

Programació Orientada a l'Objecte

Pràctica 1

Indicacions:

Raoneu i justifiqueu totes les respostes.

Per a dubtes i aclariments sobre l'enunciat, adreceu-vos al consultor responsable de la vostra aula de teoria. Per a dubtes sobre codificació, adreceu-vos al consultor responsable de la vostra aula de laboratori.

Recordeu que si convalideu la pràctica per la nota de pràctiques del semestre anterior, és igualment important que entengueu amb detall l'enunciat i la solució de la present pràctica, ja que l'examen final contindrà preguntes específiques sobre aquesta pràctica per a tots els estudiants.

Lliurament:

1. Cal lliurar la solució en un arxiu anomenat `CognomsNom_POO_Practical1`.
2. Si l'entrega inclou múltiples documents (per exemple, fitxers font), es lliuraran tots comprimits en un únic fitxer ZIP.
3. En cas d'incloure fitxers de codi font (C++), incloeu-los en un directori de nom `/src` dins el ZIP.
4. En cas d'incloure documentació de classes (doxygen), incloeu-la en un directori de nom `/doc` dins el ZIP.
5. Data límit per lliurar la solució: dimecres, 16 de novembre de 2011 (a les 23:59 hores).

És imprescindible respectar el format i data d'entrega. La no adequació a aquestes especificacions pot suposar la no avaluació de la Pràctica.

Índex

1. Introducció	3
2. Herència i polimorfisme (30%)	4
3. Relacions entre classes (35%).....	8
4. Tractament d'errors per excepcions (35%)	13
5. Criteris d'avaluació.....	15

1. Introducció

A aquesta pràctica treballarem alguns dels mecanismes fonamentals de la Programació Orientada a l'Objecte, com són:

- **Herència i Polimorfisme.** Aquests dos mecanismes caracteritzen de manera exclusiva la Programació Orientada a l'Objecte.
- **Les associacions entre classes.**
- Tractament d'errors mitjançant **excepcions.** Aquest mecanisme no és exclusiu de la Programació Orientada a l'Objecte, no obstant això, a la Programació Orientada a l'Objecte té una especial rellevància per la seva contribució a la encapsulació i ocultació dels detalls de la implementació.

Així doncs, amb aquesta pràctica es pretén treballar alguns dels conceptes i mecanismes que haureu d'integrar en el desenvolupament posterior de la pràctica 2, junt amb d'altres aspectes treballats a les PACs.

L'arxiu .zip que us hem proporcionat té la següent estructura de carpetes:

- Exercise 1
- Exercise 2
- Exercise 3

A cada carpeta teniu el material didàctic de partida per cada apartat de l'exercici corresponent. Tot el treball a realitzar està guiat a l'enunciat de l'exercici seguint aquesta mateixa estructura. Us recomanem que llegiu l'enunciat sencer de cada exercici abans de començar a treballar.

En quant al format de lliurament, heu de seguir les instruccions generals indicades al requadre explicatiu que teniu a l'inici d'aquest enunciat. En particular, noteu que tot el material l'heu de lliurar en un únic .zip. No obstant això, al final de cada exercici hi ha un apartat que us indica la manera de lliurar-lo per tal d'estructurar adequadament l'**ARXIU ZIP ÚNIC** que constitueix el lliurament global de la pràctica.

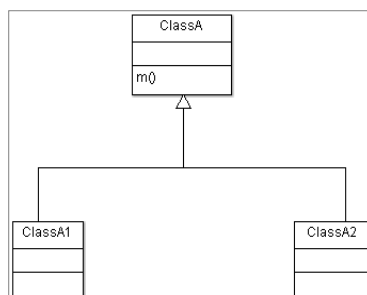
2. Herència i polimorfisme (30%)

Per realitzar aquest exercici és convenient revisar el mòdul 5 dels apunts de teoria.

El treball està guiat, de manera molt explícita, mitjançant passos. Heu de lliurar el resultat de cada exercici en una carpeta separada i totes les carpetes comprimides en un únic arxiu .zip.

Step 0

Començarem treballant el mecanisme de la herència. Tal com ja heu estudiat als mòduls de teoria, la herència és un mecanisme que ens permet implementar un tipus de relació entre classes: generalització ↔ especialització. Aquesta relació es descriu amb UML de la manera següent:



Aquesta especificació estableix que les classes ClassA1 i ClassA2 hereten (són una especialització) de la classe ClassA (que és una generalització de ClassA1 i ClassA2).

