Disseny i Programació Orientats a Objectes

Conceptes del paradigma de la Programació Orientada a Objectes (POO)

Relacions entre classes (UML)

Ester Bernadó Mansilla
Xavi Solé Beteta
Eduard de Torres Gómez

Departament d'Enginyeria
Enginyeria La Salle – Universitat Ramon Llull
9 d'octubre del 2019

Índex

1. Anàlisi i disseny orientat a objectes

L'anàlisi i el disseny orientat a objectes constitueixen la base essencial per la construcció d'un software de qualitat, robust, ben definit i de fàcil manteniment. L'anàlisi i el disseny són etapes prèvies i necessàries a la fase d'implementació d'una solució.

L'anàlisi orientat a objectes se centra a crear una descripció del domini des de la perspectiva de la classificació d'objectes, en l'obtenció de l'anomenat model del domini o diagrama de classes d'anàlisi. Aquesta descripció del domini comporta: (1) la identificació dels conceptes o classes, (2) les seves característiques o atributs rellevants i (3) les associacions que s'estableixen entre els conceptes o classes. El disseny orientat a objectes se centra a definir els objectes software i les seves col·laboracions, és a dir, com aquests treballen conjuntament per dur a termes les tasques del sistema. Els esforços dedicats a les fases d'anàlisi i disseny valen la pena, sens dubte, en relació amb els beneficis que s'obtindran la codificació.

Durant l'anàlisi i el disseny, s'elaboren models, essent el llenguatge de modelatge UML la notació més utilitzada. UML – *Unified Modeling Language* és un llenguatge formal que s'ha convertit en un estàndard de facto, amb una sintaxi i semàntica ben definida, que permet el modelatge exhaustiu i concís de sistemes. UML, entre d'altres, ofereix un gran ventall de diagrames: de casos d'ús, de classes, d'objectes, d'activitats, de seqüència, d'estats, de desplegament... que permeten analitzar, dissenyar, avaluar i criticar d'una manera ràpida una solució.

Aquest document se centra en l'anàlisi i el disseny orientat a objectes, repassa breument les relacions que es poden establir entre classes, quina és la seva notació UML corresponent, i il·lustra, mitjançant exemples, l'elaboració de diagrames de classes d'anàlisi i de disseny. Els diagrames presents en el document han estat elaborats amb l'eina de programari lliure StarUML [4].

2. Classes en UML

Com tot nou concepte, entendre què són les classes és difícil i, normalment, començar amb una analogia pot ajudar a la seva comprensió. Com analogia farem servir gossos, i en concret, el meu gos es diu Bobby.

El meu gos és un exemple d'objecte, té una identitat fàcil de reconèixer, ja que es diu Bobby, és groc i és el meu gos. Encara que m'estimi molt al meu gos, no puc pretendre que és l'únic gos que hi ha al planeta. De fet, conec molts altres gossos de la mateixa classe que en Bobby.

Una classe es pot definir com un tipus d'alguna cosa, o una plantilla per crear objectes. Els objectes i les classes estan relacionades i no podem parlar d'objectes sense parlar de classes. L'objectiu de la programació Orientada a Objectes és centrar-se en les classes, ja que són aquestes qui crearan els objectes. D'aquesta forma, una classe descriurà a un objecte sense ser l'objecte en si.

En l'exemple anterior, Bobby és un objecte de la classe Gos. Tots sabem que un Gos és una animal de quatre potes de la família dels cànids, que pot tenir diferents colors per als ulls, color de pelatge i poden tenir diferents mides i pesos, entre d'altres. Aquesta idea compartida que defineix què és un Gos per a nosaltres, és el que en el món de l'Orientació a Objectes anomenem Classe.

Mentre que una classe és una plantilla per crear objectes, un objecte és una instància d'una classe. Com a instància d'una classe, un objecte disposarà de tots els components (característiques i comportaments) que defineix la seva classe.

Seguint amb el mateix exemple, en Bobby és una instància de la classe Gos. Com a instància de Gos, en Bobby té un conjunt de característiques que el defineixen (color, altura, pes, ...) i un conjunt de comportament (bordar, estirar-se, caminar, ...) que pot dur a terme.

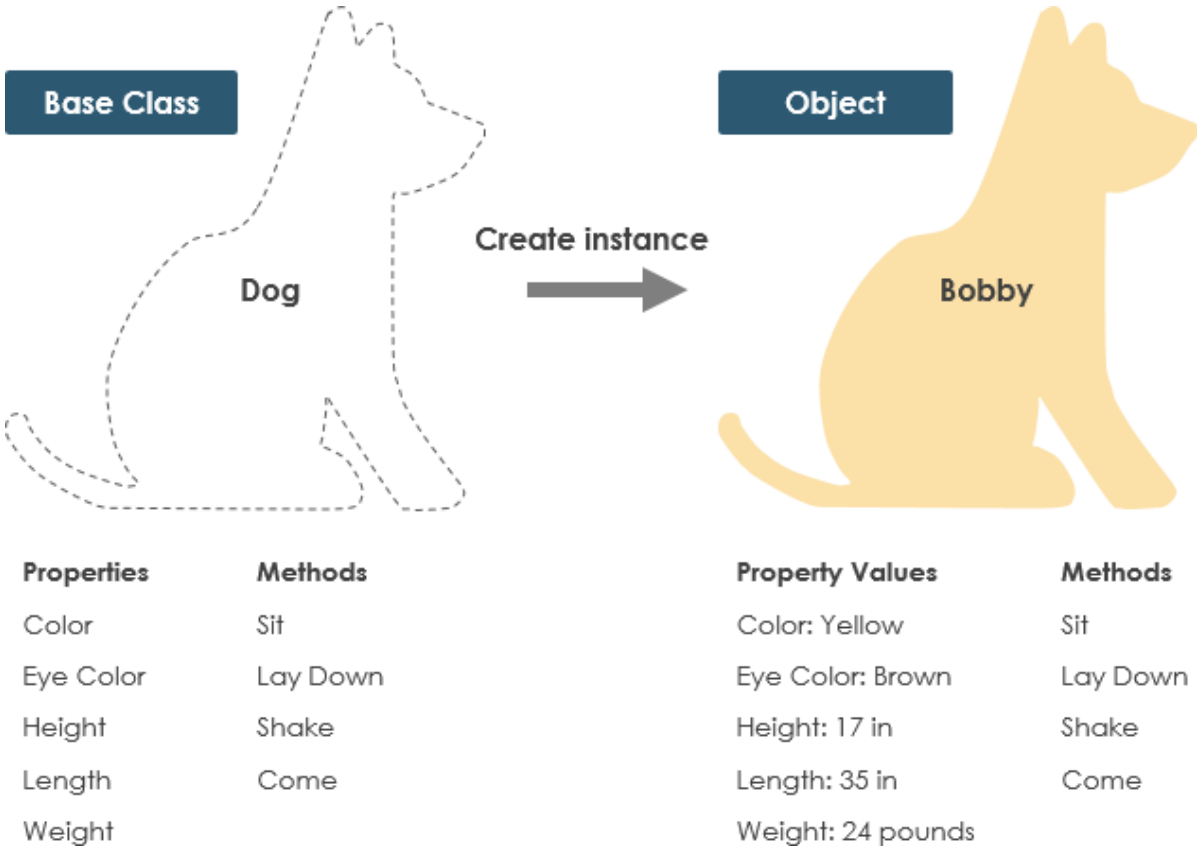

Figure 1: Exemple de classe i objecte [5]

Com es pot veure en la Figure 1, el concepte de Gos és la plantilla (Classe) que defineix un conjunt de característiques i comportaments que ens permetran crear una instància de Gos (Objecte Bobby). Aquesta instància tindrà un estat concret corresponent als valors que prenguin les seves característiques.

Pel que fa a la representació de classes en UML, l'exemple de la Figure 1 es pot expressar tal com es pot veure en la Figure 2.

