Post: b = ‘‘p i q contenen els mateixos elements afegits
              en el mateix ordre’’ }


FEspecificació
```

El mòdul `opsPila` depèn del mòdul `Pila` encara que no defineix cap tipus nou.

4 Ús de mòduls

Per usar un mòdul només ha de caldre mirar la seva especificació. Suposeu que volem fer un programa que donat un vector d'estudiants ens diu el percentatge de presentats.

Especificació:

```
funció presentats (vest: vector [1..N] d'Estudiant) ret p: real;
{Pre: --- }

{Post: p = ''percentatge de presentats, és a dir,
             d'estudiants amb nota, de vest''}
```

Implementació:

Per aconseguir el percentatge de presentats necessitem saber el nombre de presentats, és a dir, el nombre d'estudiants amb nota.

L'estratègia per resoldre aquest problema és anar comptant el nombre d'estudiants amb nota a mida que recorrem el vector. Per tant, considerem que hem comptat els estudiants amb nota fins a un punt "i" (sense incloure'l) i avançem considerant el seguent (l'i-essim). Així l'invariant és

```
{I: i<= N, n = ''nombre d'estudiants amb nota en vest[1..i-1]''}
```

Anem a raonar les instruccions del bucle.

- *Inicialitzacions.* Incialment considerem que no hi ha cap estudiant del vector tractat, això ho representem posant un `1` a la `i`, la qual cosa ens obliga a posar un `0` a la `n`, ja que no hi ha cap estudiant fins a `i-1` amb nota.

- *Condició de sortida.* Com que volem tenir al final tot el vector tractat, sortirem del bucle quan `i=N+1`, per tant ens quedem mentre `i =< N` (noteu que hem demanat a l'invariant que `i =< N+1`).

- *Cos del bucle.* Com que hem d'avançar, el més natural és incrementar la `i` de manera que s'acosti al final del vector. Però abans d'incrementar la `i`, hem de satisfer l'invariant canviant `i` per `i+1` (aplicant l'axioma de l'assignació). Necessitem, per tant, que `n` contingi finalment el "nombre d'estudiants amb nota en `vest[1..i]`". Com que actualment `n` conté el "nombre d'estudiants amb nota en `vest[1..i-1]`", només ens cal preguntar si `vest[i]` té nota (l'accés és correcte ja que `1 =< i =< N`). En cas afirmatiu el nombre d'estudiants amb nota és `n+1`, que és el que hem d'assignar a `n`. En canvi, si no té nota el nombre d'estudiants amb nota segueix sent `n`, i per tant no cal fer res.

  Un cop que `n` conté el "nombre d'estudiants amb nota en `vest[1..i]`", ja podem incrementar la `i` assegurant que satisfem l'invariant. A més la condició del bucle ens assegura que `i+1 =< N+1`.

- *Acabament.* A cada volta decreix la distància entre `N+1` i la `i`.

- *Instruccions finals.* Quan sortim del bucle tenim a `n` el "nombre d'estudiants amb nota en `vest`", per tant només cal multiplicar-lo per `100` i dividir-lo per `N`, i obtindrem el percentatge.

El programa anotat queda de la següent manera:

```
funció presentats (vest: vector [1..N] d'Estudiant) ret p: real;
{Pre: --- }
var n, i: nat fvar;
```

```
i:= 1;
n:= 0;

{ 1 =< i <= N+1, n = ‘‘nombre d’estudiants amb nota en vest[1..i-1]’’}

Mentre i <= N fer

  { mirem si vest[i] té nota }
  Si te_nota(vest[i]) llavors n:= n+1 fSi;
  i:= i+1

fMentre;

{ n = ‘‘nombre d’estudiants amb nota de vest’’}

p:= n*100/N

{Post: p = ‘‘percentatge de presentats, és a dir,
              d’estudiants amb nota, de vest’’}
```

Un altre exemple és fer un programa que, donat un vector d'estudiants, el modifica arrodonint-ne les notes a la dècima més propera (es pot fer com a acció o com a funció).
Especificació:

```
acció arrodonir_notes (e/s vest: vector [1..N] d’Estudiant);
{Pre: --- }

{Post: vest té les notes dels estudiants arrodonides respecte
       al seu valor inicial}
```

Implementació:

Per simplificar el disseny hem suposat per abstracció funciónal, que tenim una funció que ens arrodoneix un real. Aquesta funció la dissenyarem al final, i d'aquesta manera hem separat el problema tècnic de com fer l'arrodoniment d'un real, del nostre problema general que és arrodonir les notes d'un vector d'estudiants.

Les justificacions són similars al programa anterior. Considerem com invariant

```
{ vest[1..i-1] té les notes dels estudiants arrodonides respecte
  al seu valor inicial, 1 =< i <= N+1}
```

Anem a justificar informalment el programa.

- *Inicialitzacions.* Incialment considerem que no hi ha cap estudiant del vector tractat, això ho representem posant un `1` a la `i`. No cal fer res més ja que no hi ha cap estudiant fins a `i-1`.
- *Condició de sortida.* Com que volem tenir al final tot el vector tractat, sortirem del bucle quan `i=N+1`, per tant ens quedem mentre `i =< N`.

- *Cos del bucle.* Com que hem d'avançar, el més natural és incrementar la `i` de manera que s'acosti al final del vector. Però abans d'incrementar la `i`, hem de satisfer l'invariant canviant `i` per `i+1`. Necessitem, per tant, que `vest` estigui modificat en `[1..i]`. Com que actualment `vest` està modificat en `[1..i-1]`, només ens cal preguntar si `vest[i]` té nota i en cas afirmatiu modificar-li la nota, un cop arrodonida. Si no té nota no cal fer res.

  Un cop que tenim `vest` modificat arrodonint la nota dels estudiants en `[1..i]`, ja podem incrementar la `i` assegurant que satisfem l'invariant (a més la condició del bucle ens assegura que `i+1 =< N+1`).

- *Acabament.* A cada volta decreix la distància entre `N+1` i la `i`.

El programa anotat queda:

```
acció arrodonir_notes (e/s vest: vector [1..N] d'Estudiant);
{Pre: --- }
var i: nat aux:real fvar;

i:= 1;

{ I: vest[1..i-1] té les notes dels estudiants arrodonides
     respecte al seu valor inicial, 1 =< i <= N+1}

Mentre i <= N fer

  { mirem si vest[i] té nota }
  Si te_nota(vest[i]) llavors

     { obtenim la nota de vest[i] arrodonida}
     aux:=arrodonir(consultar_nota(vest[i]));
     { modifiquem la nota de vest[i] amb la nota arrodonida }
     modificar_nota(vest[i],aux) { arrodonim la nota de vest[i] }

  fSi;
  i:= i+1
fMentre

{Post: vest té les notes dels estudiants arrodonides respecte
       al seu valor inicial }
```

Hem usat la variable `aux` per aumentar la legibilitat. Noteu que hem d'usar `modificar_nota` perque l'estudiant ja té nota. Finalment, implementem la funció `arrodonir`, considerant que en el lleguatge hi ha una funció `trunc` que agafa la part entera d'un real.

```
funció arrodonir(r: real) ret a:real;
{Pre: --- }

a:=trunc(10.*(r + 0.05)) / 10.0;