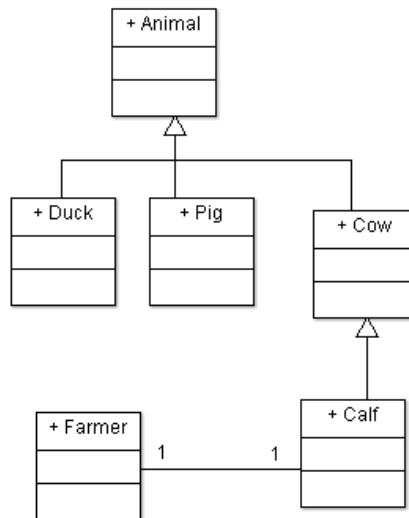
L'herència crea una relació de **tipus** entre la *super-classe* i les seves classes hereves. A l'exemple anterior, ClassA1 i ClassA2 **són-un** ClassA. Això vol dir que els objectes creats com a instàncies de les classes ClassA1 i ClassA2 són de **tipus** ClassA1 i ClassA2 respectivament i, **a més**, són de **tipus** ClassA. Aquesta relació de **tipus** és una de las bases del polimorfisme.

Les classes ClassA1 i ClassA2 hereten el comportament (atributs i mètodes) declarat a la classe ClassA i són de tipus ClassA. Així, per exemple, les classes ClassA1 i ClassA2 poden utilitzar com propi el mètode m(), que ha estat declarat a la super-classe ClassA (per això no l'hem inclòs explícitament a las classes ClassA1 i ClassA2).

L'exemple el podeu crear com a un projecte Eclipse i fer que ClassA1 i ClassA2 disposin d'un mètode m(). Aquest mètode simplement pot escriure un missatge de l'estil 'soc ClasseA1' o bé 'soc ClasseA2'. El podeu provar afegint un mètode `main`, que creï diferents objectes.

En aquesta pràctica treballarem aquests conceptes a partir de la següent jerarquia de classes que correspon a una part de la jerarquia del regne animal:

A la carpeta Exercise1 del l'arxiu de l'enunciat teniu els fitxers C++ corresponents a quatre classes següents (no heu de modificar cap d'ells). Noteu com, mitjançant l'ús de l'operador `::` s'especifiquen les diferents relacions de herència que mostra la figura.



Les classes `Cow`, `Duck` i `Pig` representen un membre dels animals. Aquestes classes hereten tot el comportament definit a la classe `Animal`.

Per exemple, intenteu comentar el codi `Animal(id, name);` de la classe `Pig` i analitzeu l'error que apareix.

Per comprovar el que acabem d'explicar, a la carpeta `Exercise1` teniu el fitxer de prova `Step0.cpp` que crea un objecte de cadascuna de les tres classes de la jerarquia de membres de l'equip que ens permeten crear instàncies: `Cow`, `Duck` i `Pig`. Llavors, heu de fer el següent:

1. Entendre el `Step0.cpp`
2. Compileu totes les classes que hi ha a la carpeta `Exercise1` relacionades amb `Step0`.
3. Executeu el programa.

El resultat que obtindreu és el que es mostra al fitxer `Step0.txt` que teniu a la mateixa carpeta.

Analitzant la execució d'aquest programa heu d'arribar a les conclusions que hem explicat als últims paràgrafs sobre el funcionament del mecanisme de la herència.

Step1

Continuem treballant el mecanisme de la herència i ara comença la vostra feina pròpiament dita.

El que farem ara és treballar com es pot estructurar la creació de l'espai (atributs i mètodes) d'una certa classe pertanyent a una jerarquia de classes fent referència a mètodes de la seva super-classe, mitjançant la clàusula `::`.

A la carpeta `Exercise1` de l'arxiu de l'enunciat teniu els següents fitxers de codi font de classes:

- `Step1.cpp`
- classe `Calf`

A continuació es mostra part del fitxer `Step1.cpp`:

```

cout << "-- Create object Cow --" << endl;
Animal *c1 = new Cow(1001, "Boby", 12);

cout << "-- Create object Pig --" << endl;
Pig *c2 = new Pig(1002, "Dana", true);
  
```

```

cout << "-- Create object Duck --" << endl;
Animal *c3 = new Duck(1003, "Jack", false);

// Create Calf object
cout << "-- Create object Calf --" << endl;
Cow *p1 = new Calf(2001, "De la Rosa", 10);

cout << c1->str() << endl;
cout << c2->str() << endl;
cout << c3->str() << endl;
cout << p1->str() << endl;

```

Les sentències destacades en groc són crides als constructors de les diferents classes. Així, la primera pretén crear un objecte de tipus `Cow` amb els següents valors als seus atributs:

- id: 1001
- name: "Boby"
- milkperday: 12

La segona pretén crear un objecte de tipus `Pig` amb els següents valors als seus atributs::

- id: 1002
- name: "Dana"
- vaccine: true

La tercera pretén crear un objecte de tipus `Duck` amb els següents valors als seus atributs:

- id: 1003
- name: "Jack"
- liver: false

Les sentències destacades en blau són crides als mètodes `str()` dels diferents objectes creats, per mostrar per pantalla els valors dels seus atributs.