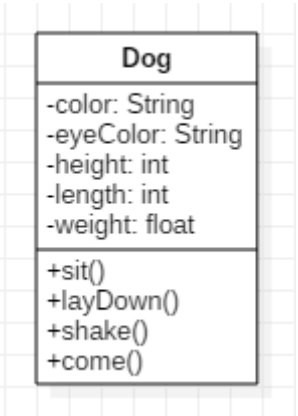

Figure 2: Classe Gos segons UML

Tota classe tindrà tres blocs:

1. Nom de la classe
2. Característiques o atributs
3. Comportaments, operacions o mètodes

2.1. Elements d'una classe

En aquest apartat detallarem els diferents elements que té una classe segons UML.

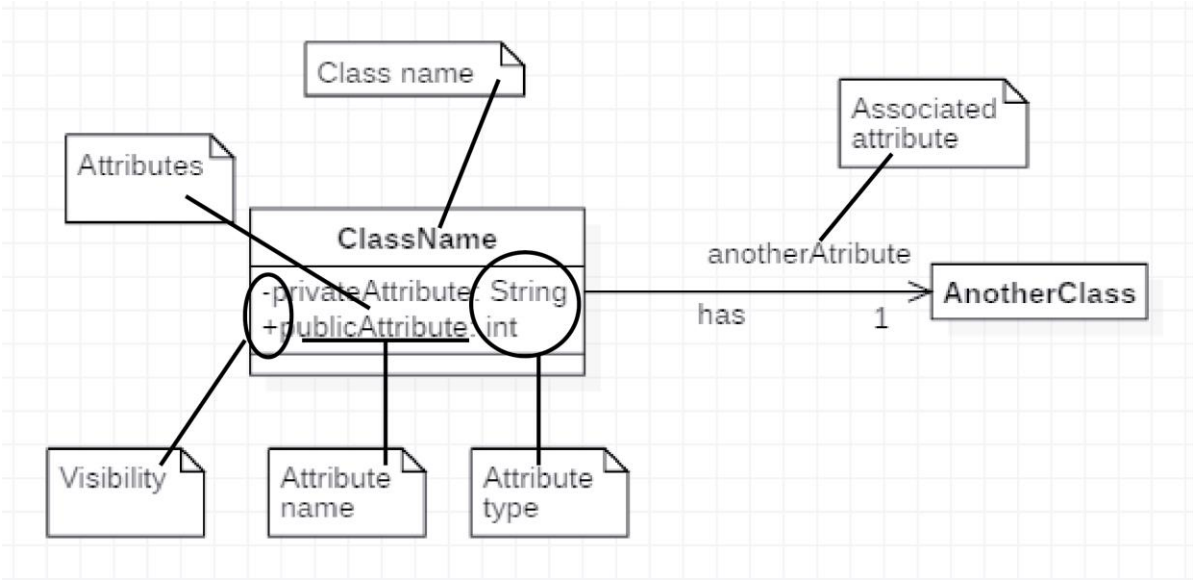

Figure 3: Elements propis d'una classe segons UML

2.1.1. Nom de la classe

El nom de la classe descriu què és aquesta classe. Per aquest motiu, voldrem que els noms de les classes siguin descriptius, simples i concisos. Per aquest motiu evitarem usar acrònims o abreviacions, excepte quan aquests siguin més habituals que el nom real (per exemple, *HTML* és més habitual que *Hyper Text Markup Language*).

Els noms de les classes els escriurem seguint el format de *UpperCamelCase*. *UpperCamelCase* defineix que tot nom s'ha d'escriure sense separadors entre paraules, amb la primera lletra de cada paraula en majúscula. *UpperCamelCase* ja és exemple del seu propi format.

2.1.2. Atributs de la classe

Els atributs o variables d'instància d'una classe són la informació que representaran l'estat actual de l'objecte. Aquests atributs poden ser de dos tipus, atributs propis o per associació. Els atributs propis són aquells que es defineixen dins la mateixa caixa que encapsula la classe, mentre que els atributs per associació, seran atributs que es generaran a conseqüència d'una relació d'associació (vegeu l'apartat 3.2) amb una altra classe.

Tot atribut constarà d'una visibilitat, un nom i un tipus. En el cas dels atributs per associació, el tipus es definirà a partir de la classe destinatària de la relació d'associació i de la seva multiplicitat (vegeu l'apartat Multiplicitat o cardinalitat d'una associació).

Pel que fa als noms dels atributs, cal recordar que l'objectiu d'un diagrama de classes és comunicar a algú el teu disseny. Per aquest motiu, serà important que els noms dels atributs siguin precisos, concisos, descriptius i autoexplicatius. Pel mateix motiu, si el teu diagrama ha de definir un software que s'haurà d'implementar amb un llenguatge concret, és important que tots els noms segueixin el format i nomenclatura definit per aquell llenguatge. Per aquest motiu, els atributs els escriurem en format *lowerCamelCase*, que implica que totes les paraules s'escriuran sense caràcters de separació amb cada un dels caràcters inicials en majúscula, a excepció del primer caràcter inicial de la primera paraula, tal com es pot veure en *lowerCamelCase.*

2.2. Encapsulament

L'encapsulament és, probablement, la característica més important per a les classes i l'orientació a objectes.

L'encapsulament és la característica més important que tenen les classes i els objectes. Com ja hem dit, tota classe defineix un conjunt de característiques o dades i un conjunt de comportament o operacions. L'encapsulament permet a les classes amagar com estan estructurades les seves dades i oferir només un conjunt d'operacions que permetin als agents externs treballar amb aquestes dades sense la necessitat de saber com estan estructurades.

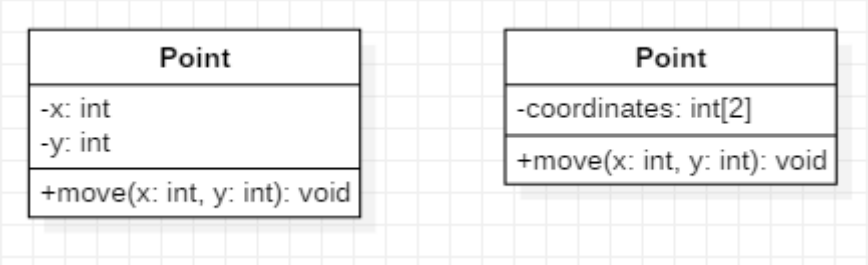

Figure 4: Diagrama de classes de la classe Point amb dues implementacions diferents de les seves característiques

En la Figure 4 podem veure, a l'esquerra, la implementació de la classe Point. Point representa un punt en un espai de coordenades de dues dimensions. Com es pot veure, els nostres punts estaran representats per dues coordenades x i y en forma de dos atributs de classe de tipus enter. A la dreta de la Figure 4, podem veure una segona representació possible de la classe Point. En aquest cas, la seva informació està representada per un únic atribut en forma d'array de dos enters. La posició 0 de l'array emmagatzemarà la coordenada x i la posició 1 de l'array emmagatzemarà la coordenada y.

Com es pot veure, la classe Point pot tenir múltiples implementacions pel que fa a les seves dades. Per contra, les operacions que oferim al nostre públic sempre són les mateixes, tenim una operació *move* que rep dos paràmetres x i y de tipus enter.

Com podem veure en aquest exemple, l'encapsulament ens permet ocultar com estem representant les nostres dades (dos enters o un array de dos enters), de forma que al nostre públic li és indiferent, ja que disposa d'un mètode *move* que ja s'adapta a les nostres dades en la seva implementació (l'implementarem nosaltres que som coneixedors de les nostres dades).

2.3. Visibilitat

Un cop explicat què és l'encapsulament que ofereix una classe, anem a veure el concepte de visibilitat.

La visibilitat és la característica que ens permetrà saber que una característica o un comportament existeix per a mi. La visibilitat està estretament relacionada amb l'encapsulament. Podríem dir que la visibilitat sorgeix quan tenim en compte l'encapsulament d'un element i el meu punt de vista actual.