{Post: a és r arrodonit a la dècima més propera (i mes gran si hi ha empat)}
```

5 Implementació de mòduls

Implementar un mòdul de dades es descomposa en dues fases: implementar el tipus, és a dir, donar una representació determinada en termes dels tipus existents, i implementar les operacions, és a dir, codificar les seves operacions en termes d'instruccions. De fet, en la secció anterior acabem d'implementar unes operacions, però ara, en aquesta secció, ho farem en el context d'un mòdul. Teniu en compte que això no és el que entenem per disseny modular, ja que aquí l'especificació del mòdul ja ens ve donada.

És important remarcar en aquest punt la independència entre l'ús dels mòduls i la seva implementació. De fet, en aquesta secció implementarem un mòdul que ja hem usat, la qual cosa mostra que no calia coneixer la seva implementació.

Per implementar els nous tipus podrem usar qualsevol combinació dels tipus coneguts de que disponem: vectors, arbres, piles, enters, ... A més, utilitzarem molt sovint un tipus de dades que ens permetra agrupar elements de tipus diferents en una mateixa unitat. Aquest tipus és la **tupla**. Una tupla està formada per camps als que hem de donar nom. Per exemple, el tipus estudiant l'implementarem mitjançant una tupla amb tres camps DNI, NOTA i DEF (que és un camp boolea per indicar si l'estudiant té nota o no).

Per accedir al contingut dels camps d'una variable de tipus tupla, usarem el nom de la variable seguit d'un punt (".") i del nom del camp. Per exemple, si `est` és una variable del tipus tupla que hem esmentat anteriorment, `est.DNI` ens donaria el contingut del camp DNI de la tupla `est`.

La tupla és també un tipus genèric, ja que podem triar el tipus dels seus camps d'entre tots els tipus coneguts. D'aquesta manera podem tenir combinacions, com ara tuples amb un enter i un vector de piles de tuples amb un arbre i un boolea.

Anem a presentar la implementació del tipus `Estudiant` que exemplificarà amplement l'ús de tuples. No inclourem justificacions ja que no hi ha cap algorisme iteratiu ni recursiu. Ara bé, inclourem molts comentaris que ajudin a entendre les instruccions.

```
Mòdul Estudiant;

Implementació

Tipus Estudiant = tupla
                    DNI: enter;
                    NOTA: real;
                    DEF: bool
                  ftupla;

{ Descripció general : el camp DNI conté el DNI, el camp NOTA la nota i
                       el camp DEF ens diu si té nota o no}

Operacions

funció crear_estudiant(dni: enter) ret est:Estudiant;
{Pre: dni >= 0 }

 est.DNI := dni;  { introduïm el DNI }
 est.DEF := fals  { indiquem que no té nota }

{Post: est  és un estudiant amb DNI dni i sense nota}
```

```
funció afegir_nota(est:Estudiant; not:real) ret sest:Estudiant;
{Pre: est és un estudiant sense nota i  not >= 0}

 sest.DNI := est.DNI; { li posem el mateix DNI }
 sest.NOTA := not;    { li posem la nota not }
 sest.DEF := cert      { indiquem que té nota }

{Post: sest  és un estudiant amb el DNI de est i amb la nota not}


funció consultar_dni(est:Estudiant) ret dni: enter;
{Pre: --- }

 dni:= est.DNI

{Post: dni  és el DNI de l'estudiant  est }


funció consultar_nota(est:Estudiant) ret not: real;
{Pre: est és un estudiant amb nota}

 not:= est.NOTA

{Post: not  és la nota de l'estudiant  est}


acció modificar_nota(e/s est: Estudiant; ent not:real)
{Pre: est és un estudiant amb nota i  not >= 0}

 est.NOTA := not;  {posem a NOTA la nova nota}

{Post: la nota de est passa a ser not}


funció te_nota(est:Estudiant) ret b: bool;
{Pre: --- }

 b := est.DEF

{Post: b  ens diu si l'estudiant  est té nota o no}


FImplementació
```

Com a exercici, i per constatar la independència de la implementació, podeu fer una implementació alternativa del tipus `Estudiant`, en que en lloc de tenir tres camps a la tupla, eliminem el camp booleá i usem el camp NOTA també per identificar quan l'estudiant no té nota posant-li algun número negatiu (per exemple el `-1.0`). Així, si el camp és positiu vol dir que té nota i si és negatiu és que no en té.

És molt important notar que, si canviem la implementació triada, en cap cas hem de revisar els algorismes de càlcul de presentats i d'arrodoniment que hem fet en la secció anterior. **Només hem de revisar els algorismes que usen un mòdul si fem algun canvi en la seva especificació**.

5.1 Enriquiment sobre piles

Ara implementarem l'enriquiment sobre piles, que hem presentat en la secció 3.1. Noteu que no coneixem la implementació del mòdul `Pila`, així que l'enriquiment ha d'estar basat només en l'especificació. Com que en aquesta implementació usarem iteracions i recursivitat, després de la implementació de cada funció iterativa o recursiva inclourem la seva justificació informal. En tota la resta del document seguirem aquest mateix patró.