Per fer funcionar el program del pas 1 heu de codificar la classe `Calf`. En relació al pas 0, heu de notar:

- La manera en que el constructor de la subclasse `Cow`, `Duck` o `Pig` crida al constructor de la seva super-classe `Animal`, mitjançant: `Animal(id, name)`.
- La manera en que el mètode `str()` de la subclasse `Cow` crida al de la seva super-classe `Animal`, mitjançant `Animal::str()`.

A partir d'aquí, el que heu de fer vosaltres és el següent:

1. Codificar la classe `Calf` seguint la mateixa idea mostrada per `Cow`, `Duck` i `Pig`.
2. Compilar les classes
3. Executar el programa

El resultat que heu d'obtenir és el que es mostra al fitxer `Step1.txt` que teniu a la mateixa carpeta.

Step 2

Ara ens centrarem en el mecanisme del polimorfisme. Per realitzar aquest exercici és convenient revisar el capítol 4 (Polimorfisme) del mòdul 5 dels apunts de teoria. De fet, ja l'heu estat utilitzant amb el mètode `str` que s'està re-definint a les diferents classes que us hem proporcionat.

Siguin, per exemple, dues classes `ClassA1` i `ClassA2` hereves d'una certa classe `ClassA`. Podem dir que, en quant al comportament heretat de `ClassA`, les tres classes tenen una mateixa "forma". El **polimorfisme** s'obté utilitzant el mecanisme de **re-definició** (**overriding** en anglès): literalment consisteix en re-definir, és a dir, re-codificar el comportament heretat d'acord a les necessitats d'especialització de la classe hereva.

Per treballar aquestes idees utilitzarem la classe `Farmer` (Granger). Heu de modificar la classe `Farmer` re-definint el mètode `reportMilkPerDay` per que retorni una cadena que contingui la mateixa informació que a la classe `Cow` i, a més, el resultat del `str()` de la classe `Farmer` si el `Calf` (Vedell) està assignat o "*Calf assignment is pending*" si no està assignat a un `Farmer`. Noteu que, de fet, aquesta mateixa idea ja l'havíem treballada al pas 1, en relació als mètodes `str()` de les classes, que tenien també codificacions diferents. El mètode `str` constitueix un exemple molt habitual de polimorfisme.

Per treballar aquest concepte, el fitxer `Step2.cpp` conté un petit programa de prova. Noteu que aquest programa fa el següent:

1. Crea els mateixos objectes que ja vàrem veure al pas 1 (afegint un objecte `Farmer`), però ara els insereix en una matriu d'objectes de tipus `Animal` anomenat `aAnimal[]`.
2. Afegeix a la matriu un objecte `Calf` que té associat un `Farmer`.
3. Fa un recorregut dels elements de la matriu `aAnimal[]` i per cadascú d'ells mostra els valors dels seus atributs, mitjançant el mètode `str()`.

Així, a cada iteració del bucle `for` que accedeix als elements (objectes `Animal`) de la matriu, no es pot saber a priori a quina classe concreta (`Cow`, `Duck`, `Pig`, etc.) pertany l'objecte en qüestió. Dit d'una altra manera, és en temps d'execució quan s'ha de decidir quin mètode `reportMilkPerDay` (i també quin mètode `str`) concret s'ha d'executar. Aquesta és la característica fonamental del polimorfisme.

A partir d'aquí, el que heu de fer vosaltres és el següent:

1. Partint del treball del pas 1, modifiqueu la classe `Calf`, re-definint el mètode `reportMilkPerDay` i afegint un atribut de tipus punter a `Farm`.
2. Compileu totes les classes.
3. Executeu el programa.

El resultat que s'ha d'obtenir és el que es mostra al fitxer `Step2.txt` que teniu a la mateixa carpeta.