UML presenta quatre nivells de visibilitat diferents. Aquesta visibilitat es pot representar, per ordre de més accessible a menys accessible, amb els símbols (*+*), (*#*), (~) i (-):

- **(+) representa la visibilitat *public***
- **(#) representa la visibilitat *protected***
- **(~) representa la visibilitat *package***
- **(-) representa la visibilitat *private***

2.3.1. Visibilitat PUBLIC

La visibilitat *public* és la visibilitat més accessible de les quatre. Es representa amb el símbol (+) davant de l'atribut o operació corresponent.

Un element PUBLIC serà sempre accessible per a qualsevol altra classe del sistema.

En la Figure 5 podem veure un exemple per il·lustrar quins són els punts de vista que poden veure un atribut *public* situat en la classe *TheClass*.[1]

[1] Nota: Un Package és un element de UML que permet organitzar i estructurar les classes que hi ha en un sistema. Es pot veure com una carpeta que ens permet classificar els continguts.

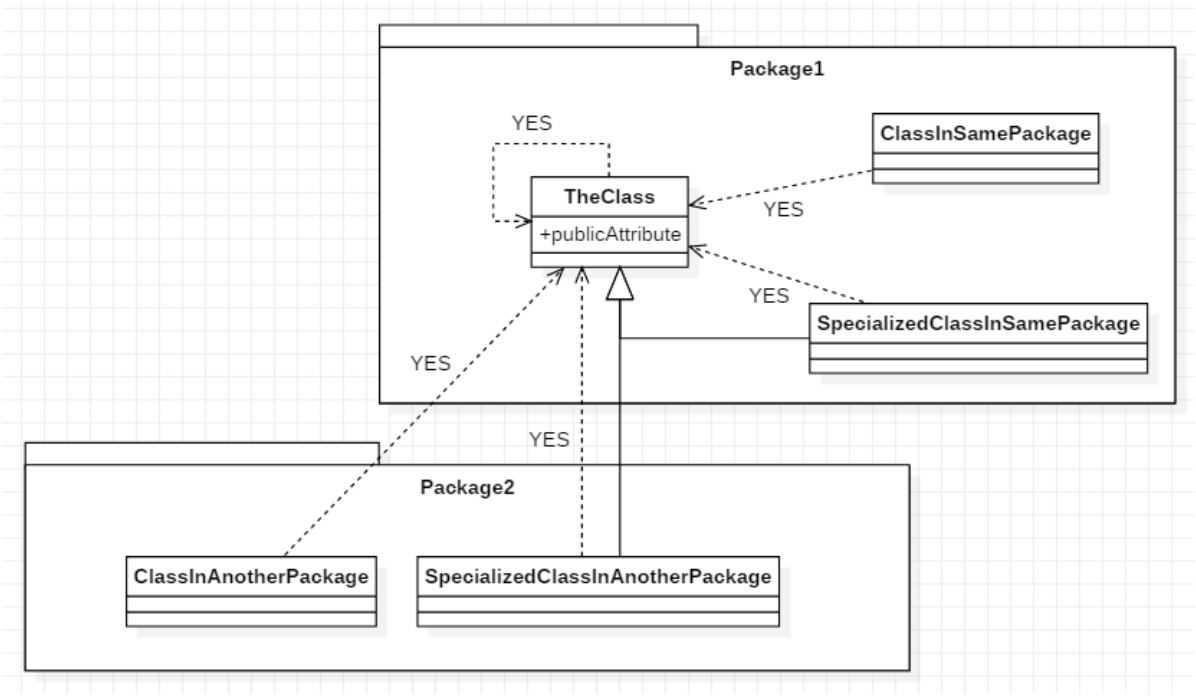

Figure 5: Diagrama de classes representant la visibilitat public

Com podem veure, qualsevol classe del sistema pot veure i accedir a aquest atribut *public*.

public o no *public?* Aquesta és la qüestió!

Molts dissenyadors d'arquitectures orientades a objectes respondrien que els atributs mai han de ser *public*. Obrir els atributs a la resta d'elements del nostre sistema és com obrir la porta de casa nostra a qualsevol persona que passi per allà: Potencialment pots provocar comportaments i usos no desitjats.

Tot i que habitualment es recomana evitar l'ús d'atributs *public*, hi ha algunes situacions en les que es pot fer una excepció. Una d'aquestes situacions és quan es treballa amb un atribut constant. És a dir, atributs que són de tipus *read-only*. Aquests atributs *read-only*, no podran canviar mai el seu valor, de forma que no és tan perillós per al sistema que siguin *public*.

Tanmateix, cal tenir en compte que un atribut constant *public*, no podrà ser subjecte a canvis en la seva representació sense afectar a totes les parts del sistema que l'utilitzin (revisar exemple de la Figure 4).

2.3.2. Visibilitat PROTECTED

La visibilitat *protected* es representa amb el símbol (#) davant de l'atribut o operació corresponent. És un nivell de visibilitat menys accessible que *public*, però més accessible que *private*.

Un element PROTECTED serà sempre accessible, tant per lectura, com per escriptura per:

1. la mateixa classe on es troba
2. per qualsevol classe que hereti d'ella (veure apartat 3.3)

En la Figure 6 podem veure un exemple per il·lustrar quins són els punts de vista que poden veure un atribut *protected* situat en la classe *TheClass*.

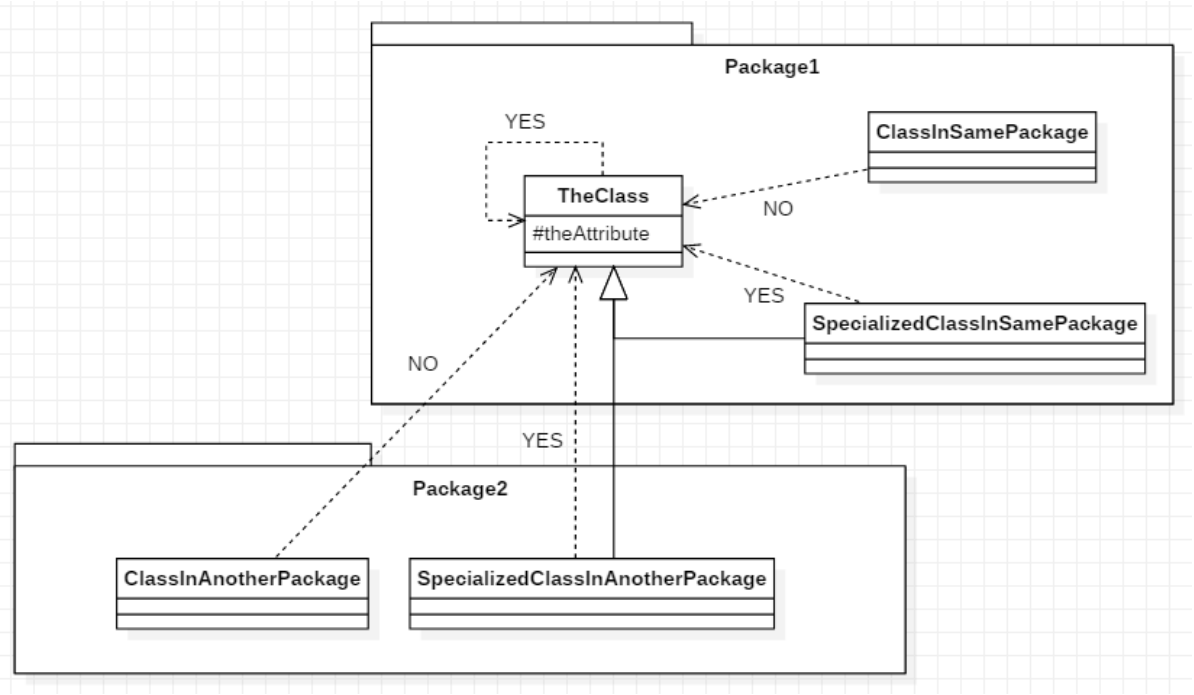

Figure 6: Diagrama de classes representant la visibilitat protected

Com podem veure en el diagrama de la Figure 6, només la mateixa classe *TheClass* i aquelles que heretin d'ella, com per exemple, SpecializedClassInSamePackage o SpecializedClassInAnotherPackage veuen l'atribut protected i poden accedir a ell.

