Introducció al disseny modular i al disseny basat en objectes

Assignatura PRO2

Febrer 2015

Índex

1 Disseny modular I **5**

1.1 Mòduls i classes 6

1.1.1 Propietats desitjables dels programes 6

1.1.2 Abstracció funcional i de dades 7

1.1.3 Disseny de mòduls: distinció entre especificació i implementació 10

1.1.4 Encapsulament de dades en llenguatges orientats a objectes: classes i objectes; atributs i mètodes 10

1.2 Especificació i ús de la classe `Estudiant` 12

1.2.1 Especificació 12

1.2.2 Ús. Vectors d'`Estudiant` 15

2 Disseny modular II **19**

2.1 Dependència entre mòduls. Diagrames modulars 19

2.1.1 Ús d'una classe per una altra: `Cjt_estudiants` sobre `Estudiant` 19

2.1.2 Ampliació d'una classe: noves operacions de `Cjt_estudiants` 22

2.1.3 Ús de biblioteques 24

2.2 Implementació de classes 25

2.2.1 Declaració dels atributs i codi dels mètodes 25

2.2.2 La classe `Estudiant` 25

2.2.3 La classe `Cjt_estudiants` 29

2.2.4 Ampliació de la classe `Cjt_estudiants` 34

2.3 Metodologia de disseny modular 37

2.3.1 Resum d'elements metodològics 39

Capítol 1

Disseny modular I

En general, la introducció de la matèria d'un curs de programació i, en particular, la introducció del disseny modular en aquest curs, es pot orientar de dues maneres diferents:

- Basada en llenguatges:
  - Presentació de llenguatges concrets i comparació de les seves característiques principals.
  - Elecció d'un llenguatge concret i estudi de les seves característiques i aplicacions.
- Basada en conceptes:
  - Presentació de les característiques principals presents a la majoria de llenguatges d'interès per al curs i identificació de les propietats desitjables dels programes.
  - Estudi de tècniques associades a aquestes característiques per a la resolució de problemes i la construcció de programes que implementin les solucions amb les propietats desitjades.
  - Elecció d'un llenguatge concret com a exemple, cas d'estudi i eina de desenvolupament de treballs pràctics associats als conceptes introduïts al curs.

Una raó molt simple per no escollir el primer enfoc seria, per exemple, la gran quantitat de llenguatges diferents que s'han dissenyat amb característiques similars dins de l'àmbit d'interés del curs, que el faria summament repetitiu. En canvi, una orientació més conceptual pot permetre entendre més ràpidament les característiques comunes i les particularitats de diferents llenguatges, i l'anàlisi de l'adequació d'un llenguatge a un objectiu concret.

Per aquestes raons, en aquest curs, seguirem fonamentalment la segona orientació restringint-nos a la classe de llenguatges imperatius orientats a objectes. Usarem C++ com a llenguatge concret i eina per il·lustrar els conceptes introduïts i realitzar les pràctiques de laboratori.

En concret, en aquesta assignatura usarem l'orientació a objectes com a mecanisme del C++ per codificar dissenys basats en els conceptes de *mòdul* i *disseny modular*.

Hem de tenir en compte que l'elecció d'un llenguatge concret com el C++, imposa un cert estil de programació i una manera d'abordar els conceptes generals del curs, als que necessàriament ens haurem d'adaptar.

En aquests apunts, introduirem en primer lloc els conceptes associats a la modularitat com a mecanisme general d'estructuració de programes i els aplicarem al desenvolupament de programes en C++. Com ja hem dit, això requerirà la introducció d'aspectes concrets del llenguatge.

1.1 Mòduls i classes

1.1.1 Propietats desitjables dels programes

El nostre objectiu 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 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 la mida 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 llegibilitat 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 trossos qualssevol de mida petita, no és una política gaire 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 ells, de manera que es puguin fer canvis en un mòdul sense que això obligui a modificar els altres mòduls. Això és molt rellevant de cara a la modificabilitat i el manteniment dels programes, així com per facilitar el treball en equip.

- 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 equip.

Per aconseguir programes amb aquestes propietats necessitem d'una banda mecanismes de raonament i metodologies de treball adequats i, de l'altra, un llenguatge de programació que proporcioni les eines d'implementació necessàries.

1.1.2 Abstracció funcional i de dades

El mecanisme de raonament per aconseguir bones descomposicions modulars que considerarem en aquest curs és l'*abstracció*. Quan fem una abstracció d'un problema, el que fem és oblidar-nos de certs detalls per tal de transformar el nostre problema en un de més simple o més general.

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 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*. Una especificació Pre/Post està formada per tres elements.

- la **capçalera** de l'operació: resultat, nom, paràmetres, etc;
- la **precondició**: propietats que han de complir els paràmetres perquè l'operació faci el que està previst;
- la **postcondició**: propietats que han de complir els resultats després de cridar l'operació, incloent-hi els paràmetres per referència si volem modificar-los.

Una especificació d'aquest tipus ha d'interpretar-se com que, si les dades de l'operació satisfan la precondició, llavors els resultats compliran la postcondició. En cas contrari, l'operació podria donar errors d'execució o guardar valors absurds als resultats, però en qualsevol cas el responsable d'aquest mal funcionament és l'usuari de l'operació. Per a tota operació s'ha de disposar dels mitjans per comprovar la precondició abans de cada crida.

Per exemple, aquesta funció vol calcular potències de nombres enters:

```
int pot(int a, int b)
/* Pre: a>0 i b>=0 */
/* Post: el resultat és a multiplicat per si mateix b vegades */
```

Per usar la funció `pot` no ens cal saber com està dissenyada, només ens cal saber què han de complir les dades que li passem i què satisfaran els resultats quan la funció s'hagi executat. És a dir, l'especificació Pre/Post ens permet conèixer simplement *què* fa la funció o acció, sense necessitat de saber *com* ho fa.

Existeixen altres estils o formalismes d'especificació que, en general, ens permeten ocultar detalls d'implementació, i que seran més o menys adients depenent dels objectius fixats i de les necessitats concretes. L'especificació com a tal no és l'objectiu d'aquest curs però sí una eina necessària. Per mostrar el que necessitem, d'acord als objectius del curs, entrarem només una mica més en detall.

En el següent exemple d'especificació s'explica que, donat un conjunt de reals `Creals` guardats en un vector, l'acció `arrodonir` el modifica arrodonint-ne els valors que conté a la dècima més propera.

```
void arrodonir(vector<double> &Creals);
/* Pre: cert */
/* Post: Creals conté els valors originals arrodonits a la dècima més propera */
```

Igual que al cas de la funció `pot`, ens abstraiem de com s'implementaria `arrodonir` i només mostrem què fa i com s'usaria. Una conseqüència molt important i molt útil, que usarem en aquest curs, és que qualsevol canvi a la seva implementació que no afecti a la seva Pre/Post tampoc no afectarà el seu ús. En aquest sentit, podríem considerar que una especificació d'aquest estil és com una mena de contracte d'ús de l'operació.

Per fixar què volem en aquest curs, anem una mica més enllà. Ens pot interessar fer abstracció de la implementació concreta del conjunt de reals fent servir un nou tipus `ConjReals` que ens permeti escriure

```
void arrodonir(ConjReals &Creals);
/* Pre: cert */
/* Post: Creals conté els valors originals arrodonits a la dècima més propera */
```

Fixem-nos que ara, a més a més, un possible usuari d'aquesta operació no necessita saber com s'implementen els conjunts de reals per poder usar-la. D'aquesta manera, canvis en la implementació del nou tipus no tenen perquè afectar l'ús de l'acció `arrodonir`.

Fins aquí arriba l'ús de l'abstracció per especificació que ens interessa ja que, com veurem, el nostre objectiu és garantir la *propietat d'independència de la implementació* entre les unitats (mòduls) de programes que dissenyarem i, disposar d'una *metodologia de disseny modular* que ens ajudi a raonar sobre la descomposició en mòduls dels nostres programes.

Com ja hem dit, existeixen altres mecanismes d'*abstracció per especificació* que permeten assolir altres objectius que s'escapen als d'aquest curs. Per exemple, si volguéssim fer algun tipus de demostració de la correcció dels nostres programes necessitaríem que les nostres especificacions Pre/Post estiguessin escrites en un llenguatge formal o matemàtic com la lògica de predicats (veure [Bal93, Pe05]).

El problema, aleshores, és que precondicions i postcondicions formals no necessàriament poden ocultar per si mateixes com estan implementats els paràmetres. No és difícil veure-ho si formalitzem l'anterior especificació de la següent manera

```
void arrodonir(ConjReals &Creals);
/* Pre:  ∀i : 1 ≤ i ≤ N : Creals[i] = R_i  */
/* Post: ∀i : 1 ≤ i ≤ N : Creals[i] = int(10.0 * (R_i + 0.05))/10.0  */
```

Només direm que, si volguéssim mantenir l'abstracció de dades hauríem d'escollir altres formes d'especificació diferents a la Pre/Post com, per exemple, la que es diu *especificació axiomàtica* (veure [Bal93, Pe05]), que permet expressar propietats dels tipus de dades usant també expressions lògiques i, a més, permeten demostrar la correció de les seves possibles implementacions.