Exercici 1 - Lliurament

El lliurament consisteix en els fitxers font corresponents al pas 2, amb totes les classes i mètodes degudament comentats. Si us quedeu al pas 1, heu de indicar-ho al fitxer de documentació que, en aquest cas, acompanyarà el lliurament de la pràctica. Els fitxers font els heu d'incloure en una carpeta anomenada `Exercise1` dins de l'ARXIU ZIP ÚNIC que constitueix el lliurament global de la pràctica

3. Relacions entre classes (35%)

Per realitzar aquest exercici es convenient revisar el capítol 3 del mòdul 3 (que explica les relacions entre classes) i el mòdul 4 dels apunts de teoria (que explica els recorreguts).

Explicació teòrica de les relacions

Per poder descriure un problema real no és suficient amb definir classes i crear objectes, també és necessari crear unes relacions entre aquestes classes, que representen la realitat. Les relacions entre les classes permeten definir:

- La **multiplicitat**: número d'objectes de cada classe involucrats.
- La **direccionalitat** o **navegabilitat**: quina classe coneix a l'altra.
- El **rol**: no modifica el comportament però ens permet definir els noms dels integrants.

Complementant els apunts de teoria, es proporciona un exemple de relació i la seva implementació en C++. Es tracta d'una relació **bidireccional** amb multiplicitat **un a molts**. El diagrama de esta relació seria:



El diagrama ens indica que:

- Cada objecte de tipus `ClassA` pot conèixer i estar relacionat amb molts objectes de tipus `ClassB`, però també pot ser que no estigues relacionat amb cap (**0..***).
- Cada objecte de tipus `ClassB` està relacionat amb 1 i només 1 objecte de tipus `ClassA`.
- L'objecte de tipus `ClassA` que forma part de la relació s'anomena `theAclass`.
- Els objectes de tipus `ClassB` que formen part de la relació s'anomenen `bes`.

Fixeu-vos que al diagrama de classes anterior, els atributs privats `theAclass` i `bes` no apareixen especificats com a tals. La pròpia relació ens indica implícitament que faran falta i, per tant, no s'han d'afegir a la llista d'atributs que es mostra a cadascuna de les classes.

La codificació d'aquestes dues classes es farà de la següent manera:

`ClassA` ha de tenir un atribut privat de tipus `vector<ClassB *>` per emmagatzemar el objectes de tipus `ClassB`. . Aquest atribut s'anomenarà `bes`.

- `ClassA` ha de tenir un mètode que s'anomenarà `bool addBes (ClassB *o)` que s'utilitzarà per afegir els objectes de tipus `ClassB` al vector (retorna `true` si l'objecte s'ha afegit correctament). També pot tenir mètodes per esborrar o llistar aquests objectes.
- `ClassB` ha de tenir un atribut privat de tipus `ClassA*` per assignar l'objecte amb el qual està relacionat. Aquest atribut ha d'anomenar-se `theAclass`.
- `ClassB` ha de tenir un mètode que s'anomenarà `void setTheAclass(ClassA *o)` i que s'utilitzarà per assignar l'objecte de tipus `ClassA` a la variable privada `theAclass`. Fixeu-vos que en el cas anterior el

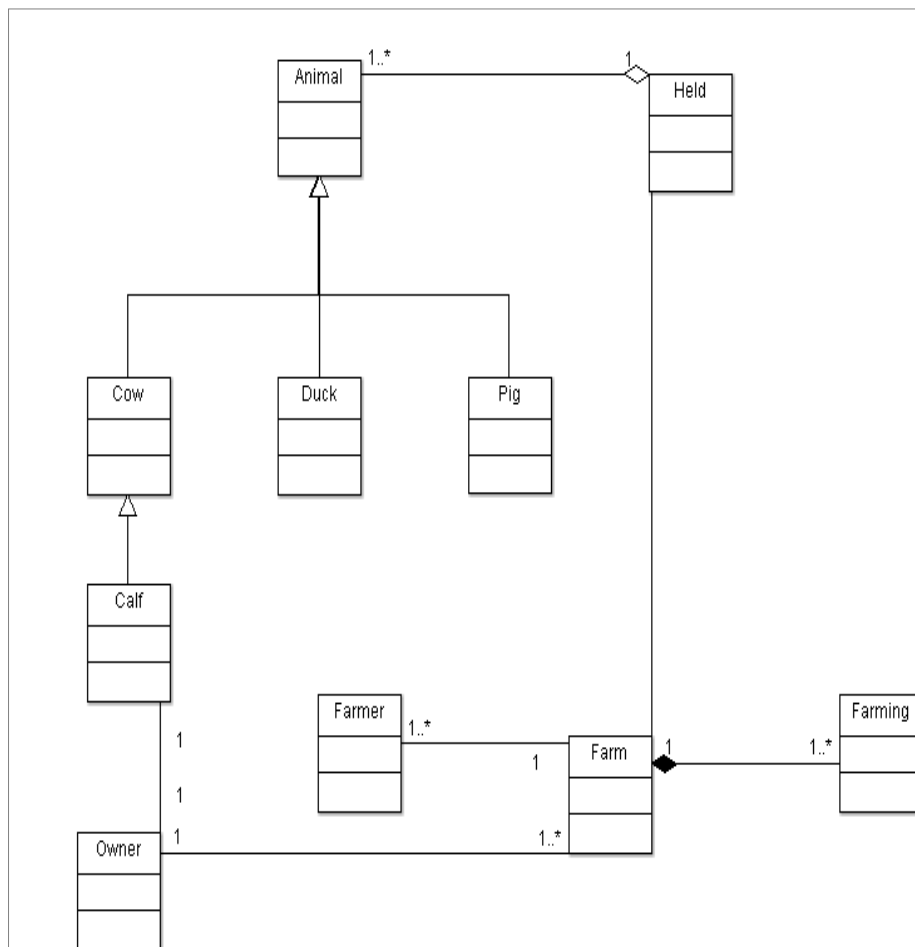
mètode comença per **add**, ja que es poden afegir molts objectes, i en aquest cas per **set**, que ja ens indica que és només per assignar un únic objecte.