Aquest nivell de visibilitat és molt important en classes que tinguis previst que serveixin de base per a futures classes especialitzades que vulguin heretar d'ella.

Java i protected

Fins ara en l'apartat Visibilitat PROTECTED, hem explicat com defineix UML la visibilitat *protected*. El problema que hi ha amb Java és que la forma que utilitza per implementar aquesta visibilitat no compleix amb l'estàndard UML.

La visibilitat *protected* en Java agrupa les visibilitats *package* i *protected*. En l'apartat 2.3.3 veurem exactament en què consisteix la visibilitat *package*. Java, amb la visibilitat *protected*, permet que les classes que es troben en el mateix *package* que la classe que conté l'element *protected*, també vegin i puguin treballar amb ell.

En el diagrama de la Figure 7 podem veure el mateix exemple que en la Figure 6, segons la implementació de Java.

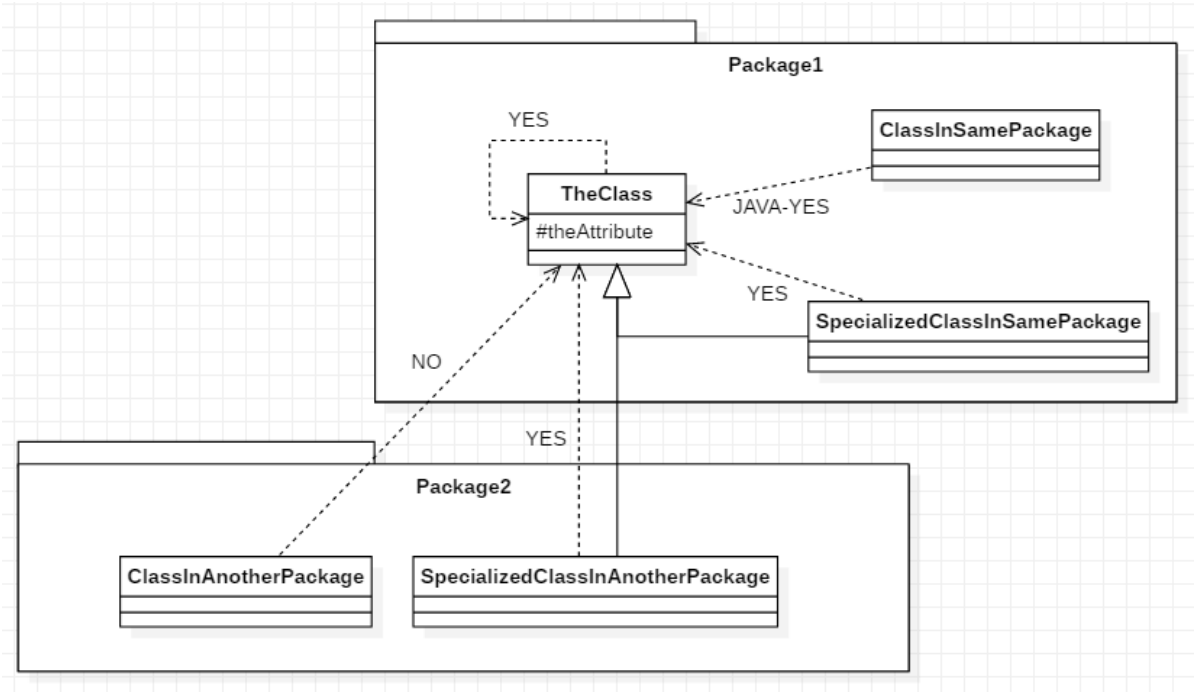

Figure 7: Diagrama de classes representant la visibilitat protected segons Java

Podem veure que la principal diferencia és que segons UML la classe *ClassInSamePackage* no pot veure l'atribut *protected*, mentre que la implementació de Java sí que permet a aquesta classe accedir a l'atribut.

2.3.3. **Visibilitat PACKAGE**

La visibilitat *package* es representa amb el símbol (~) davant de l'atribut o operació corresponent. És un nivell de visibilitat que es troba entre *protected* i *private*.

Un element PACKAGE serà sempre accessible, tant per lectura, com per escriptura per:

1. la mateixa classe on es troba
2. qualsevol classe en el mateix Package

En la Figure 8 podem veure un exemple per il·lustrar quins són els punts de vista que poden veure un atribut *package* situat en la classe *TheClass*.

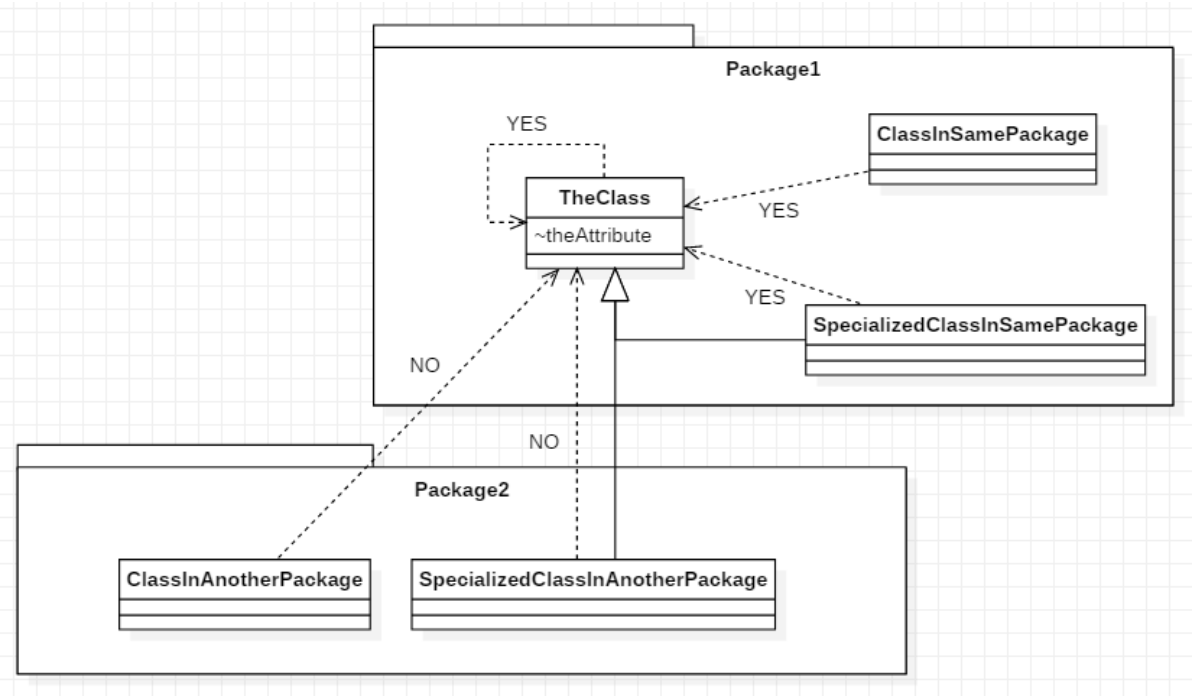

Figure 8: Diagrama de classes representant la visibilitat package

Com es pot veure, totes les classes dins el *Package1* (que és on es troba la classe *TheClass* que conté l'atribut *package-visible*) poden accedir a l'atribut, mentre que qualsevol classe ubicada en un altre Package no en coneixerà la seva existència.

2.3.4. **Visibilitat PRIVATE**

La visibilitat *private* es representa amb el símbol (-) davant de l'atribut o operació corresponent. És el nivell de visibilitat més estricte de tots.

Un element PRIVATE <u>només</u> serà accessible per la mateixa classe on es troba.

En la Figure 9 podem veure un exemple per il·lustrar quins són els punts de vista que poden veure un atribut *private* situat en la classe *TheClass*.