Com veurem en temes posteriors, com que les nostres especificacions seran Pre/Post escrites en llenguatge no formal podrem mantenir l'abstracció de dades però només podrem arribar a fer raonaments també no formals sobre la correcció dels programes.

Resumint, en general, l'*abstracció per especificació* oculta els detalls d'implementació i, en aquest curs, usarem especificació Pre/Post no formal per realitzar dos tipus de descomposició o abstracció:

1. ***Abstracció funcional*** Aquest tipus de descomposició ja apareix en problemes de mida petita i ja s'ha usat en el curs anterior. Consisteix en "encarregar" la solució d'una part del problema a alguna operació independent, descrita simplement amb els seus paràmetres i la seva especificació i deixant la seva implementació per a futurs refinaments. Podem considerar que afegim noves operacions al nostre llenguatge de programació que ens permeten resoldre el nostre problema.

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

   - 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.

Així doncs, podem considerar dos tipus de mòduls depenent, respectivament, de si només encapsulem operacions amb la seva especificació, o, a més a més, aquestes operacions estan definides sobre un nou tipus de dades:

- ***Mòduls de dades***. Contindran la definició d'un nou tipus i les seves operacions. Aquest serà el tipus de mòdul usat en els nostres dissenys.

- ***Mòduls funcionals***. Contindran un conjunt d'operacions noves necessàries per resoldre algun problema o subproblema.

En tots dos casos s'ocultarà la implementació concreta de les operacions i, al primer cas, també la definició concreta del nou tipus de dades.

1.1.3 Disseny de mòduls: distinció entre especificació i implementació

Com ja hem dit, l'abstracció ens permet definir mòduls independents. Per assegurar-nos d'aconseguir aquesta independència i gaudir dels seus avantatges, és fonamental que respectem les següents fases en el disseny de mòduls:

1. **Fase d'especificació:** Suposarem l'existència d'una representació i d'unes operacions per manipular-la. És a dir, ens abstraurem de representacions concretes però assumirem un cert comportament de les operacions. En aquest sentit, *una especificació és una mena de contracte d'ús del tipus de dades*.

2. **Fase d'implementació:** Decidirem la representació més adequada i la codificació de les operacions sobre aquesta representació. Aquesta representació s'ha de poder canviar quan es consideri oportú sense que això afecti a l'especificació.

En aquest capítol i en el següent usarem la definició d'un nou tipus `Estudiant` per il·lustrar en què consisteix cada fase, però abans hem d'introduir la notació i les eines bàsiques del llenguatge C++ per poder fer-ho.

1.1.4 Encapsulament de dades en llenguatges orientats a objectes: classes i objectes; atributs i mètodes

Com ja hem dit, un *mòdul de dades* defineix un *tipus* com un determinat domini de valors i un conjunt d'operacions que treballen sobre diversos paràmetres. Si aquestes operacions fossin implementades com accions o funcions estàndards de C++ (o de qualsevol altre llenguatge), un d'aquests paràmetres hauria de ser del tipus que s'està definint i representaria la dada sobre la qual actua l'operació.

Com veurem, no és exactament així com funcionen les coses en la majoria de llenguatges orientats a objectes. Concretament, aquests llenguatges permeten encapsular dades en unitats o mòduls anomenats *classes* que defineixen una estructura d'*atributs* (representació del tipus) i operacions anomenades *mètodes*.

Donada una classe, podem definir *objectes* del tipus de la classe. A la programació clàssica, una *variable* pot *contenir* un valor del seu tipus i es pot fer servir a les operacions o expressions definides sobre el seu tipus. En canvi, un *objecte no és un contenidor de dades sinó una instància de la classe*. A nivell conceptual, això significa que la classe defineix una mena de patró de com han de ser els objectes d'un cert tipus. De fet, aquest concepte és molt més proper al significat que donem quotidianament a les paraules classe i objecte d'una classe o d'un tipus. La classe és una descripció genèrica i l'objecte és un espècimen concret d'una classe.

En la terminologia orientada a objectes es diu que *cada objecte és propietari dels seus atributs i mètodes* i això, al nivell més pràctic, té unes certes implicacions. Per exemple, com que els mètodes són considerats com uns components més dels objectes (al igual que els atributs), aquests no hi figuren com a paràmetres. És a dir, l'objecte propietari d'un mètode (que pot ser creat, consultat o modificat per aquest) no apareix explícitament a la seva capçalera. Per això, ens referirem a ell com **paràmetre implícit**. No obstant, quan es fan crides al mètode, l'objecte

apareix a cada crida particularitzant el nom del mètode. Evidentment, això també implicarà certs canvis en la forma d'implementar mètodes respecte a com implementàvem les accions i funcions estàndards (per exemple de C++), però això ho veurem al tema següent.

Per exemple, considerem l'existència d'una operació `te_nota` que pertany a una classe `Estudiant` (veurem la versió completa a l'apartat següent). La seva capçalera en C++, si no s'usa orientació a objectes, seria

```
bool te_nota(const Estudiant &e)
/* Pre: cert */
/* Post: el resultat indica si e té nota */
```

En canvi, si fem servir orientació a objectes s'haurà d'escriure

```
bool te_nota() const
/* Pre: cert */
/* Post: el resultat indica si el paràmetre implícit té nota */
```

Fixem-nos que en aquesta capaçalera no apareix cap nom d'estudiant explícitament, per això hem dit que l'objecte consultat és el paràmetre implícit de l'operació.

En general, quan fem una crida a un mètode d'una classe, l'objecte sobre el que s'aplica precedeix al nom del mètode i a la resta de paràmetres, separat per un punt

```
<nom_de_l'objecte>.<nom_del_mètode>(<altres paràmetres>)
```

A l'exemple, si `est` és un objecte ja creat de la classe `Estudiant` i `b` és una variable booleana, la crida **no** es realitzarà així

```
b = te_nota(est);
```

sinó en la forma

```
b = est.te_nota();
```

Una altra qüestió pràctica a tenir en compte és que el paràmetre implícit de tot mètode d'una classe funciona a tots els efectes com un paràmetre per referència del C++. Ara bé, igual que pels paràmetres convencionals, podem usar la paraula `const`, com a l'exemple anterior, per evitar que el mètode pugui modificar-lo.

En canvi, si volem implementar en C++ un mètode `modificar_nota` dins de la classe `Estudiant`, la seva especificació seria

```
void modificar_nota(double nota)
/* Pre: el paràmetre implícit té nota i "nota" és una nota vàlida */
/* Post: la nota del paràmetre implícit passa a ser "nota" */
```

Fixem-nos que s'aprofita la modificabilitat del paràmetre implícit per canviar-li la nota. Per exemple, si `est` és un `Estudiant` amb nota i `x` és una nota vàlida, la nota d'`est` passarà a ser `x` si fem la crida

```
est.modificar_nota(x);
```

Per tant, si volem conservar el valor original d'un objecte després d'executar una operació de modificació, haurem d'obtenir una còpia de l'objecte prèviament a la crida.

1.2 Especificació i ús de la classe `Estudiant`

Suposem que per gestionar les notes d'una assignatura volem crear un nou tipus de dades amb la informació dels estudiants. En aquest apartat presentarem l'especificació i alguns casos d'ús de la classe corresponent, que servirà per introduir la notació necessària de C++ que usarem al llarg del curs. En el tema següent, usarem el mateix exemple per introduir com s'implementen les classes en C++.

1.2.1 Especificació

Presentem a continuació un exemple del conjunt de mètodes que podria contenir una classe `Estudiant` i de la notació de C++ necessària per especificar-la. Noteu que a la part etiquetada amb `private` no hem escrit res. Més endavant veurem com aquest espai es fa servir, entre d'altres propòsits, per declarar els atributs de la representació de la classe. Recordem que l'objectiu és aconseguir independència de la implementació i, per això, prohibirem que els atributs apareguin a l'especificació de la classe i a les especificacions Pre/Post dels mètodes. En canvi, les capçaleres dels mètodes i la seva especificació apareixen a la part etiquetada amb `public`, ja que és la informació que la classe ofereix a d'altres programes o classes per poder ser usada.

```
class Estudiant {

// Tipus de mòdul: dades
// Descripció del tipus: conté el DNI d'un estudiant,
// que és un enter no negatiu, i pot tenir nota, que seria un double

private:

public:

  //Constructores

  Estudiant();
  /* Pre: cert */
  /* Post: el resultat és un estudiant amb DNI=0 i sense nota  */

  Estudiant(int dni);
  /* Pre: dni >= 0 */
  /* Post: el resultat és un estudiant amb DNI=dni i sense nota */

  // Destructora: esborra automàticament els objectes locals en sortir
  // d'un àmbit de visibilitat

  ~Estudiant();

  //Modificadores
```

```
  void afegir_nota(double nota);
  /* Pre: el paràmetre implícit no té nota, 0 <= "nota" <= nota_maxima() */
  /* Post: la nota del paràmetre implícit passa a ser "nota" */

  void modificar_nota(double nota);
  /* Pre: el paràmetre implícit té nota, 0 <= "nota" <= nota_maxima() */
  /* Post: la nota del paràmetre implícit passa a ser "nota" */

  //Consultores

  int consultar_DNI() const;
  /* Pre: cert  */
  /* Post: el resultat és el DNI del paràmetre implícit */

  bool te_nota()  const;
  /* Pre: cert  */
  /* Post: el resultat indica si el paràmetre implícit té nota o no */

  double consultar_nota() const;
  /* Pre: el paràmetre implícit té nota */
  /* Post: el resultat és la nota del paràmetre implícit */

  static double nota_maxima();
  /* Pre: cert */
  /* Post: el resultat és la nota màxima dels elements de la classe */

  // Lectura i escriptura

  void llegir();
  /* Pre: estan preparats al canal estandar d'entrada un enter no negatiu i
     un double */
  /* Post: el paràmetre implícit passa a tenir els atributs llegits del canal
     estàndard d'entrada; si el double no pertany a l'interval [0..nota_maxima()],
     el p.i. es queda sense nota  */

  void escriure() const;
  /* Pre: cert */
  /* Post: s'han escrit els atributs del paràmetre implícit al canal estàndard
     de sortida: el dni seguit de un NP o la nota en format double,
     separats per un espai en blanc */
};
```