Step 0

La classe `Farm` (Granja) representa un terreny rural en el que s'exerceix l'agricultura o la cria de bestiar. La classe `Farming` representa les terres de les quals disposa la `Farm`. La classe `Farmer` representa cadascú dels grangers que treballen en una granja. En aquest cas existeix la classe `Owner` que indica que el propietari d'una granja és únic.

Comentant les relacions: es té una relació de composició (tipus particular d'agregació) entre les classes `Farm` i `Farming`. Escollint aquest tipus de relació es vol indicar que les terres són parts integrants d'una granja i tenen sentit com a tals i no de manera individual. La relació entre els animals de la granja (`Animal`) i el remat que conformen (`Herd`) es pot entendre com una agregació. En canvi, la relació entre `Farm` i `Farmer` s'ha modelat com una associació "d'un a molts", entenent que no existeix el significat de pertinença anterior (s'entén que un granger pot existir sense que estigui en una granja). Noteu que la diferència entre aquests tres tipus de relacions és purament conceptual atès que, des del punt de vista d'implementació, impliquen exactament el mateix, la classe `Farm` ha de tenir una llista o col·lecció d'objectes `Farmer`, `Farming` i `Herd` i un objecte `Owner`. És a dir una granja està composta per un conjunt de grangers, terres, ramats i un propietari. S'ha de notar també que, en funció de la navegabilitat definida en el diagrama, així com `Herd` conté com a atribut una col·lecció d'objectes `Animal`, i `Animal` no conté cap atribut de tipus `Herd`.

El diagrama és el següent:



A la carpeta `Exercise2Step01` de l'arxiu de l'enunciat podreu algunes de les classes anteriors codificades menys la classe `Farm`. Familiaritzeu-vos amb la seva codificació compilant i executant `Step0.cpp`. Per exemple, veureu que a la classe `Herd` hi ha els següents mètodes:

- public bool **addAnimal** (Animal *animal) - Afegeix un objecte `Animal` a la seva llista corresponent.
- vector<Animal*> getAnimals() - Retorna els animals que conformen un ramat (`Herd`).

La classe `Farm` ha de contenir els següents mètodes:

public bool addFarmer(Farmer *farmer) - Afegeix un objecte `Farmer` a la seva llista corresponent.

public bool addFarming(Farming *farming) - Afegeix un objecte `Farming` a la seva llista corresponent.

public bool addHerd(Herd *herd) - Afegeix un objecte `Herd` a la seva llista corresponent.

Heu de veure que el programa `Step0` fa el següent:

- Crea objectes de tipus `Animal` (`Cow`, `Pig`, `Dog`, `Calf`).
- Crea un `Owner`.
- Crea diferents `Farms`, `Farmings` i `Farmers`, relacionant-los.
- Llista els objectes `Farms`.

La sortida que genera el programa `Step0.cpp` la teniu a l'arxiu `Step0.txt`. Observeu que es controla amb una sentència condicional que no hi hagi elements repetits a la llista de `Farmings` de una `Farm`. Al tercer exercici introduïrem una manera alternativa d'implementar mecanismes de control d'errors.