Com es pot veure, l'única classe que pot accedir a l'atribut és ella mateixa. Cap altra classe del sistema pot accedir a l'atribut.

Aquest tipus de visibilitat és la més habitual per als atributs d'una classe, ja que d'aquesta forma l'atribut pot ser subjecte de canvis sense afectar els usos que en fa la resta de classes del sistema. També és habitual trobar operacions *private* quan es vol implementar una operació auxiliar que no es vol publicar per a la resta d'element del sistema.

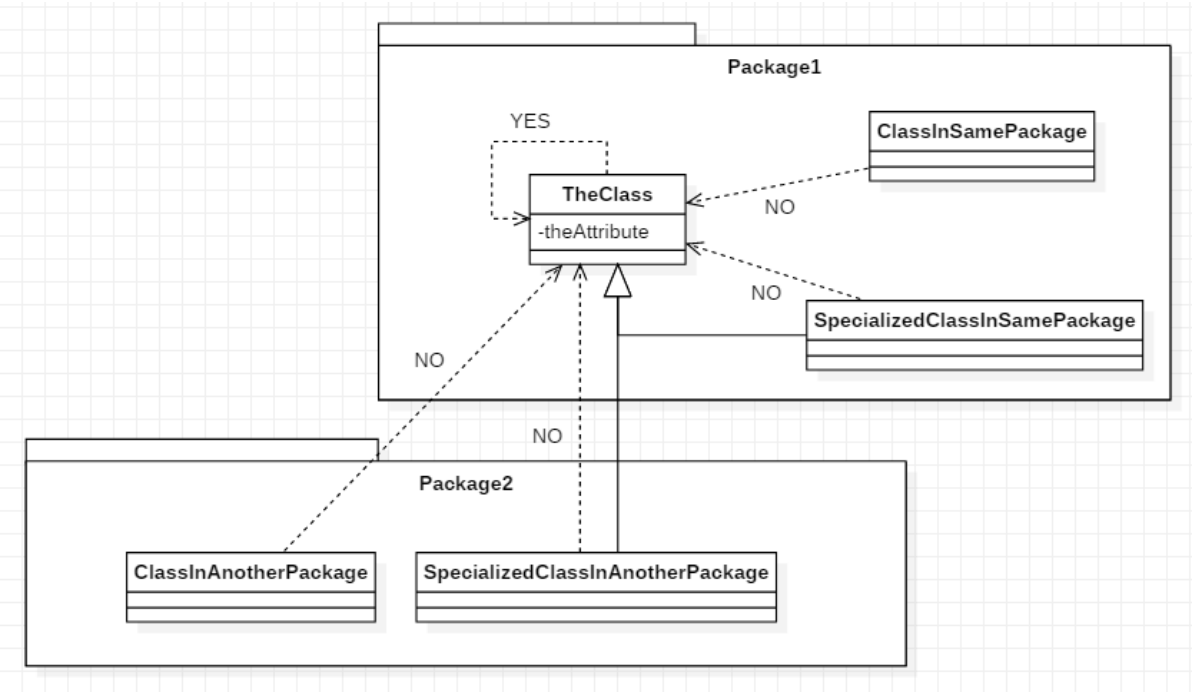

Figure 9: Diagrama de classes representant la visibilitat private

3. Relacions entre classes

Els següents punts són un breu repàs de les relacions que es poden establir entre classes.

3.1. Relació de Dependència (*Dependency*)

La dependència és la relació més dèbil que existeix entre classes. Ens indica que una classe necessita conèixer l'altra amb la finalitat d'utilitzar objectes d'aquesta altra classe momentàniament. És a dir, les classes únicament treballen conjuntament en un moment concret.

Observem a la Figure 10 com la classe Printer depèn de Document, ja que necessitarà fer ús del document per tal d'imprimir-lo.

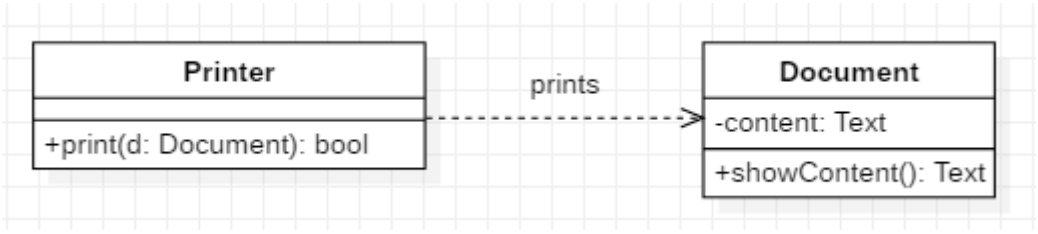

Figure 10: Diagrama de classes per una relació de dependència.

3.2. Relacions d'Associació

Les relacions d'associació, agregació, composició i classes d'associació s'exposen conjuntament en aquest apartat, ja que comparteixen certes similituds a l'hora d'implementar-se.

3.2.1. Relació d'Associació (*Association*)

La relació de dependència exposada anteriorment ens indica que una classe utilitza els serveis d'una altra esporàdicament. En canvi, l'associació significa que una classe conté una referència d'un objecte, o objectes, d'una altra classe en forma d'atribut. En aquest cas, les classes treballen conjuntament per tal d'implementar certes funcionalitats del sistema, no treballen de forma esporàdica, sinó que ho fan per un temps indeterminat.

Per exemple, el diagrama de la Figure 11 ens mostra la relació d'associació entre les classes *Worker* i *Company*, les quals mantenen aquesta relació per tal de representar que un treballador treballarà per una companyia, de forma que necessitem emmagatzemar una referència entre treballador i companyia.

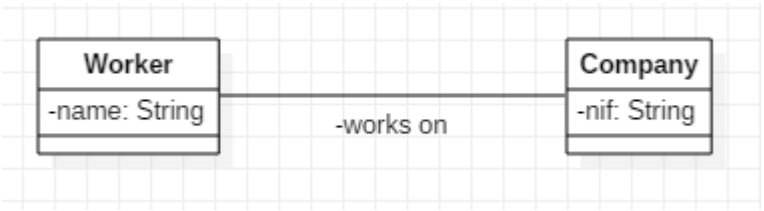

Figure 11: Diagrama de classes per una relació d'associació

Navegabilitat o direccionalitat d'una associació

Les relacions d'associació, agregació i composició, poden presentar una navegabilitat unidireccional o bidireccionalitat en funció de les referències que calgui emmagatzemar en les classes participants de la relació.

D'aquesta forma, podem tenir relacions unidireccionals quan tan sols una de les dues classes conté una referència de l'altra classe en forma d'atribut. En la Figure 12 podem observar una relació unidireccional d'associació entre la classe Companyia i la classe Treballador. En aquest exemple, només la classe Companyia emmagatzema una referència a la classe Treballador en forma d'atribut. Per contra, la classe Treballador, no contindrà cap referència de la companyia. Les relacions d'associació unidireccionals, també es poden anomenar, associacions simples o directes.

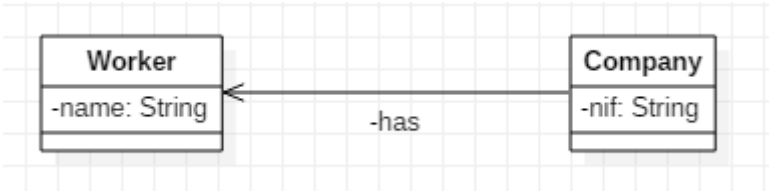

Figure 12: Diagrama de classes per una relació unidireccional d'associació

En la Figure 11 vista anteriorment, la relació entre treballador i companyia és bidireccional. En aquest cas, la relació expressa que tant el treballador com la companyia contenen referències en forma d'atributs a l'altra classe.

Multiplicitat o cardinalitat d'una associació

Les relacions d'associació, agregació i composició, poden presentar una multiplicitat en funció del nombre d'instàncies que participin d'una relació. La multiplicitat en una relació d'associació, agregació o composició indicarà, per cada un dels objectes, amb quantes instàncies es pot relacionar. Tal com es pot veure en la Figure 13, les multiplicitats s'indicaran sempre al costat de la classe destí de la relació. Les multiplicitats podran expressar valors concrets o rangs de valors.