Com mostrarem a la següent secció, per usar la classe `Estudiant` no cal saber ni com s'ha implementat el tipus `Estudiant` ni les seves operacions. En particular, no cal saber com s'ha gestionat internament la qüestió de tenir nota o no (per exemple, amb un valor booleà o usant

valors negatius a la nota).

Fixeu-vos que hem classificat les operacions en les quatre següents categories:

- ***Creadores d'objectes***. Són funcions que serveixen per crear objectes nous amb una informació mínima inicial o resultat de càlculs més complexos.

  El llenguatge C++ proporciona un cas particular de creadores anomenades *constructores*, que tenen el mateix nom de la classe i retornen un objecte nou d'aquest tipus, sense paràmetre implícit. Per això a la seva capçalera apareix aquest nom una sola vegada. Aquestes operacions s'executen automàticament quan declarem un objecte nou.

  Per exemple, la declaració

  ```
  Estudiant est;
  ```

  realitza una crida a la constructora `Estudiant()` i produeix un estudiant amb dni igual a zero i sense nota.

  Podem definir diferent versions de constructores d'una classe diferenciades per la seva llista de paràmetres. Per exemple, per crear un estudiant amb un DNI inicial `x` i sense nota escriuríem

  ```
  Estudiant est(x);
  ```

  En cas de no definir-se cap d'aquestes, C++ proporciona una per defecte sense paràmetres, que crea un objecte nou sense informació definida. Això implica que si una classe té una constructora d'aquestes amb algun paràmetre, llavors no es pot suposar que disposi d'una altra sense paràmetres, tret que s'hagi definit explícitament.

- ***Modificadores***. Transformen el paràmetre implícit amb informació aportada per altres paràmetres, si cal. Conceptualment, no haurien de poder modificar altres objectes encara que C++ ho permet via el pas de paràmetres per referència. Normalment, seran accions (`void`).

  Exemples típics són les operacions `afegir_nota` i `modificar_nota`. La raó per distingir-les 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 augmenta tant la fiabilitat en l'ús del tipus com la llegibilitat (i per tant la modificabilitat i la mantenibilitat).

- ***Consultores***. Proporcionen informació sobre el paràmetre implícit, potser amb ajuda d'informació aportada per altres paràmetres. Generalment són funcions, llevat que hagin de retornar més d'un resultat, en aquest cas poden ser accions amb més d'un paràmetre per referència.

  Exemple: l'operació `consultar_nota`. Per obtenir la nota d'un estudiant `est` i guardar-la en una variable `x` escriuríem

```
double x = est.consultar_nota();
```

La necessitat de definir les consultores `te_nota` i `nota_maxima` apareix perquè hi ha operacions que tenen com a requisit que l'estudiant tingui o no tingui nota o que un determinat valor `double` sigui una nota vàlida, i hem de tenir la possiblitat de comprovar aquests fets.

- ***Lectura i escriptura***. Es comuniquen amb els canals estàndard, llegint objectes de la classe o escrivint-los. Normalment són accions, tot i que les operacions d'escriptura no modifiquen el paràmetre implícit. Amb freqüència, en una primera aproximació suposem que les dades ja tenen el format correcte, i completem el detalls necessàris després de la implementació, ja que generalment no els coneixem quan estem especificant. Aquest és l'únic cas on permetem això. Al nostre exemple tenim `llegir_estudiant` i `escriure_estudiant`.

1.2.2 Ús. Vectors d'`Estudiant`

Tal com hem explicat, per usar un mòdul ha de ser suficient amb disposar de la seva especificació. Suposeu que volem fer un programa que donat un vector d'estudiants ens digui el percentatge de presentats, és a dir, d'estudiants amb nota.

Especificació

```
double presentats(const vector<Estudiant> &vest)
/* Pre: vest conté almenys un element */
/* Post: el resultat és el percentatge de presentats de vest */
```

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 següent (l'`i`-èssim).

El programa anotat queda de la següent manera

```
double presentats(const vector<Estudiant> &vest)
/* Pre: vest conté almenys un element */
/* Post: el resultat és el percentatge de presentats de vest */
{
  int numEst = vest.size();
  int n = 0;
  for (int i = 0; i < numEst; ++i) if (vest[i].te_nota()) ++n;
  /* n és el nombre d'estudiants amb nota de vest */
  double pres = n*100./numEst;
  return pres;
}
```

Un altre exemple és el següent: Dissenyarem un programa que, donat un vector d'estudiants, el modifica arrodonint-ne les notes a la dècima més propera mitjançant una acció.

```
void arrodonir_notes(vector<Estudiant> &vest);
/* Pre: cert */
/* Post: vest té les notes dels estudiants arrodonides respecte
         al seu valor inicial */
```

Per simplificar el disseny hem suposat, per abstracció funcional, que tenim una funció que s'encarrega d'arrodonir un `double`. Aquesta funció la dissenyarem al final, i d'aquesta manera separem el problema tècnic de com fer l'arrodoniment d'un `double`, del nostre problema general que és tractar les notes d'un vector d'estudiants, arrodonint-les.

Així el programa comentat proposat és

```
void arrodonir_notes(vector<Estudiant> &vest)
/* Pre: cert */
/* Post: vest té les notes dels estudiants arrodonides respecte
       al seu valor inicial */
{
  int numEst = vest.size();
  for (int i = 0; i < numEst; ++i) {
    /* mirem si vest[i] té nota */
    if (vest[i].te_nota()) {
      /* obtenim la nota de vest[i] arrodonida */
      double aux = arrodonir(vest[i].consultar_nota());
      /* modifiquem la nota de vest[i] amb la nota arrodonida */
      vest[i].modificar_nota(aux);
    }
  }
}
```

Hem usat la variable `aux` per augmentar la llegibilitat. Noteu que hem d'usar `modificar_nota` perquè l'estudiant ja té nota. Finalment, implementem la funció `arrodonir`, usant la funció `int()` de C++. Noteu que si arrodonim la nota d'un estudiant (que per definició és vàlida), el resultat també és una nota vàlida i per tant es compleix la precondició de la crida a `modificar_nota`.

```
double arrodonir(double r)
/* Pre: cert */
/* Post: el resultat és el valor original de r arrodonit a
         la dècima més propera (i més gran si hi ha empat) */
{
  return int(10.*(r + 0.05)) / 10.0;
}
```

Per acabar, presentem l'especificació de la cerca per DNI d'un estudiant a un vector d'estudiants

```
bool cerca_lineal(const vector<Estudiant> &vest, int dni)
/* Pre: cert */
/* Post: el resultat indica si l'estudiant amb DNI = dni hi és a vest */
```

i la seva implementació

```
bool cerca_lineal(const vector<Estudiant> &vest, int dni)
/* Pre: cert */
/* Post: El resultat indica si l'estudiant amb DNI = dni hi és a vest */
{
  bool b = false;
  int numEst = vest.size();
  int i = 0;
  while (i < numEst and not b) {
    if (vest[i].consultar_DNI() == dni) b = true;
    else ++i;
  }
  return  b;
}
```

Capítol 2

Disseny modular II

2.1 Dependència entre mòduls. Diagrames modulars

En aquest apartat veurem com podem definir nous mòduls funcionals o de dades sobre tipus de dades ja existents, ja siguin predefinits pel llenguatge o dissenyats per nosaltres o qualsevol altre programador. Com veurem a l'últim apartat d'aquest tema, això indueix una certa metodologia de disseny de programes modulars basats en la relació entre classes derivada de les dependències entre els tipus de dades associats al problema que es vol resoldre.

En aquest curs considerarem un tipus de dependència molt bàsic que és la *relació d'ús* d'un mòdul per un altre. Aquesta apareix fonamentalment per dos mecanismes: la definició de nous tipus de dades a partir d'altres tipus ja existents, i l'*ampliació* d'un tipus amb noves operacions.

Direm que aquestes relacions establertes entre mòduls són *visibles*, en contraposició a d'altres relacions que poden aparèixer com a resultat d'implementacions concretes i que en direm *ocultes*.