Step 1

En aquest pas heu de codificar la classe `Herd` per implementar les relacions amb `Animal` i `Farm` tal i com s'ha descrit al paràgraf anterior. A més dels atributs i accessoris necessaris, la classe ha d'implementar els següents mètodes:

- public bool **addAnimal**(Animal *animal) - Afegeix un objecte de tipus `Animal` a la seva llista corresponent.
- public bool **setFarm**(Farm *farm) - Afegeix un objecte de tipus `Farm`.

El mètode `addAnimal` indica si la inserció s'ha realitzat correctament.

A la carpeta `Exercise2Step01` teniu tot el material de partida per aquesta part de l'exercici (el mateix que el pas anterior).

Un cop fet això, compileu totes les classes i executeu el programa de prova: `Step1.cpp`. El resultat esperat el teniu al fitxer `Step1.txt`.

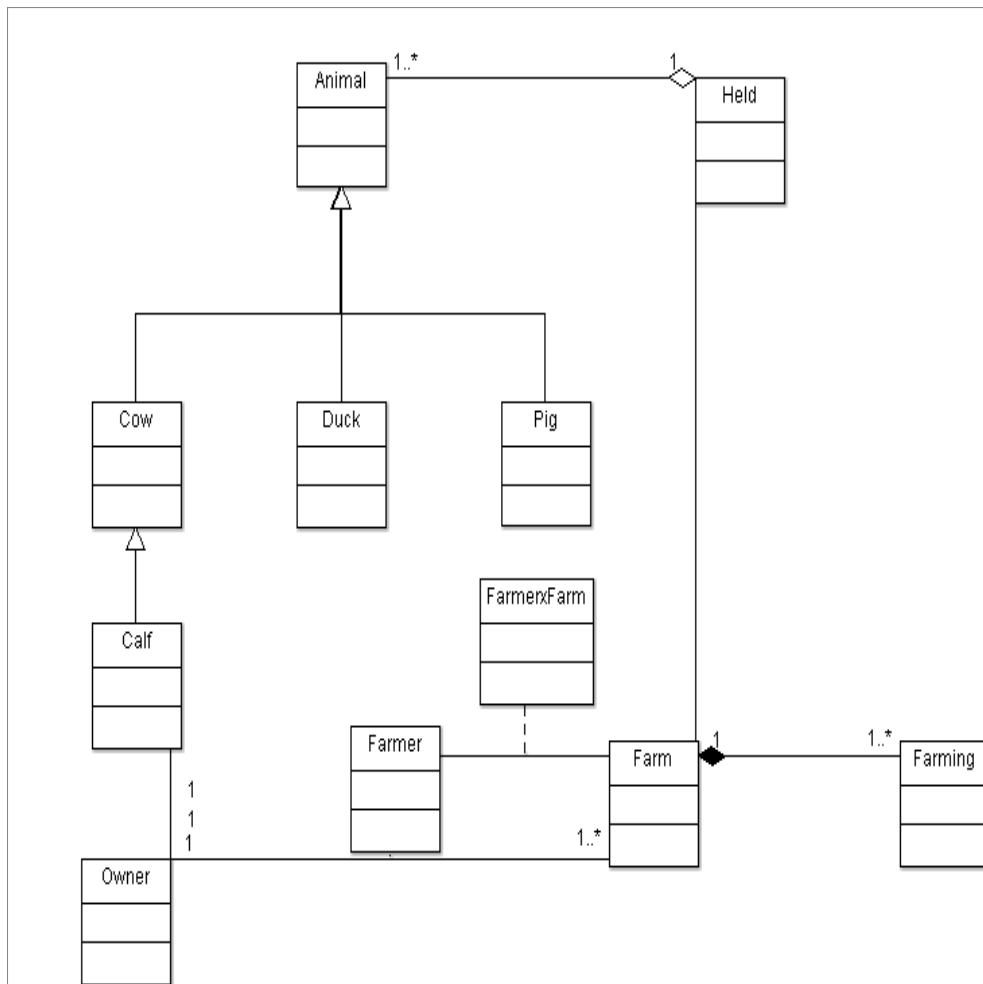
Step 2

En aquest apartat treballarem la implementació de les classes associatives. Si us fixeu en la estructura de classes que hem utilitzat fins ara, l'associació entre `Farm` i `Farmer` només permet relacionar un `Farmer` amb una única `Farm`. Si un `Farmer` vol treballar en una segona `Farm`, perdrem les dades de la primera. Per tant, no es pot desar un històric de les `Farms` en les quals han participat els `Farmers`, per mantenir estadístiques del que han treballat a cada `Farm`.