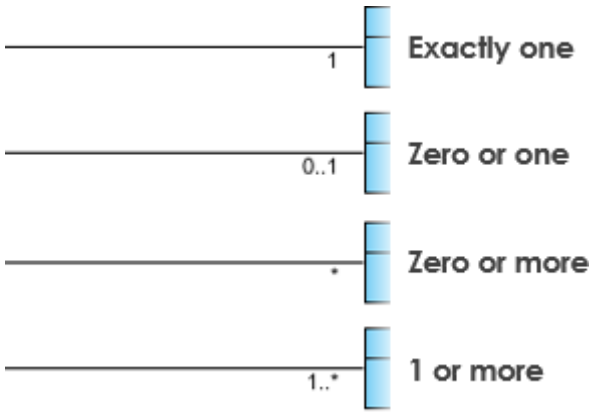

Figure 13: Tipus de multiplicitats

En la Figure 14 podem veure un exemple d'una relació d'associació bidireccional entre les classes treballador i companyia. Aquest exemple es pot llegir de la següent manera:

- Un objecte de la classe Treballador, sempre contindrà una referència a un objecte de la classe Companyia
- Un objecte de la classe Companyia, sempre contindrà una referència a múltiples objectes de la classe Treballador.

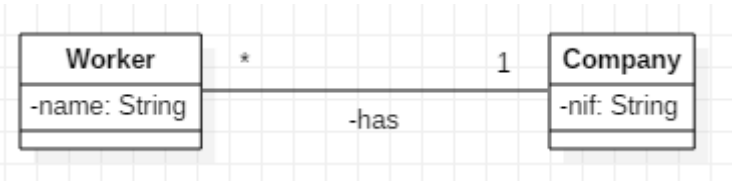

Figure 14: Diagrama de classes per una relació d'associació amb multiplicitats

Rol o atributs associats d'una associació

Les relacions d'associació, agregació i composició, poden presentar un atribut associat o rol com a part de la relació. El rol indicarà el nom de l'atribut que es crearà pel fet d'existir la relació.

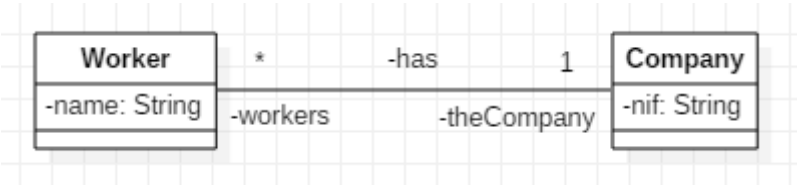

Figure 15: Diagrama de classes per una relació d'associació bidireccional amb multiplicitats i rols

En la Figure 15 podem veure un exemple complet d'un diagrama de classes amb una relació d'associació amb una navegabilitat bidireccional, multiplicitat i rols. Aquesta relació es pot llegir de la següent manera:

- Un objecte de la classe Treballador, sempre contindrà una referència a un objecte de la classe Companyia, en forma d'un atribut anomenat *theCompany* que presentarà un encapsulament o visibilitat *private*.
- Un objecte de la classe Companyia, sempre contindrà una referència a múltiples objectes de la classe Treballador, en forma d'un atribut anomenat *workers* que presentarà un encapsulament o visibilitat *private*.

Les relacions d'associació, agregació i composició, també poden ser unidireccionals. En aquest cas, només indicarem el rol i multiplicitat dels atributs que es generen.

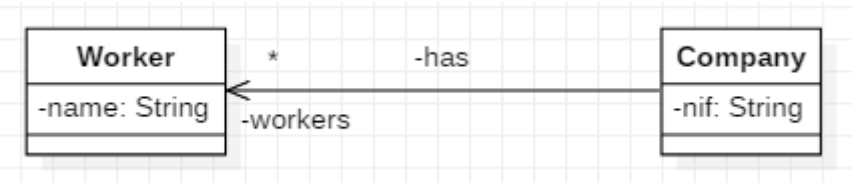

Figure 16: Diagrama de classes per una relació d'associació unidireccional amb multiplicitats i rols

En la Figure 16 podem veure un exemple complet d'un diagrama de classes amb una relació d'associació amb una navegabilitat unidireccional, multiplicitat i rols. Aquesta relació es pot llegir de la següent manera:

- Un objecte de la classe Companyia, sempre contindrà una referència a múltiples objectes de la classe Treballador, en forma d'un atribut anomenat *workers* que presentarà un encapsulament o visibilitat *private*.
- Els treballadors no tenen la informació sobre a quina companyia treballen (potser no és important per al sistema).

3.2.2. **Relació d'Agregació (*Aggregation*)**

L'agregació és un tipus de relació d'associació que mostra més informació sobre com es relacionen dues classes. Una agregació indica que un objecte d'una classe és part d'un objecte d'una altra (relació "tot-part").

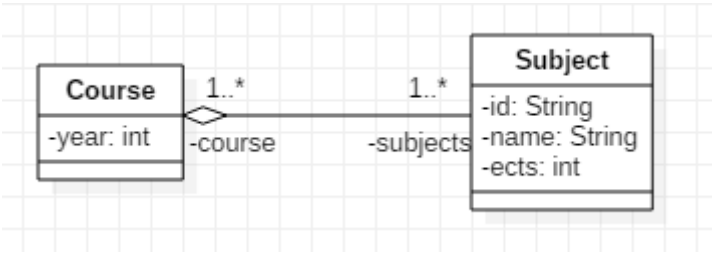

Figure 17: Diagrama de classes per una relació d'agregació.

El llenguatge de modelatge UML especifica que el diamant vuit es col·loca a la banda del "tot", que l'objecte "part" pot ser agregat per diverses classes, i que normalment no ens prenem la molèstia de posar nom a aquesta relació, doncs la mateixa notació porta implícita que una classe és part de l'altra.

Observem a la Figure 17 un exemple de com la relació d'agregació ens indica que un curs està format per diverses assignatures. La classe Curs és el "tot" i Assignatura és la "part". Observeu també que en aquest exemple, la relació té una navegabilitat bidireccional per expressar que un curs està format per una o més assignatures i que l'assignatura forma part de, com a mínim, un curs. Noteu també que els rols ens indiquen que els objectes de la classe curs tindran una referència de tipus assignatura anomenada *subjects*, i que els objectes de la classe assignatura, tindran una referència de tipus curs anomenada *course*.

3.2.3. **Relació de Composició (*Composition*)**

La composició és un tipus especial d'agregació que imposa certes restriccions. En aquest cas, el "tot" posseeix fortament a les seves parts. Si es copia o s'esborra l'objecte del "tot", les seves parts també es copien o s'eliminen. És a dir, existeix una dependència entre els cicles de vida de les instàncies del "tot" i les "parts". Perquè això sigui possible la composició ha de ser navegable des del "tot" fins a les "parts". A més a més la multiplicitat de l'extrem del "tot" ha de ser 1 o 0..1. És a dir, a diferència de l'agregació una part no pot pertànyer a més d'una classe.