Els **diagrames modulars** que veurem al llarg del curs són representacions gràfiques de les relacions d'ús visibles entre els diferents mòduls que formen un programa. Sempre tindrem un mòdul distingit (representat a la part superior del diagrama) que contindrà el programa principal. Normalment la resta de mòduls seran mòduls de dades (denotats per una doble linia horitzontal). Immediatament per sota del mòdul del programa principal tindrem aquells mòduls que són usats directament des d'aquest. Més abaix tindríem els mòduls usats per aquests últims i així successivament. Tot i que de vegades ens referim a la *jerarquia* de mòduls (o de classes) d'un programa, en realitat un diagrama modular no serà sempre un arbre, sinó quelcom una mica més general, en concret un graf dirigit acíclic, como el que mostrem a la figura 2.1.

2.1.1 Ús d'una classe per una altra: `Cjt_estudiants` sobre `Estudiant`

Al tema anterior van presentar una especificació de la classe `Estudiant`. Ara, podem plantejarnos definir un nou tipus de dades per manipular conjunts d'estudiants. A continuació, presentem una especificació de la classe `Cjt_estudiants` que defineix aquest nou tipus usant la classe `Estudiant`. En C++ aquesta relació s'estableix mitjançant la paraula `include` seguida del nom de l'arxiu de C++ que conté l'especificació de la classe usada, en aquest cas `Estudiant.hh`. No-

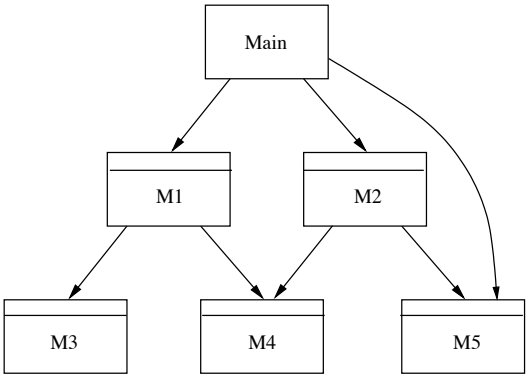

Figura 2.1: Exemple de diagrama modular

teu que per especificar la nova classe no cal disposar de la implementació de la classe `Estudiant`.

```
#include "Estudiant.hh"

class Cjt_estudiants {

  // Representa un conjunt d'estudiants ordenat per DNI. Els seus elements es
  // poden consultar i modificar a partir d'un DNI donat o per posició a l'ordre.

  private:

  public:

  //Constructores

  Cjt_estudiants();
  /* Pre: cert */
  /* Post: el resultat és un conjunt d'estudiants buit */

  //Destructora

  ~Cjt_estudiants();

  //Modificadores

  void afegir_estudiant(const Estudiant &est);
  /* Pre: el paràmetre implícit no conté cap estudiant amb el DNI d'est; el
     nombre d'estudiants del p.i. es menor que la mida màxima permesa */
  /* Post: s'ha afegit l'estudiant est al paràmetre implícit */
```

```
void modificar_estudiant(const Estudiant &est);
/* Pre: existeix un estudiant al paràmetre implícit amb el DNI d'est  */
/* Post: l'estudiant del paràmetre implícit amb el DNI d'est
   ha quedat substituït per est */

void modificar_iessim(int i, const Estudiant &est);
/* Pre: 1 <= i <= nombre d'estudiants del paràmetre implícit,
   l'element i-èssim del conjunt en ordre creixent per DNI conté
   un estudiant amb el mateix DNI que est */
/* Post: l'estudiant i-èssim del paràmetre implícit
   ha estat substituït per l'estudiant est*/

//Consultores

int mida() const;
/* Pre: cert */
/* Post: el resultat és el nombre d'estudiants del paràmetre implícit */

static int mida_maxima();
/* Pre: cert */
/* Post: el resultat és el nombre maxim d'estudiants que pot arribar
   a tenir el paràmetre implícit */

bool existeix_estudiant(int dni) const;
/* Pre: dni >= 0 */
/* Post: el resultat indica si existeix un estudiant al paràmetre implícit
   amb DNI = dni */

Estudiant consultar_estudiant(int dni) const;
/* Pre: existeix un estudiant al paràmetre implícit amb DNI = dni */
/* Post: el resultat és l'estudiant amb DNI = dni que conté el
   paràmetre implícit */

Estudiant consultar_iessim(int i) const;
/* Pre: 1 <= i <= nombre d'estudiants que conté el paràmetre implícit */
/* Post: el resultat és l'estudiant i-èssim del paràmetre implícit
   en ordre creixent per DNI */

// Lectura i escriptura

void llegir();
/* Pre: estan preparats al canal estàndar d'entrada un enter entre 0 i
   la mida maxima permesa, que representa el nombre d'elements que llegirem,
   i les dades de tal nombre d'estudiants diferents  */
/* Post: el paràmetre implícit conté el conjunt d'estudiants llegits
```

```
     del canal estàndard d'entrada */

  void escriure() const;
  /* Pre: cert */
  /* Post: s'han escrit pel canal estàndard de sortida els estudiants del
     paràmetre implícit en ordre ascendent per DNI */
};
```

Veureu que les operacions de lectura i escriptura es diuen igual que les de la classe `Estudiant`. Això no és un problema, ja que serà el objecte que usem per cridar-les el que triarà l'operació corresponent a la seva classe.

Noteu que si es vol comprovar la precondició de `afegir_estudiant`, es pot fer fàcilment amb `existeix_estudiant`, `mida` i `mida_maxima`.

2.1.2 Ampliació d'una classe: noves operacions de `Cjt_estudiants`

La situació que ens plantegem ara és diferent. No volem definir un nou tipus de dades sinó incorporar noves funcionalitats (més mètodes) a un tipus ja existent. És el que en diem *ampliació* del tipus. Com a exemple, partim de l'especificació de la classe `Cjt_estudiants` i considerem que ens demanen una operació d'esborrat d'estudiants (o ens adonem que fóra interessant disposarne d'una). Una altra operació interessant podria ser la consulta de l'estudiant amb nota màxima d'un conjunt.

Per fer ampliacions tenim bàsicament tres opcions. A continuació, descrivim breument cadascuna d'elles. A l'apartat d'implementació en presentarem exemples.

1. *Modificar la classe* existent per afegir els nous mètodes. És l'opció més simple però té implicacions que hem de tenir en compte. Concretament, hauríem de modificar la implementació però també l'especificació de la classe. És a dir, canviaríem el "contracte" d'ús de la classe, afectant la forma de ser usada pels altres mòduls o programes. En cert sentit, trenquem la independència entre mòduls desitjada, però ho podrem fer si tenim accés a la implementació original i els beneficis ho justifiquen.

   A l'especificació de l'apartat anterior, afegiríem

   ```
   class Cjt_estudiants {
   ...

   // Modificadores
   ...
   void esborrar_estudiant(int dni);
   /* Pre: existeix un estudiant al paràmetre implícit amb DNI = dni */
   /* Post: el paràmetre implícit conté els mateixos estudiants que
      l'original menys l'estudiant amb DNI = dni */

   // Consultores
   ```

```
...
Estudiant estudiant_nota_max() const;
/* Pre: el paràmetre implícit conté almenys un estudiant amb nota */
/* Post: el resultat és l'estudiant del paràmetre implícit amb nota màxima;
   si en té més d'un, és el de DNI més petit */
};
```

2. Definir les noves operacions *fora de la classe*. No es modifica ni l'especificació ni la implementació de la classe original i, per tant, tampoc com és el seu ús. Les noves operacions s'especifiquen i s'implementen allà on es facin servir o en un nou mòdul funcional, donant lloc en aquest segon cas a un *enriquiment* de l'original. El desavantatge és que els nous mètodes no serien orientats a objecte, és a dir, no serien propietat dels objectes de la classe original. A tots els efectes pràctics, serien accions o funcions convencionals amb totes les conseqüències d'ús de notació, etc. Això es pot fer sempre, encara que els resultats podrien no ser tan bons com amb el mètode anterior.

   Per especificar un mòdul funcional en C++ no farem servir cap classe, sinó que definirem en un arxiu `.hh` les especificacions Pre/Post de les noves operacions, tenint en compte que aquestes no tindran cap paràmetre implícit associat. Per exemple, podríem definir un arxiu `E_Cjt_estudiants.hh` amb el següent contingut

```
#include "Estudiant.hh"
#include "Cjt_estudiants.hh"

void esborrar_estudiant(Cjt_estudiants &Cest, int dni);
/* Pre: existeix un estudiant a Cest amb DNI = dni */
/* Post: Cest conté els mateixos estudiants que el seu valor original
   menys l'estudiant amb DNI = dni */

Estudiant estudiant_nota_max(const Cjt_estudiants &Cest);
/* Pre: Cest conté almenys un estudiant amb nota */
/* Post: el resultat és l'estudiant de Cest amb nota màxima;
   si en té més d'un, és el de DNI més petit */
```

   Com veurem en la següent secció, per separar la implementació de l'especificació del mòdul funcional, implementarem les noves operacions en un arxiu `.cc` que no serà visible pels mòduls que usin l'enriquiment.