Per no complicar-lo gaire, ara es vol registrar el nombre d'hores realitzades per un `Farmer` a cada `Farm`. Així, les estadístiques vindran determinades per la següent informació:

- Les hores treballades a una `Farm` per cada `Farmer`.
- Les hores treballades d'un `Farmer` per cadascuna de les `Farms` que ha treballat.

Per aconseguir-ho, s'ha de modificar el diagrama de classes que podria quedar com mostra la figura següent:



A partir d'aquí, es completarà la funcionalitat que es vol donar al programa. Ens interessa:

- Generar un llistat amb les estadístiques (hores treballades) pels `Farmers` que han treballat en una `Farm`.

- Generar un llistat amb les estadístiques (hores treballades) per un `Farmer` a les `Farms` en les que ha treballat.

Per aconseguir-ho, en relació a la versió del pas 1, s'hauran de fer les modificacions sobre las classes que s'indiquen a continuació. En aquest cas, la explicació no és tan detallada como en els anteriors. Els passos a realitzar són el següents:

- Codifiqueu la classe `FarmerxFarm`. Classe associativa entre `Farmer` i `Farm` que desa les hores treballades per un `Farmer` a una `Farm`.
- A la classe `Farm`, incloure els atributs nous necessaris i els mètodes següents:

```
o public void end()
```

Aquest mètode representa una granja que tanca. Ha de fer el següent:

- Recórrer els grangers associats a ella.
- Per cadascú d'ells, cridar al seu mètode `end`. Passant un objecte de tipus `FarmerxFarm`.
- Finalment, afegir l'objecte de tipus `FarmerxFarm` a les estadístiques de la `Farm`.

```
o public string listFarmerStatistics ()
```

Aquest mètode retorna una llista amb les estadístiques dels grangers que han treballat.

- A la classe `Farmer`, incloure els atributs nous necessaris i els mètodes següents:

```
o public void end(FarmerxFarm *farmerxfarm)
```

Afegir l'objecte de tipus `FarmerxFarm`.

```
o public bool updateHours(Farm *farm, int num_hours)
```

Actualitza les hores treballades por un granger en una granja concreta.

```
o public string listFarmerStatistics()
```

Aquest mètode retorna una llista amb les estadístiques dels grangers que hi han treballat.

A la carpeta `Exercise2` teniu el material didàctic que constitueix el punt de partida per aquest últim apartat del exercici. Penseu que teniu que partir de la versió final de les classes que heu fet al pas 1 **sense que aquest deixi de funcionar**.

L'arxiu `Step2.cpp` constitueix el programa de prova de l'exercici. El que fa, concretament, és el següent:

- Crea els mateixos objectes que els dos passos anteriors.
- Tanca dues granges.
- Actualitza les hores treballades dels grangers a les granges tancades.
- Llista totes les estadístiques de les granges.
- Llista totes les estadístiques dels grangers.

Un cop efectuat el treball descrit, compileu totes les classes i executeu el programa `Step2`. El resultat esperat el teniu al fitxer `Step2.out`.

4. Tractament d'errors per excepcions (35%)

Per realitzar aquest exercici és convenient revisar l'apartat 2.4 del Mòdul 7 (El llenguatge de programació C++) dels materials.

Explicació teòrica de les excepcions

Molts llenguatges de programació incorporen el mecanisme de gestió d'excepcions per tractar els errors que poden produir-se en temps d'execució. Es considera una excepció aquell error que provoca que l'aplicació deixi de funcionar correctament, parcial o totalment. Un exemple habitual és la divisió entre zero: si en el moment de realitzar la operació de divisió el divisor és zero, l'error aritmètic que es produeix fa que, de no ésser tractat, l'aplicació interrompi la seva execució.

Per evitar aquestes situacions, a vegades es pot comprovar que es compleixen totes les precondicions necessàries per que una determinada operació finalitzi amb èxit (e.g. comprovant que el divisor sigui diferent de zero abans d'efectuar la divisió). Però fer això en tot el codi font no sempre serà possible (i menys quan fem codi font que d'altres re-utilitzaran) i, en qualsevol cas, donaria lloc a un codi font difícil d'interpretar i mantenir (ple de sentències condicionals prèvies a l'execució de les operacions susceptibles de provocar errors greus o irrecuperables).