Imaginem un tauler d'escacs format per 64 caselles. Cada Casella és part del Tauler ("tot"), i no tindria sentit copiar o eliminar un tauler sense copiar o eliminar les caselles que el conformen. En aquest cas la composició és apropiada, podem veure la seva representació a la figura següent:

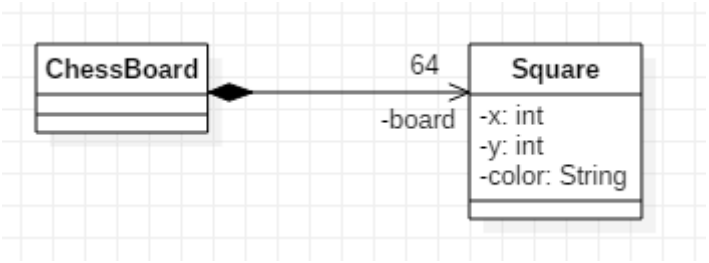

Figure 18: Diagrama de classes per una relació de composició

3.2.4. **Classes d'associació (*Association class*)**

Les classes d'associació ens permeten introduir atributs i mètodes a les relacions entre classes. La característica intrínseca d'aquest tipus de relació és que una única instància de la classe d'associació està relacionada amb el parell d'objectes de la relació. En la Figure 19 podem veure un exemple:

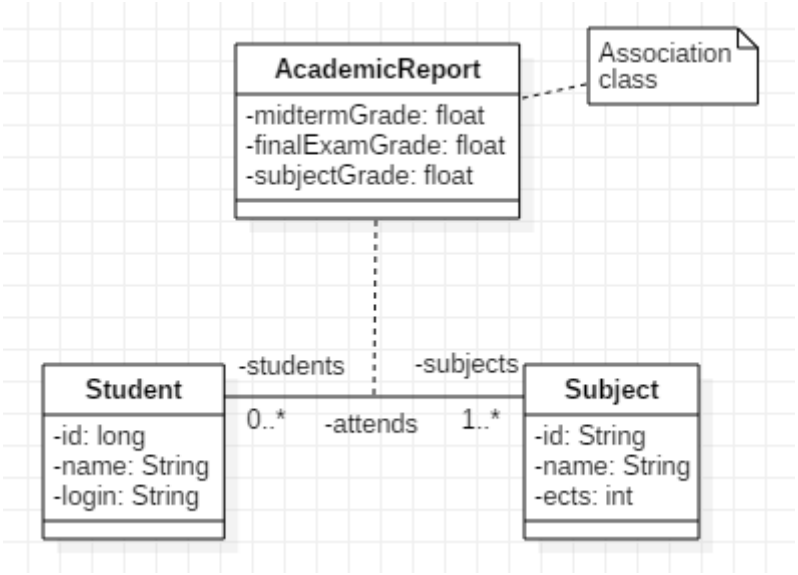

Figure 19: Diagrama de classes per una classe d'associació

Observem com un Alumne pot estar matriculat de diverses assignatures, i necessitem guardar les notes obtingudes en el seu expedient acadèmic. Podem guardar aquesta informació afegint els atributs *midtermGrade*, *finalExamGrade* i *subjectGrade* a l'associació entre Alumne i Assignatura donant lloc a la classe d'associació ExpedientAcademic. La restricció comentada anteriorment ens relaciona el parell Alumne i Assignatura amb una única instància d'ExpedientAcadèmic. És a dir, un alumne pot estar matriculat de diverses assignatures, però l'expedient acadèmic obtingut en cada assignatura és únic. Cal tenir en compte que si un alumne obtingués més d'un expedient acadèmic per assignatura, llavors una classe d'associació no seria correcta per modelar aquesta relació.

Podem modelar aquesta mateixa relació d'una manera diferent, fent que ExpedientAcadèmic sigui una classe per si sola, tal com es pot veure en Figure 20. Cal fixar-se en els valors de les multiplicitats, i veure que una i només una instància Nota relaciona dos objectes de les altres classes. Això està relacionat amb la restricció que imposa una classe d'associació.

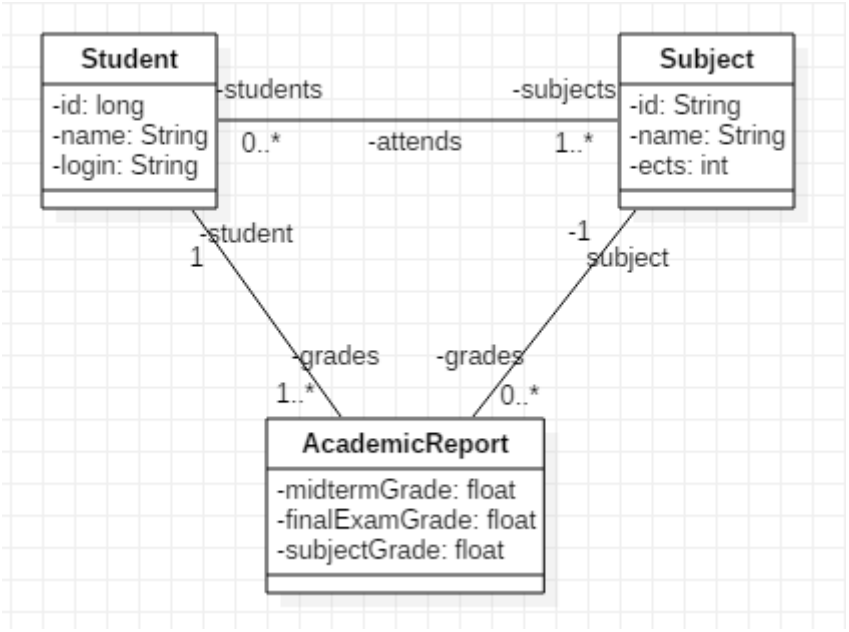

Figure 20: Diagrama de classes alternatiu per a una classe d'associació

3.3. **Relació d'Herència (*Generalization*)**

La relació d'herència o generalització s'utilitza per descriure que una classe és un subtipus d'una altra classe. Una generalització és una relació entre un element més general i un element més específic. Cada instància de l'element específic és també una instància indirecta de l'element general. Així, l'element específic hereta les característiques i comportaments de l'element més general. Una herència o generalització, habitualment es pot detectar per l'ús de l'expressió "és un" ("is a"). Observem un exemple en les Figure 21 i Figure 22 :