   El diagrama modular resultant, amb un hipotètic programa principal que només provi les dues operacions de `E_Cjt_estudiants`, tindria l'aspecte mostrat a la figura 2.2

   És important remarcar que el fet d'agrupar les noves operacions en un nou mòdul funcional només té un cert sentit en el cas que considerem aquestes noves operacions com prou generals com per ser usades hipotèticament des de diferents mòduls o en la resolució de

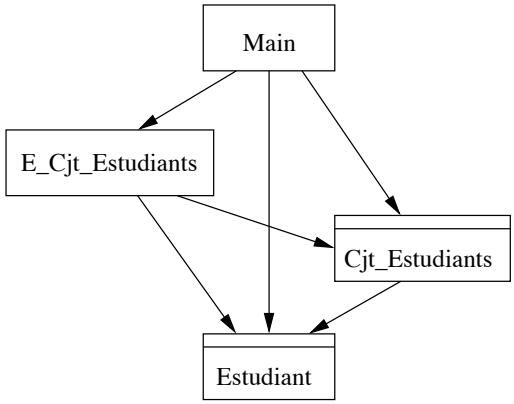

Figura 2.2: Diagrama modular de l'exemple amb `E_Cjt_estudiants`

diferents problemes. Per contra, si les noves operacions només es necessiten per resoldre un problema molt particular, llavors és molt millor especificar-les i implementar-les com a operacions auxiliars o privades de la classe on calguin. Veurem alguns exemples d'això últim als problemes de disseny modular que presentarem al llarg del curs.

3. Extensió de la classe usant l'*herència* del llenguatge. L'herència permet definir nous tipus de dades que heretin tant atributs com mètodes d'un tipus definit per una classe existent. És un tipus de relació entre classes fonamental als llenguatges orientats a objecte i el seu ús permet un tipus de programació amb notables avantatges, per exemple una molt òbvia és l'aprofitament de feina existent. No entrarem en més detall ja que l'herència s'estudiarà en cursos posteriors, juntament amb d'altres mecanismes d'extensió de classes.

   En el cas de la classe `Cjt_estudiants`, l'ús d'herència ens permetria definir una nova classe que heretés els atributs i els mètodes de l'original i que incorporés els mètodes nous. Però això també té el seu contra: en realitat definiríem un nou tipus de dades, és a dir, els nous mètodes serien propietat només dels objectes de la nova classe.

2.1.3 Ús de biblioteques

Com hem vist a l'apartat anterior, quan un programa consisteix en més d'un arxiu la instrucció `#include` ens permet usar elements procedents d'altres classes. Un altre exemple d'inclusió és l'ús d'elements procedents de biblioteques del propi llenguatge C++. Les biblioteques són col·leccions de procediments o classes que extenen el llenguatge i són d'utilitat en el desenvolupament dels programes. Cal distingir entre les biblioteques estàndards d'un llenguatge i qualsevol altra biblioteca que l'extengui.

La *Standard C++ Library* és la biblioteca estàndard de C++ i proporciona les funcionalitats bàsiques per realitzar diferent tasques com, per exemple, la interacció amb el sistema operatiu, contenidors i manipuladors de dades i algorismes d'ús habitual.

Mòduls típics de la biblioteca estàndard que usarem en aquest curs són: `<iostream>`: canals estàndard d'entrada i sortida; `<string>`: classe *string*; `<cmath>`: funcions matemàtiques.

La biblioteca estàndard també proporciona `templates`, que són una mena de classes genèriques en el sentit que, per exemple, ens permeten declarar diferents objectes que poden, a la seva vegada, contenir diferents tipus. Un exemple que ja coneixeu del curs anterior és el `<vector>`. En aquest curs també usarem els `templates` `<queue>`, `<stack>` i `<list>`.

2.2 Implementació de classes

2.2.1 Declaració dels atributs i codi dels mètodes

Implementar una classe de dades es descomposa en dues fases: implementar el tipus, és a dir, donar una representació determinada dels seus atributs en termes dels tipus existents, i implementar les operacions, és a dir, codificar les seves operacions en termes d'instruccions. De fet, ja hem implementat operacions com accions i funcions, però ara, hem d'implementar mètodes en el context d'una classe i, per tant, haurem d'introduir la notació del llenguatge C++ necessària per fer-ho.

És important remarcar en aquest punt la necessària independència entre l'ús dels mòduls i la seva implementació. De fet, implementarem la classe `Estudiant` que hem usat prèviament, la qual cosa mostra que no calia conèixer la seva implementació. Com ja hem comentat, C++ ens proporciona les paraules `private` i `public` com a eina per determinar quines parts seran usables i quines no, respectivament, des de fora de la classe. Així, el atributs seran implementats com a privats, perquè només es puguin manipular des de fora de la classe a través de crides als mètodes amb capçaleres definides com públiques.

Per implementar els atributs del nou tipus podrem usar qualsevol combinació dels tipus coneguts dels quals disposem (vectors, booleans, enters, ...) o qualsevol altre que creem. Dins de la classe, quan implementem els mètodes, hem de poder accedir al contingut dels atributs d'un objecte. La notació que usarem és el nom de l'objecte seguit d'un punt (".") i del nom de l'atribut. Per exemple, si `x` és un objecte de tipus `T`, que s'ha definit amb un atribut `camp`, l'expressió `x.camp` ens dóna el valor de l'atribut corresponent.

Una implementació d'una classe en C++ està formada normalment per dos arxius:

- el `.hh`, que ha de contenir almenys les capçaleres dels mètodes i la representació del tipus
- el `.cc`, que ha d'incloure almenys el codi dels mètodes.

A continuació presentem la implementació de la classe `Estudiant`.

2.2.2 La classe `Estudiant`

Definirem l'arxiu `Estudiant.hh` en primer lloc. Les capçaleres dels mètodes són les mateixes que a la seva especificació i com ja hem dit es declaren públics. No repetim les especificacions de les operacions, que ja hem introduït al capítol anterior.

Respecte a la representació del tipus, considerarem tres atributs o camps: un enter pel DNI, un real per la nota i un booleà per definir si l'estudiant té nota o no. Tots ells es declaren privats. Per limitar el rang de notes vàlides introduïm una constant `MAX_NOTA`, que serà declarada `static` per tal que sigui compartida per tots els objectes de tipus `Estudiant`. Ja hem vist que es poden definir operacions `static`, com ara `nota_maxima`, que només tindràn accés als atributs `static` de la classe.

Per a cada classe, definirem propietats per delimitar els possibles valors que els camps poden prendre per representar objectes vàlids de la classe. Aquestes propietats s'han de mantenir en tot moment i es consideren implícites a les pre i les post de les operacions *públiques*. Globalment ens referirem a aquesta informació com *invariant de la representació*. A més, s'ha d'afegir al fitxer `.hh` com a part de la seva documentació.

L'invariant de la representació de la classe `Estudiant`, té en compte la descripció informal de la classe, donada a l'especificació, i l'especificació de les operacions. Per a classes més complexes també intervindran altres consideracions, com ara decisions d'eficiencia.

Noteu que no necessitem el constructor `struct` del C++ per agrupar els atributs. La raó és que aquests ja estan agrupats per formar un únic tipus dins la classe. Sí que és possible que, en altres implementacions, necessitem un `struct` per definir tipus auxiliars dins d'una classe.

```
class Estudiant {

private:
  int dni;
  double nota;
  bool amb_nota;
  static const int MAX_NOTA = 10;
  /*
     Invariant de la representació:
     - 0 <= dni
     - si amb_nota, llavors 0 <= nota <= MAX_NOTA
  */

public:
  /* Constructores */
  Estudiant();
  Estudiant(int dni);

  /* Destructora por defecte */
  ~Estudiant();

  /* Modificadores */
  void afegir_nota(double nota);
  void modificar_nota(double nota);

 /* Consultores */
  int consultar_DNI() const;
```

```
  bool te_nota() const;
  double consultar_nota() const;
  static double nota_maxima();

  /* Entrada / Sortida */
  void llegir();
  void escriure() const;
};
```

Passem ara a l'arxiu `Estudiant.cc`, on codifiquem els mètodes. Al fitxer `.cc` de cada classe, s'ha de precedir el nom de cada operació pel nom de la classe, separat per `::` (en aquest cas `Estudiant::`). D'aquesta manera identifiquem cada operació amb les seva capçalera al corresponent arxiu `.hh` i li donem el dret de veure els camps dels objectes de la classe.

Recordem que si tenim un objecte `est` de la classe `Estudiant`, accedirem al seu camp `dni` amb la notació `est.dni` i el mateix amb els altres camps (com hem vist, es tracta del mateix mecanisme que es fa servir amb les operacions).

En ocasions, al codi dels mètodes serà necessari referir-se al paràmetre implícit d'alguna manera. La paraula que el C++ reserva per aquest propòsit és `this`. Serà obligatori usar-la quan coexisteixin en un mateix bloc de codi alguna variable amb el mateix nom que un atribut. Com a exemple, es pot veure el codi dels mètodes `afegir_nota` i `modificar_nota` en els quals existeix la possibilitat de confondre l'atribut `nota` amb el paràmetre del mateix nom. Si no es dóna aquesta situació, accedirem a cada camp del paràmetre implícit només citant el nom del camp. Per exemple, si escrivim només `dni`, ens estem referint al camp `dni` del paràmetre implícit.