```
Mòdul opsPila;

Implementació

Sobre Pila;

Operacions

funció altura( p: Pila ) ret n: nat;
{ Pre: p = P }

n:=0;

{I: n = "nombre d'elements de la part tractada de P",
    p = "elements de P que queden per tractar" }

Mentre !es_buida(p) fer
   n:= n+1;
   p:=desempilar(p)
fMentre

{ Post: n = "nombre d'elements que conté P" }
```

Com que modifiquem el valor del paràmetre `p` hem hagut de modificar la especificació (mantenit el mateix significat), de manera que a la postcondició parlem del valor inicial de la `p`. Noteu que en el nostre llenguatge algorísmic considerem que els canvis fets als paràmetres d'entrada dins de la funció no tenen efectes fora. L'alternativa és no modificar la `p` i usar una variable auxiliar a la que li assignem `p` al principi.

Per a la funció `altura` la justificació informal ha estat la següent:

- *Inicialitzacions.* Incialment considerem que no hi ha cap element de `P` tractat (`p` conté tots els elements), per tant hem de posar un `0` a `n`.
- *Condició de sortida.* Com que volem tenir al final tota la pila `P` tractada, sortirem quan `p` no tingui cap element, és a dir sigui una pila buida. Per tant, ens quedem mentre `!es_buida(p)`.
- *Cos del bucle.* Com que hem d'avançar, el més natural és eliminar el cim de la pila `p` (per la condició sabem que no és buida). Però abans d'eliminar-lo, hem de satisfer l'invariant canviant

`p` per `desempilar(p)`. Necessitem, per tant, que `n` contingui el nombre d'elements de la part tractada després de desempilar (és a dir, els que no són a `desempilar(p)`). Com que, actualment, `n` conté el nombre d'elements que no són a `p`, només ens cal sumar-li `1` a `n`, per tenir el nombre d'elements que no són a `desempilar(p)`. Un cop que tenim `n` modificat, ja podem eliminar el cim de `p` assegurant que satisfem l'invariant.

- *Acabament.* A cada volta decreix el tamany de `p`.

Dissenyem ara la funció `iguals`, que resoldrem recursivament.

```
funció iguals( p, q: Pila ) ret b: bool;
{ Pre: --- }

Si es_buida(p) \/ es_buida(q)   --> b:= (es_buida(p) /\ es_buida(q));
[] !es_buida(p) /\ !es_buida(q) -->
            Si (cim(p) != cim(q)) --> b:= fals;
            [] (cim(p) = cim(q))  --> b:= iguals(desempilar(p),desempilar(q));
                     { HI: b = "desempilar(p) i desempilar(q) contenen
                         els mateixos elements afegits en el mateix ordre" }
            fSi;
fSi;

{ Post: b = "p i q contenen els mateixos elements afegits
                en el mateix ordre" }

FImplementació
```

La justificació informal d'aquesta funció recursiva ha estat la següent.

- *Casos senzills.*

  1. Si alguna de les dues piles és buida, llavors les piles són iguals si les dues són buides. Per tant assignem a `b` l'expressió `(es_buida(p) i es_buida(q))`.
  2. Si les dues piles no són buides, però els elements que hi ha al cim són diferents, llavors no poden ser iguals. Per tant posem `b` a fals.

- *Cas recursiu.* Si les piles no són buides i els cims són iguals, llavors les piles són iguals si coincideixen en la resta d'elements. Això ho podem obtenir, per hipòtesi d'inducció, cridant recursivament a la funció amb les piles sense l'element del cim. Per això assignem a `b` el resultat de la crida recursiva.

- *Acabament.* A cada crida recursiva fem més petita la primera pila.

5.2 Un exemple amb arbres binaris

Dissenyem ara una funció que, donat un arbre binari, ens calculi el seu tamany, és a dir, el nombre d'elements que conté.