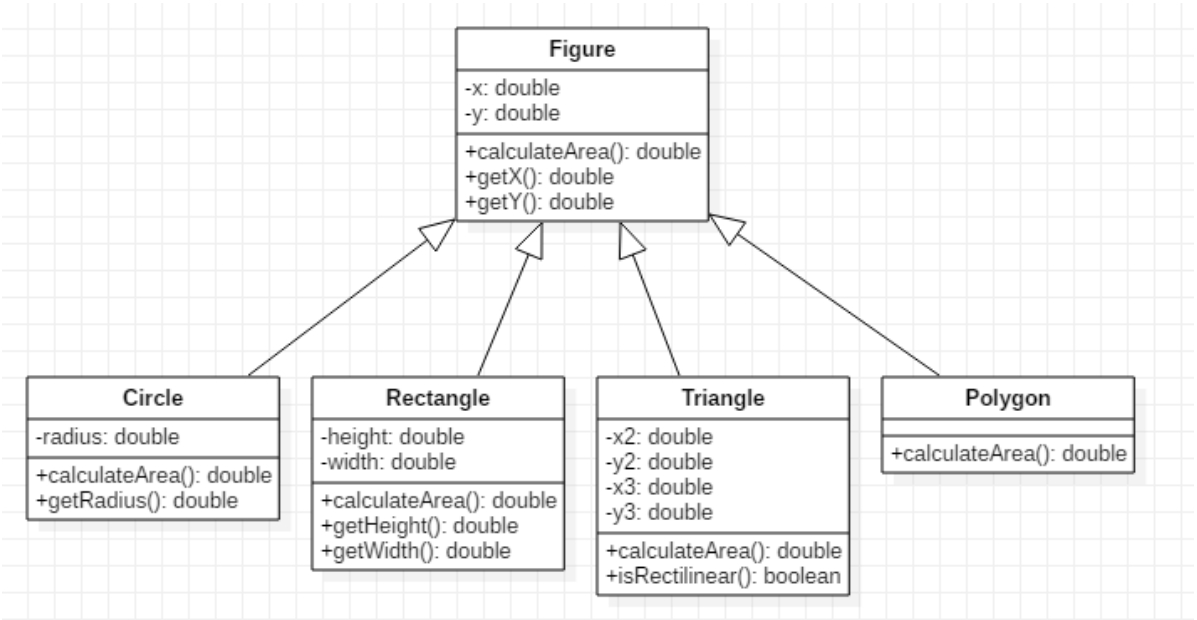

Figure 21: Diagrama de classes d'un conjunt de relacions d'herencia

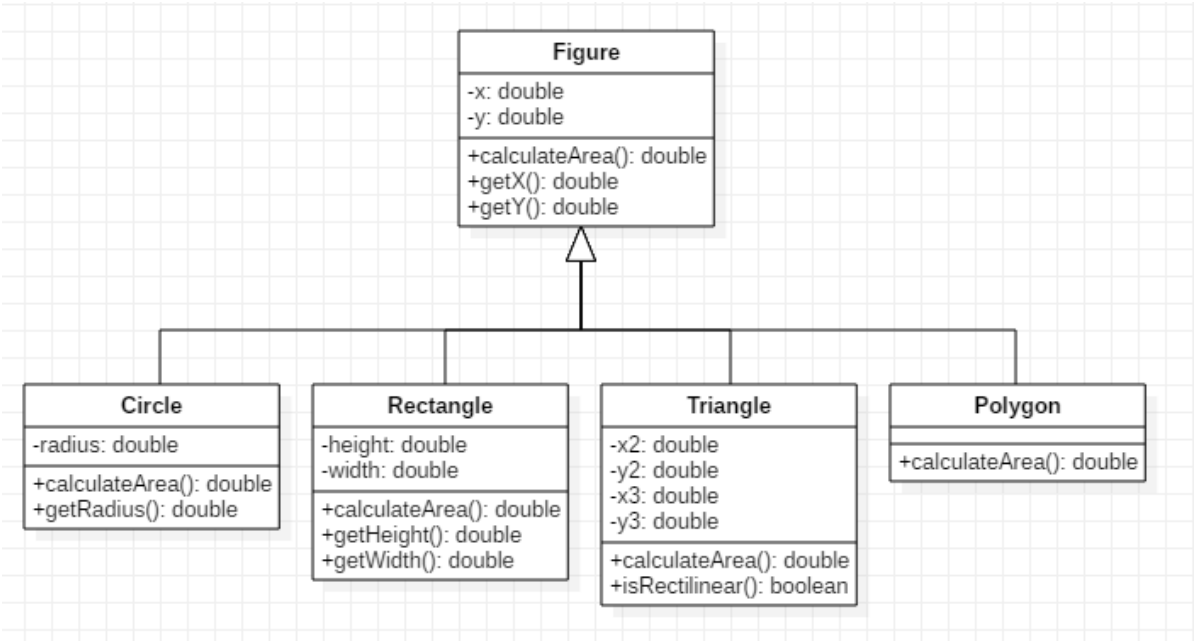

Figure 22: Diagrama de classes d'un conjunt de relacions d'herència amb les relacions agrupades

Tal com es pot veure en les figures Figure 21 i Figure 22, les classes Cercle, Rectangle, Triangle i Polígon hereten de la classe Figura. D'aquesta forma, es pot llegir la relació com que "un Cercle és una Figura", "un Rectangle és una Figura", ...

Pel fet d'heretar de Figura, totes les instàncies de la classe Cercle heretaran també els atributs x i y de la classe Figura, això significa que un objecte Cercle tindrà en total els atributs *x*, *y* i *radius*, mentre que un objecte Rectangle tindrà els atributs *x*, *y*, *height* i *width*.

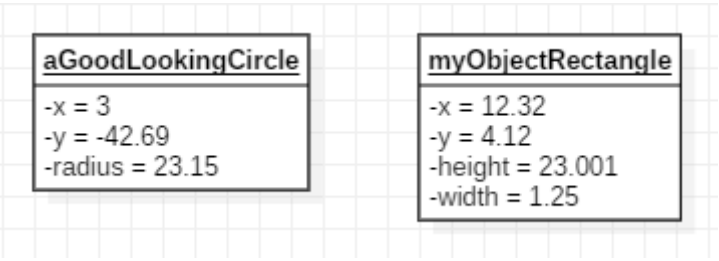

Figure 23: Diagrama d'objectes d'un cercle i un rectangle

Encapsulament o visibilitat *protected*

Cal destacar que, encara que un objecte hereti tots els atributs i mètodes definits en la classe general (Classe Figura en l'exemple Figure 22), l'encapsulament segueix aplicant. D'aquesta forma, qualsevol operació implementada dins la classe Cercle, per exemple, podrà veure els atributs de la mateixa classe (radius), però no veurà els atributs de la seva classe general (x i y), ja que el seu encapsulament és privat.

Per tal d'aconseguir que les classes que heretin de Figura puguin veure els atributs x i y, caldrà treballar amb l'encapsulament *protected* (#).

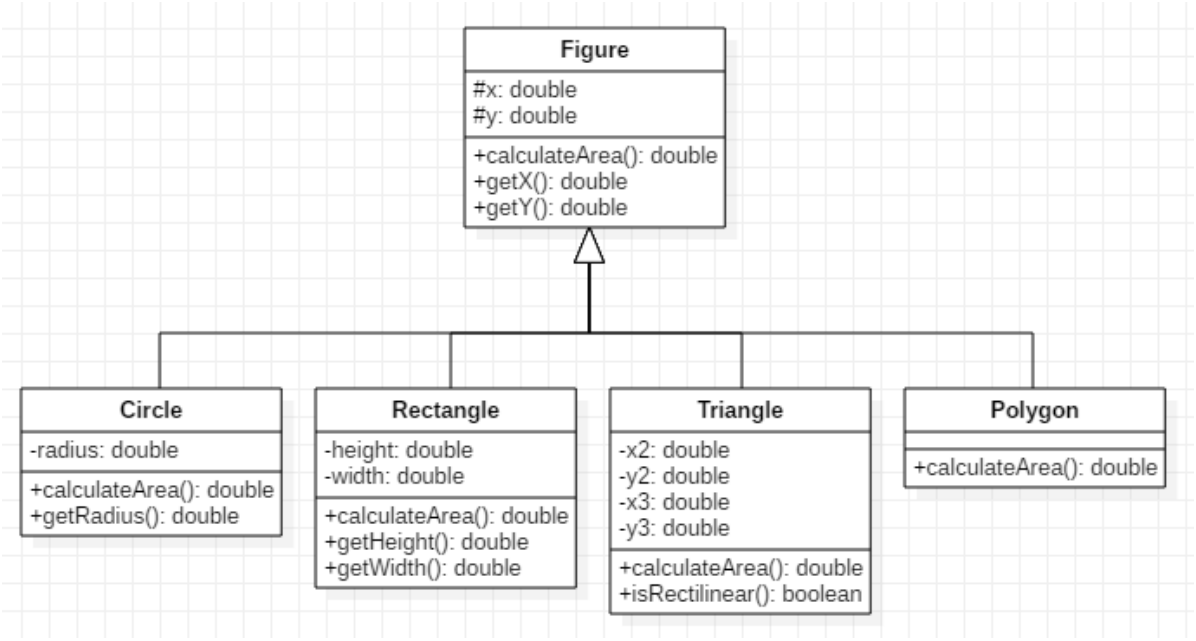

Figure 24: Diagrama de classes on els atributs de la classe general son protected

4. Bibliografia

[1] Stevens, Perdita i Pooley, Rob. *Utilización de UML en Ingeniería del Software con Objectos y Componentes*. 2a Edició. Madrid: Pearson Education, 2007. ISBN 978-84-7829-086-4.

[2] Miles, Russ and Hamilton, Kim. Modeling a System's Logical Structure: Advanced Class Diagrams. *Learning UML 2.0 - A Pragmatic Introduction to UML*. 1a Edició. Estats Units d'Amèrica: O'Reilly, 2006, p. 83-89.

[3] Bennet, McRobb i Farmer. Análisis de requisitos. *Análisis y diseño orientados a objetos de sistemas. Usando UML.* 3a Edició. India: Mc Graw Hill, 2006, p.171-221.

[4] StarUML - The Open Source UML/MDA Platform. Disponible a: <http://staruml.io >

[5] UML Class Diagram Tutorial. Disponible a <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>