També serà necessari usar el `this` quan es fa referència al paràmetre implícit en el seu conjunt, per exemple, si s'ha passar com a paràmetre no implícit d'alguna operació.

Noteu que dins de la implementació els camps de la classe són visibles i per tant es poden usar al codi, però no a les especificacions de les operacions públiques: a la pre de `modificar_nota` no seria correcte dir `0<= "nota≪=MAX_NOTA`.

```
Estudiant::Estudiant()
/* Pre: cert */
/* Post: el resultat és un estudiant amb DNI=0 i sense nota */
{
    dni = 0;
    amb_nota = false;
}

Estudiant::Estudiant(int dni)
/* Pre: dni>=0 */
/* Post: el resultat és un estudiant amb DNI=dni i sense nota */
{
    this->dni = dni;
    amb_nota = false;
}
```

```
Estudiant::~Estudiant(){}
// esborra automàticament els objectes locals en sortir d'un àmbit de visibilitat

void Estudiant::afegir_nota(double nota)
/* Pre: el paràmetre implícit no té nota i 0<="nota"<=nota_maxima() */
/* Post: la nota del paràmetre implícit passa a ser "nota" */
{
    this->nota = nota;
    /* una notació equivalent alternativa és  (*this).nota = nota; */
    amb_nota = true;
}

void Estudiant::modificar_nota(double nota)
/* Pre: el paràmetre implícit té nota i 0<="nota"<=nota_maxima() */
/* Post: la nota del paràmetre implícit passa a ser "nota" */
{
    this->nota = nota;
}

int Estudiant::consultar_DNI() const
/* Pre: cert  */
/* Post: el resultat és el DNI del paràmetre implícit  */
{
    return dni;
}

double Estudiant::consultar_nota() const
/* Pre: el paràmetre implícit té nota */
/* Post: el resultat és la nota del paràmetre implícit */
{
    return nota;
}

double Estudiant::nota_maxima()
/* Pre: cert */
/* Post: el resultat és la nota màxima dels elements de la classe */
{
    return MAX_NOTA;
}


bool Estudiant::te_nota() const
/* Pre: cert */
/* Post: el resultat indica  si el paràmetre implícit té nota o no */
{
```

```
    return amb_nota;
}

void Estudiant::llegir()
/* Pre: estan preparats al canal estandar d'entrada un enter no negatiu i
   un double */
/* Post: el paràmetre implícit passa a tenir els atributs llegits del canal
   estàndard d'entrada;  si el double no pertany a l'interval [0..nota_maxima()],
     el p.i. es queda sense nota  */
{
    cin >> dni;
    double x;
    cin >> x;
    if (x >= 0 and x <= MAX_NOTA) {
       nota = x;
       amb_nota = true;
    }
    else amb_nota = false;
}

void Estudiant::escriure() const
/* Pre: cert */
/* Post: s'han escrit els atributs del paràmetre implícit al canal estàndard
   de sortida: el dni seguit de un NP o la nota en format double, separats per
   un espai en blanc */
{
    if (amb_nota) cout << dni << " " << nota << endl;
    else cout << dni << " NP" << endl;
}
```

Com a exercici, i per constatar la independència de la implementació, podeu fer una implementació alternativa del tipus `Estudiant`, on en lloc de tenir tres atributs, eliminem l'atribut booleà i usem l'atribut `nota` també per identificar quan l'estudiant no té nota, posant-li algun valor negatiu, per exemple el `-1.0`. Així, si l'atribut és diferent de `-1.0` vol dir que té nota i si és `-1.0` significa que no en té.

És molt important notar que, si canviem la implementació triada sense que afecti a l'especificació, no haurem de revisar els algorismes que usin la classe. Només haurem de revisar els algorismes que l'usen si fem algun canvi en la seva especificació.

2.2.3 La classe `Cjt_estudiants`

En aquest apartat presentarem una implementació de la classe `Cjt_estudiants`. En primer lloc presentem l'arxiu `Cjt_estudiants.hh`. Per limitar la capacitat dels conjunts i poder implementarlos amb un vector, introduïm una constant `MAX_NEST`, que serà declarada `static` per tal que sigui compartida per tots els possibles objectes de tipus `Cjt_estudiants`. Per últim, fem servir el

camp `nest` per guardar el nombre d'elements, o *mida*, del conjunt; delimitem la part del vector que conté els estudiants del conjunt amb l'interval `[0..nest-1]`.

Com que el conjunt ha d'estar ordenat pels DNI dels estudiants, mantindrem el vector ordenat en tot moment. Això penalitza l'eficiència de l'operació `afegir_estudiant` i la de lectura (o bé requereix que els elements es llegeixin ja ordenats), però afavoreix molt les operacions de cerca, ja que podran fer ús de l'esquema de cerca dicotòmica o binària, i la d'escriptura.

També introduïm dues operacions privades, `ordenar_cjt_estudiants` i `cerca_dicot`, que farem servir com a auxiliars de diverses operacions públiques. A la Pre/Post d'una operació privada sí que poden aparèixer d'altres elments privats de la classe, com ara els camps. Normalment, les operacions privades no es detecten a la fase d'especificació i és perfectament vàlid introduir-les a la implementació. Noteu que declarem la segona com a `static` ja que li passem el vector d'`Estudiant` com a paràmetre i, per tant, no li cal el paràmetre implícit.

Com hem fet per a la classe `Estudiant`, no repetim les especificacions de les operacions públiques, que ja hem introduït al capítol anterior. Per tant, només mostrem les Pre/Post de les operacions privades.

```
#include "Estudiant.hh"
#include <vector>

class Cjt_estudiants {

  private:

    vector<Estudiant> vest;
    int nest;
    static const int MAX_NEST = 20;
    /*
       Invariant de la representació:
       - vest[0..nest-1] està ordenat creixentment pels DNI dels estudiants
       - 0 <= nest <= vest.size() = MAX_NEST
    */

    void ordenar_cjt_estudiants(); // només per llegir
    /* Pre: cert */
    /* Post: els elements rellevants del paràmetre implícit estan ordenats
       creixentment pels seus DNI */

    static int cerca_dicot(const vector<Estudiant> &vest, int left, int right, int x)
    /* Pre: vest[left..right] està ordenat creixentment per DNI,
       0<=left, right<vest.size() */
    /* Post: si a vest[left..right] hi ha un element amb DNI = x, el resultat és
         una posició que el conté; si no, el resultat és -1 */

  public:
```

```
    Cjt_estudiants();

    ~Cjt_estudiants();

    void afegir_estudiant(const Estudiant &est);

    void modificar_estudiant(const Estudiant &est);

    void modificar_iessim(int i, const Estudiant &est);

    int mida() const;

    static int mida_maxima();

    Estudiant consultar_estudiant(int dni) const;

    Estudiant consultar_iessim(int i) const;

    void llegir();

    void escriure() const;
};
```