```
funció tamany( a: arbre  ) ret n: nat;
{ Pre: --- }
```

```
Si es_buit(a)  --> n:= 0;
[] !es_buit(a)  -->
            n1:= tamany( fill_esq(a));
            n2:= tamany( fill_dret(a));
            { HI: n1 = "nombre d'elements de fill_esq(a)" /\
                  n2 = "nombre d'elements de fill_dret(a)" }
            n:=1+n1+n2;
fSi;

{ Post: n = "nombre d'elements d'a" }
```

La justificació informal d'aquesta funció recursiva ha estat la següent.

- *Cas senzill.*

  Si l'arbre és buit té zero elements i aquest és el valor que s'ha d'assignar a `n`.

- *Cas recursiu.* Si l'arbre no és buit, llavors existeixen els seus subarbres esquerre i dret. El tamany de l'arbre serà, doncs, la suma dels tamanys dels seus subarbres més 1 (la contribució de l'arrel).

  Els tamanys dels subarbres s'obtenen per hipòtesi d'inducció, cridant recursivament a la funció. Per últim, fem la suma i el programa queda complet.

- *Acabament.* A cada crida recursiva fem més petit l'arbre.

5.3 Un exemple amb cues

Escriguem una funció que busqui un element en una cua.

```
funció cerca_cua( c: cua; x: elem  ) ret b: bool;
{ Pre: --- }

Si es_buida(c)  --> b:= fals;
[] !es_buida(c)  -->
       Si primer(c) = x --> b:= cert
       [] primer(c) != x -->
            b:= cerca_cua( avançar(c), x  );
            { HI: b = x és a avançar(c)}
       fSi
fSi

{ Post: b = "x és a c" }
```

La justificació informal d'aquesta funció recursiva ha estat la següent.

- *Casos senzills.*

  1. Si la cua és buida no hi ha cap element i el resultat ha de ser fals.
  2. Si la cua no és buida, es descomposa en el seu primer element i la resta. Si el seu primer element és el que busquem, el resultat serà cert.

- *Cas recursiu.* Si la cua no és buida i el seu primer element no és el que busquem, tot dependrà de si aquest és troba a la resta de la cua. Això s'obté per hipòtesi d'inducció, cridant recursivament a la funció.
- *Acabament.* A cada crida recursiva fem més petit la cua.

Si volem la mateixa funció amb codi iteratiu

```
funció cerca_cua( c: cua; x: elem  ) ret b: bool;
{ Pre: c = C }

b:=fals;

{I: b = "x és a la part visitada de C",
    c = "elements de C que queden per visitar" }

Mentre !es_buida(c) /\ !b fer
   b:=(primer(c)=x);
   c:=avançar(c);
fMentre

{ Post: b = "x és a C" }
```

Per a aquesta versió la justificació informal ha estat la següent:

- *Inicialitzacions.* Incialment considerem que no hem visitat cap element de `C` i per tant `x` no és a la part visitada i `b` ha de ser fals.
- *Condició de sortida.*

  Sortirem si hem visitat tota `C` sense trobar `x` (és a dir, quan `c` sigui buida), o bé si `b` és cert (per l'invariant, això implica que hem trobat `x`).

  Per tant, ens quedem mentre `!es_buida(p)` $\wedge$ `!b`.
- *Cos del bucle.* A cada volta visitem un nou element (el primer de `c`). Per tant per tal de que es mantingui l'invariant, s'ha d'afegir a `b` la informació relativa a si aquest element és `x`. Com que en entrar al bucle `b` és fals, només hem d'asignar-li `primer(c)=x`.
- *Acabament.* A cada volta decreix el tamany de `c`.