L'alternativa està en l'ús dels mecanismes de gestió d'excepcions que C++ (como la majoria de llenguatges d'alt nivell) proporciona. Aquest mecanisme consta de tres elements:

Element	Descripció
Excepció	Objecte que defineix el problema. Quan detecta que alguna cosa va malament, es crea un nou objecte Exception (o normalment una classe hereva) amb la paraula clau new
Llençar l'excepció	Si no es pot o no es vol tractar l'error, llancem l'excepció cap a capes superiors del programa: aquelles que han cridat al mètode que finalment ha produït l'excepció. Això provoca que es detingui momentàniament l'execució del programa i que es busqui qui pot tractar aquesta excepció. Quan es troba un lloc del programa on es vol capturar la excepció, l'execució continua en aquell punt
Capturar l'excepció	<p>Quan una part del programa pot provocar un error i llençar una excepció, es pot usar l'estructura <code>try/catch</code> per intentar capturar l'excepció, tractar-la correctament i continuar amb l'execució del programa.</p> <p>Tractar una excepció podria consistir simplement en informar a l'usuari de que l'operació no s'ha realitzat amb èxit.</p>

Per capturar una excepció en C++ s'utilitza la següent estructura:

```
try {  
    // part of code that may fail  
    // method calls?  
    // mathematic operations?  
    // file input/output?  
} catch (ExceptionName e) {  
    // Code execute if ExceptionName is caught  
}
```

Si quan s'executa la part del programa que es troba dins del `try { ... }` no es produeix cap excepció, el bloc que hi ha dins del `catch { ... }` no s'executa. En cas contrari, la part de codi que hi ha en el `try { ... }` interromp la seva execució en el punt on s'ha produït l'excepció i aquesta es continua a la primera línia del bloc `catch { ... }`.

Afegint una excepció

S'ha vist que la classe `Farmer` es podria inicialitzar accidentalment com una `Farm` igual a `NULL`, fent que tota la resta del programa funcionés incorrectament. Es controlarà aquest possible error en el constructor i, si es produeix, es llançarà una excepció. El programa principal no podrà ignorar aquest error i haurà de capturar i tractar l'excepció de manera controlada.

El primer que s'ha de fer és crear una excepció pròpia pel programa anomenada `FarmerException`. En C++ les excepcions normalment hereten de la classe `Exception`. La trobareu al directori `Exercise3`.

Un cop s'ha creat la excepció, es modificarà el constructor de la classe `Farmer` per comprovar els possibles errors i llençar l'excepció si és necessari. Fixeu-vos en l'ús que es fa quan es detecta el problema i es vol llençar l'excepció. Aneu en compte perquè s'ha modificat la signatura de la constructora.

```
Farmer::Farmer(int id, string name, Farm* f) {  
    this->id = id;  
    this->name = name;  
    if (f == NULL)  
        throw FarmerException("Farmer " + name + " has no farm!");  
    farm = f;  
    cout << "Constructor --> Farmer Created" << endl;  
}
```

Un cop feta la modificació, el programa principal deixarà de funcionar. S'ha de modificar de manera que capturi l'excepció.

Concretament, si es captura l'excepció, s'informarà a l'usuari de que s'ha produït un error i acabarem l'execució.

```
try  
{  
    m1= new Farmer(0,"Juan", farm1);  
}  
catch (FarmerException e)  
{  
    cout << "Exception: " << e.str() << endl;  
}
```

Es demana:

- 1 Estudieu com es crea, propaga i captura una excepció mitjançant la explicació i el codi anterior.
- 2 Afegiu una nova causa d'error a l'excepció que es llençarà quan es vulgui afegir hores de feina a una granja on el granger no hi treballi.
- 3 Afegiu una nova causa d'error a l'excepció que es llençarà quan es vulgui crear un `Farming` a la `Farm` i aquesta ja existeixi.
- 4 Feu els canvis necessaris al codi font per afegir el control d'excepcions descrit en aquest apartat.
- 5 Utilitzeu el fitxer que es proporciona, `Step0.cpp`, per provar la vostra codificació. Heu de codificar les classes de tal manera que, en executar el programa `Test3`, el resultat obtingut sigui el que es troba a `Step0.out`.

Lliurament

Heu de lliurar el codi font de totes les classes dins d'una carpeta anomenada **Exercise3**. No poseu els fitxers `.class` resultants de la compilació.

5. Criteris d'avaluació

A l'hora de corregir la pràctica es tindran en compte els següents factors:

- 1 La pràctica compila. Per optar a la valoració de la pràctica es imprescindible que **COMPILI A LA PRIMERA**. No es corregiran pràctiques que no compleixin aquest requisit.
- 2 L'execució no s'interromp en cap moment.
- 3 Es respecten els principis de la programació orientada a l'objecte.
- 4 La programació és eficient.
- 5 Es fa un ús correcte de les classes de l'API que convinguin.
- 6 La **documentació** de les classes es correcta.
- 7 La programació és elegant i la tabulació correcta.