Ara passem a l'arxiu `Cjt_estudiants.cc` on codifiquem els mètodes. Com hem dit, el camp `vest` és un vector d'estudiants ordenat creixentment per DNI, des de 0 fins a `nest-1`. Això, juntament amb la propietat $0 \leq$ `nest` $\leq$ `vest.size() = MAX_NEST`, forma part de manera implícita de les precondicions i postcondicions de totes les operacions.

```
#include "Cjt_estudiants.hh"

Cjt_estudiants::Cjt_estudiants()
/* Pre: cert */
/* Post: el resultat és un conjunt d'estudiants buit */
{
  nest = 0;
  vest = vector<Estudiant>(MAX_NEST);
}

Cjt_estudiants::~Cjt_estudiants(){}

void Cjt_estudiants::afegir_estudiant(const Estudiant &est)
/* Pre: el paràmetre implícit no conté cap estudiant amb el DNI d'est; el
   nombre d'estudiants del p.i. es menor que la mida màxima permesa */
/* Post: s'ha afegit l'estudiant est al paràmetre implícit */
{
  int i = nest-1;
```

```
  bool b = false;
  int dni = est.consultar_DNI();
  while (i >= 0 and not b) {
    if ( dni > vest[i].consultar_DNI()) b = true;
    else {
      vest[i+1]=vest[i];
       --i;
    }
  }
  // i és la posició més avançada amb el DNI més petit que dni, si n'hi ha;
  // si no, i=-1
  vest[i+1] = est;
  ++nest;
}

int Cjt_estudiants::cerca_dicot(const vector<Estudiant> &vest, int left, int right, int x)
/* Pre: vest[left..right] està ordenat creixentment per DNI,
        0<=left, right<vest.size() */
/* Post: si a vest[left..right] hi ha un element amb DNI = x, el resultat és
         una posició que el conté; si no, el resultat és -1 */
{
  int i; bool found=false;
  while (left <= right and not found) {
    i = (left + right)/2;
    if (x < vest[i].consultar_DNI()) right = i - 1;
    else if (x > vest[i].consultar_DNI()) left = i + 1;
    else found = true;
  }
  // si l'element buscat existeix, i es la posició que volem
  if (found) return i;
  else return -1;
}

void Cjt_estudiants::modificar_estudiant(const Estudiant &est)
/* Pre: existeix un estudiant al paràmetre implícit amb el DNI d'est  */
/* Post: l'estudiant del paràmetre implícit amb el DNI d'est
   ha quedat substituït per est */
{
  // per la Pre, segur que trobem el DNI d'est com a DNI d'algun element
  // de vest[0..nest-1]; apliquem-hi la cerca dicotòmica
  int i = cerca_dicot(vest,0,nest-1,est.consultar_DNI());
  // i és la posició amb el DNI d'est
  vest[i] = est;
}
```

```
void Cjt_estudiants::modificar_iessim(int i, const Estudiant &est)
/* Pre: 1 <= i <= nombre d'estudiants del paràmetre implícit, l'element i-èssim
   del p. i. en ordre creixent per DNI conté un estudiant amb el mateix DNI que est */
/* Post: l'estudiant i-èssim del paràmetre implícit ha quedat substituït
   per est */
{
   vest[i - 1]= est;
}

int Cjt_estudiants::mida() const
/* Pre: cert */
/* Post: el resultat és el nombre d'estudiants del paràmetre implícit */
{
  return nest;
}

int Cjt_estudiants::mida_maxima()
/* Pre: cert */
/* Post: el resultat és el nombre maxim d'estudiants que pot arribar
   a tenir el paràmetre implícit */
{
  return MAX_NEST;
}

bool Cjt_estudiants::existeix_estudiant(int dni) const
/* Pre: dni >= 0 */
/* Post: el resultat indica si existeix un estudiant al paràmetre
   implícit amb DNI = dni */
{
  // apliquem la cerca dicotòmica a l'interval [0..nest-1]
  int i = cerca_dicot(vest,0,nest-1,dni);
  return (i!=-1);
}

Estudiant Cjt_estudiants::consultar_estudiant(int dni) const
/* Pre: existeix un estudiant al paràmetre implícit amb DNI = dni */
/* Post: el resultat és l'estudiant amb DNI = dni que conté el
   paràmetre implícit */
{
  // per la Pre, segur que trobem dni com a DNI d'algun element
  // de vest[0..nest-1]; apliquem-hi la cerca dicotòmica
  int i = cerca_dicot(vest,0,nest-1,dni);
  // i és la posició amb DNI = dni
  return vest[i];
}
```

```
Estudiant Cjt_estudiants::consultar_iessim(int i) const
/* Pre: 1 <= i <= nombre d'estudiants que conté el paràmetre implícit */
/* Post: el resultat és l'estudiant i-èssim del paràmetre implícit
   en ordre creixent per DNI */
{
  return vest[i - 1];
}

void Cjt_estudiants::llegir()
/* Pre: estan preparats al canal estàndar d'entrada un enter entre 0 i
   la mida màxima permesa, que representa el nombre d'elements que llegirem,
   i les dades de tal nombre d'estudiants */
/* Post: el paràmetre implícit conté el conjunt d'estudiants llegits
   del canal estàndard d'entrada */
{
  cin >> nest;
  for (int i = 0; i < nest; ++i) vest[i].llegir(); // llegir de la classe Estudiant
  ordenar_cjt_estudiants(); // noteu que l'apliquem sobre el p. i.
}

void Cjt_estudiants::escriure() const
/* Pre: cert */
/* Post: s'han escrit pel canal estàndard de sortida els estudiants del
   paràmetre implícit en ordre ascendent per DNI */
{
  for (int i = 0; i < nest; ++i) vest[i].escriure(); // id. escriure
}
```

Deixem com a exercici la implementació de l'operació privada d'ordenació i una possible millora d'`afegir_estudiant` que faci ús de l'esquema de cerca dicotòmica.

2.2.4 Ampliació de la classe `Cjt_estudiants`

En aquest apartat presentarem implementacions de les dues primeres opcions per ampliar la classe `Cjt_estudiants`. En primer lloc presentarem la segona que, recordem, consistia en definir un enriquiment, és a dir, un mòdul funcional amb les noves operacions que, en aquest cas, no seran mètodes orientats a objecte. Per implementar el mòdul funcional hem de definir un arxiu `E_Cjt_estudiants.cc` amb el següent contingut

```
#include "E_Cjt_estudiants.hh"

void esborrar_estudiant(Cjt_estudiants &Cest, int dni)
/* Pre: existeix un estudiant a Cest amb DNI = dni */
/* Post: Cest conté els mateixos estudiants que el seu valor original
```

```
   menys l'estudiant amb DNI = dni */
{
  Cjt_estudiants Cestaux;
  int i = 1;
  while (dni != Cest.consultar_iessim(i).consultar_DNI()) {
        Cestaux.afegir_estudiant(Cest.consultar_iessim(i));
        ++i;
    }
  // per la pre, segur que trobarem a Cest un estudiant amb DNI = dni;
  // en aquest punt del programa, aquest estudiant és Cest.consultar_iessim(i);
  // ara hem d'afegir els elements següents a Cestaux
  for (int j = i + 1; j <= Cest.mida(); ++j)
      Cestaux.afegir_estudiant(Cest.consultar_iessim(j));

  Cest = Cestaux;
}

Estudiant estudiant_nota_max(const Cjt_estudiants &Cest)
/* Pre: Cest conté almenys un estudiant amb nota */
/* Post: el resultat és l'estudiant de Cest amb nota màxima;
   si en té més d'un, és el de DNI més petit */
{
  int i = 1;
  while (not Cest.consultar_iessim(i).te_nota()) ++i;
  int imax = i; ++i;
  // per la pre, segur que trobarem a Cest un estudiant amb nota;
  // imax n'és el primer; comprovem la resta
  while (i <= Cest.mida()){
    if (Cest.consultar_iessim(i).te_nota())
       if (Cest.consultar_iessim(imax).consultar_nota() <
           Cest.consultar_iessim(i).consultar_nota()) imax = i;
    ++i;
  }
 return Cest.consultar_iessim(imax);
}
```

Noteu que la solució que hem proposat per `esborrar_estudiant` és molt dolenta, ja que hem hagut de construir un nou conjunt amb els estudiants que han de romandre a `Cest` i al final li hem assignat. Això és degut a que, encara que disposem d'operacions per modificar `Cest` directament, cap d'elles serveix per reduir-ne la mida. De seguida mostrarem com, si treballem dins de la classe i podem accedir-ne als camps, la mateixa operació es pot implementar de manera molt més elegant i amb menys instruccions.

Ara veurem el cas de modificar directament l'arxiu `Cjt_estudiants.hh` afegint les capçaleres de les noves operacions i l'arxiu `Cjt_estudiants.cc` amb el codi de les noves operacions. En aquest cas podem aprofitar per fer els canvis que creguem oportuns en la implementació del

tipus per aconseguir una implementació més eficient de les operacions. Com ja vam explicar, la introducció de noves operacions canvia l'especificació de la classe `Cjt_estudiants` i per tant, les seves regles d'ús.

Concretament, suposem que introduïm un nou atribut al `Cjt_estudiants.hh` que ens permeti recordar la posició de l'estudiant amb la nota màxima

```
...
  private:
   vector<Estudiant> vest;
   int nest;
   static const int MAX_NEST = 60;
   int imax;  // Aquest és el nou atribut
...
```

Quant a les noves operacions, ara les podríem implementar de la següent manera dintre del `Cjt_estudiants.cc`

```
...
void Cjt_estudiants::esborrar_estudiant(int dni)
/* Pre: existeix un estudiant al paràmetre implícit amb DNI = dni */
/* Post: el paràmetre implícit conté els mateixos estudiants que
   l'original menys l'estudiant amb DNI = dni */
{
  int i = 0;
  while (dni != vest[i].consultar_DNI())  ++i;

  // per la pre, segur que trobarem a Cest un estudiant amb DNI = dni;
  // en aquest punt del programa, aquest estudiant és vest[i];
  // ara desplacem els elements següents per ocupar el lloc de vest[i],
  // actualitzem la mida i mirem si s'ha d'actualitzar imax

  for (int j = i; j < nest - 1; ++j) vest[j] = vest[j + 1];
  --nest;
  if (i == imax) recalcular_posicio_imax();
  else if (imax > i) --imax;
}

Estudiant Cjt_estudiants::estudiant_nota_max( ) const
/* Pre: el paràmetre implícit conté almenys un estudiant amb nota */
/* Post: el resultat és l'estudiant del paràmetre implícit amb nota màxima;
   si en té més d'un, és el de DNI més petit */
{
 return vest[imax];
}
...
```

Fixem-nos que a la implementació d'`esborrar_estudiant` ens estalviem totes les crides a `afegir_estudiant`, cadascuna de les quals suposava un recorregut del vector, i l'assignació amb el conjunt auxiliar. Respecte a `estudiant_nota_max`, notem que ja no requereix buscar la nota màxima del vector, ja que la tenim a la posició `imax`.

Deixem com exercici l'operació privada `recalcular_posicio_imax` que es crida a `esborrar_estudiant` així com una possible millora d'aquesta que faci ús de l'esquema de cerca dicotòmica.

També s'han d'aplicar tots els canvis que siguin necessaris a la implementació de les operacions públiques a `Cjt_estudiants.cc` per inicialitzar i mantenir actualitzat l'atribut `imax`. S'haurà de considerar un valor especial d'`imax` pel cas que cap estudiant del conjunt no tingui nota.

2.3 Metodologia de disseny modular

El disseny modular es basa en construir una sèrie de mòduls de dades o funcionals i combinar-los per resoldre un problema determinat. Per saber quins mòduls necessitem, primer hem de descompondre el problema en termes de les seves dades (i funcionalitats). Aquestes dades seran abstractes, no seran necessàriament tipus de dades existents, com ara llistes o vectors, sinó que introduirem nous tipus de dades, com ara Estudiant, i les seves operacions associades que creguem necessàries. En particular, totes les operacions que es cridin des del programa principal han de declarar-se com a públiques d'alguna classe.

En una fase posterior, treballarem la seva implementació, que moltes vegades estarà basada en tipus de dades, ara sí, coneguts, com ara llistes, vectors, etc. Per aconseguir una veritable modularitat, és molt important mantenir aquesta idea d'abstracció de dades.

Per exemple, suposem que volem tractar un text analitzant-lo paraula per paraula. Hom podria pensar ràpidament en utilitzar els vectors de caràcters com a tipus de dades per tractar el text. Ara bé, depenent del tractament que vulguem aplicar, pot ser millor considerar llistes de caràcters o una altra estructura. És a dir, nosaltres sabem que necessitem "paraules" i certes operacions sobre elles, i que les podem implementar a posteriori amb vectors, llistes, etc. Aquest és el fet que ha de guiar el nostre disseny modular. Hem de detectar quins tipus de dades necessitem, com ara `Paraula`, i dotar-los de les operacions que considerem necessàries per resoldre el nostre problema.

Per suposat, els tipus nous poden estar relacionats entre ells: per exemple podem necessitar el tipus `Grup_de_paraules` i el tipus `Paraula`, cadascun amb les seves operacions. També és possible que inicialment hàgim proposat alguna operació que finalment no sigui necessària, o bé, que ens n'hàgim deixat alguna. Tot això ho detectarem a mesura que avanci el disseny. En aquest sentit, l'estratègia de disseny es pot veure com una contínua "negociació"entre el que estem dissenyant i el que encara no hem dissenyat.

Per exemple, si féssim un disseny descendent "pur", proposaríem l'algorisme de més alt nivell (o, si el cas, el programa principal) sense haver especificat encara cap mòdul. Els mòduls ens apareixerien "a petició" de l'algorisme que estiguéssim dissenyant. Si féssim un disseny ascendent "pur", primer proposaríem uns mòduls concrets i després intentaríem usar-los a l'algorisme.

En ambdós casos, una mala decisió sobre el que estem dissenyant podria perjudicar el que encara no ha estat dissenyat.

La solució passa per fer un procés iteratiu que incrementalment vagi refinant els diferents mòduls de l'aplicació i que eviti certes decisions prematures en el disseny. Per exemple, primer podem tenir una certa intuïció sobre quines coses farà l'algorisme principal (fem una mica de disseny descendent). A continuació proposem uns mòduls (principalment de dades) amb operacions que ens ajudin a refinar aquesta idea intuïtiva de l'algorisme que volem dissenyar (fem una mica de disseny ascendent). A continuació refinem l'algorisme. Pot passar que:

- Necessitem alguna operació en algun mòdul que encara no hem definit. Cap problema, l'afegim.
- Hi ha alguna altra operació que no necessitem. Cap problema, la traiem.
- Veiem que la descomposició en mòduls proposada no és la més adequada. Cap problema, la canviem i tornem a refinar l'algorisme inicial.

Ara repetiríem el procés per refinar els mòduls. En general, és millor refinar primer els mòduls que són més amunt en la jerarquia de mòduls que hem creat (els més propers al programa principal). Refinar un mòdul requereix implementar el tipus i les seves operacions. Per fer això, novament establiríem primer de manera aproximada l'estratègia de resolució de les operacions, a la vegada que pensaríem la millor implementació del tipus que ens ajudi a aconseguir la millor implementació de les operacions. Novament aquest procés de refinament ens mostrarà la necessitat d'afegir noves operacions a algun mòdul o la necessitat de crear un nou tipus. També ens pot permetre detectar operacions innecessàries.

Un dubte molt típic en aquest punt del disseny sorgeix a l'hora de decidir a quin mòdul s'ha de definir una certa operació. Noteu que si l'operació només té paràmetres d'una classe la decisió és immediata, però si té paràmetres de més d'una classe, pot no estar clar a quina d'elles ha de pertànyer. Encara que no sempre existeix una única opció vàlida, hi ha alguns criteris que ens poden ajudar a prendre la decisió. Un d'ells és esbrinar quina de les classes és la més "afectada" per l'operació. Per exemple, si un dels paràmetres s'ha de modificar i els altres només es consulten, el més normal és que l'operació pertany a la classe del primer. Un altre es basa en analitzar quines modificacions de les altres classes comportaria el fet de definir l'operació en una de elles, per exemple respecte a la necessitat de definir operacions noves d'aquestes. L'opció més interessant seria la que menys impacte tingués sobre el conjunt de les classes.

En qualsevol cas, com que en tot moment estem usant els nous mòduls de forma independent de la seva implementació, qualsevol canvi intern en el refinament d'un mòdul no tindrà cap efecte en l'ús d'aquest mòdul en altres parts del disseny. Per tant, només hem de revisar la feina ja feta en l'algorisme principal o en els mòduls si canviem els paràmetres o l'especificació d'alguna operació visible (és a dir, les que hem posat a l'especificació del mòdul) o si movem alguna operació d'un mòdul a un altre.

Finalment, teniu en compte que, per decidir quins mòduls de dades cal crear, hem de pensar en quin tipus d'informació hem de manegar i, principalment, en quines operacions ens interessa tenir. Noteu que l'essència d'un tipus no només ens la diu el seu nom, o la informació que

conté, sinó també les seves operacions. Per exemple, les diferències entre una pila i una cua ens les mostren les seves operacions. El que no s'ha de fer mai en el moment de proposar els nous mòduls de dades és pensar en com emmagatzemarem la informació (és a dir, com implementarem els tipus), ja que això com hem dit dependrà molt de com implementem les operacions i, per tant, en cas que ho féssim aixi, prendríem moltes decisions arbitràries (basades en una anàlisi molt poc acurada del que ens cal) que amb alta probabilitat hauríem de canviar a posteriori. A més, com que volem assolir la independència de la implementació, decidir aquesta en un moment tan prematur no ens ajudarà en res (i de fet, és possible que ens dificulti la feina).

2.3.1 Resum d'elements metodològics

Encara que el lloc on es treballarà el disseny modular és el laboratori, concloem aquest capítol amb un seguit d'elements pràctics, derivats del contingut que hem presentat, que aplicarem extensivament als exemples de les sessions.

Quan ens donen l'enunciat d'un problema nou, podem abordar-lo en cinc fases:

1. Detectar les classes de dades implicades en el nostre problema, a partir de l'enunciat
2. Obtenir un esquema preliminar per al programa principal
3. Especificar les classes detectades que no puguem reusar d'alguna situació anterior
4. Escriure el programa principal amb detall, fent servir objectes de les classes especificades i instrucciones reals
5. Implementar les classes noves

Les fases 3 i 5 poden repetir-se durant la implementació, ja que poden aparèixer noves classes o operacions auxiliars que no eren evidents al principi del disseny o bé alguna operació pot canviar de classe o desaparèixer.

La relació entre els mòduls ha de poder representar-se gràficament amb un diagrama modular sense cicles. Els cicles normalment són simptoma d'operacions mal definides o que pertanyen a classes on no haurien d'estar.

Les operacions d'una classe que s'hagin de fer servir fora d'aquesta classe (és a dir, a d'altres classes o al programa principal) han de declarar-se públiques. Les operacions d'una classe que només es facin servir dins de la pròpia classe han de declarar-se privades. Les operacions d'una classe que no necessitin un paràmetre implícit han de declarar-se `static`.

Les operacions públiques d'una classe només haurien de tenir paràmetres de tipus simples o objectes de la classe o d'altres classes, però no contenidors estandards com ara vectors, llistes, etc. Si en algun moment ens plantegem la necessitat d'aquests, probablement és simptoma de que el nostre disseny requereix classes noves per representar aquesta informació.

Si hem decidit que necessitem una operació però no estem segurs sobre a quina classe ha de pertànyer, hem de tenir en compte quina és la classe més afectada per l'operació i ponderar l'impacte que la pertinença a una de les classes tindria sobre les altres.

Les dades de la representació d'una classe (els camps) han de declarar-se privats i quan s'usen fora de la classe només es poden consultar o modificar a través d'operacions de la classe.

Referències:

[Bal93] Balcázar, José Luis, *Programación Metódica*, McGraw-Hill, 1993
[Pe05] Peña, Ricardo. *Diseño de programas: formalismo y abstracción*, Prentice Hall, 2005