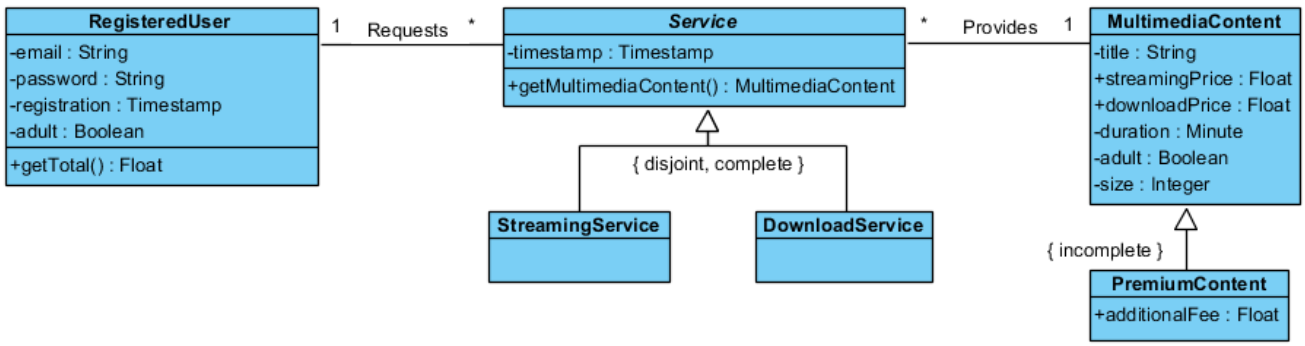
Anàlisi i Disseny amb Patrons

Pràctica 1: Principis de disseny, patrons d'anàlisi i arquitectònics

Ens han contractat per desenvolupar un programari per gestionar el streaming de pel·lícules en línia. El programari formarà part d'un sistema més gran, però només haurà de gestionar informació sobre les pel·lícules ofertes, els membres de la plataforma i les reproduccions i descàrregues que realitzen a la mateixa.

Obviarem molts elements que un sistema real tindria. La raó d'això no és altra que la de fer una pràctica didàctica, i que la dificultat i dedicació prevista siguin les desitjades.

Disposem ja d'un anàlisi previ que podem utilitzar com a punt de partida, però que caldrà corregir i millorar:



Claus de les classes:

- *RegisteredUser: email*
- *Service: timestamp*
- *MultimediaContent: title*

Restriccions d'integritat:

- Per tot servei sol·licitat per un usuari registrat, la data i hora del servei ha de ser posterior a la data i hora de registre de l'usuari a la plataforma.
- Contingut multimèdia per a adults no pot ser ofert a usuaris registrats no adults.

Com podem veure al diagrama de partida, els usuaris tenen una operació que retorna l'import total pagat per tots els serveis que han sol·licitat. El preu d'un servei es calcula de la següent manera:

- Per a tots els usuaris que volen veure un contingut multimèdia per streaming, s'aplica el preu de streaming d'aquest contingut multimèdia.
- Per a tots els usuaris que volen descarregar un contingut multimèdia de la plataforma, s'aplica el preu de descàrrega d'aquest contingut multimèdia.
- Si el contingut és premium, en qualsevol dels casos anteriors s'afegeix el càrrec addicional especificat a l'atribut additionalFee.

Suposem que la implementació d'aquest mètode és com el següent pseudocodi:

```
public class RegisteredUser
{
  private List <Service> services;
  public float getTotal ()
  {
    float total = 0.0;
    foreach (Service s in services)
    {
      MultimediaContent mc = s.getMultimediaContent ();
      if (typeof (s) == StreamingService) total + = mc.streamingPrice;
      else if (typeof (s) == DownloadService) total + = mc.downloadPrice;
      if (typeof (mc) == PremiumContent) total + = mc.additionalFee;
    }
    return total;
  }
}
```

Exercici 1 (10%)

Revisem el pseudocodi de l'operació *getTotal* de la classe *RegisteredUser* i ens preocupa que el seu disseny sigui una mica fràgil ja que no veiem clar si contempla els possibles escenaris futurs i el seu impacte:

- canviem el tipus de dades d'imports o la forma de calcular un càrrec addicional als nostres continguts premium; i / o
- ampliem el sistema per admetre un altre tipus de servei (com la compra en suport físic).

a) Indica si aquest disseny satisfà o no la Llei de Demeter i raona la teva resposta.
b) Indica si aquest disseny satisfà o no el principi de disseny Obert-Tancat i raona la teva resposta.

Solució:

a) No, aquesta solució no satisfà la Llei de Demeter, ja que la classe *RegisteredUser,* en el mètode *getTotal,* utilitza la classe *MultimediaContent,* que no té directament associada. És a dir, obté instàncies de *MultimediaContent* que no té associades, invoca mètodes de les mateixes i, per tant, *parla amb desconeguts*. En conseqüència, qualsevol canvi en aquesta lògica pròpia dels continguts multimèdia, afectaria el codi de RegisteredUser.
b) No, aquesta solució viola el principi de disseny Obert-Tancat, ja que la classe *RegisteredUser* és conscient de les subclasses de *Service* i n'invoca un mètode o un altre segons la subclasse a la qual pertany l'objecte de cada iteració. Si, més endavant, ampliem el sistema per admetre una nova subclasse de *Service* hauríem de modificar el codi de *RegisteredUser* per obtenir el preu corresponent dins d'un nou else-if.

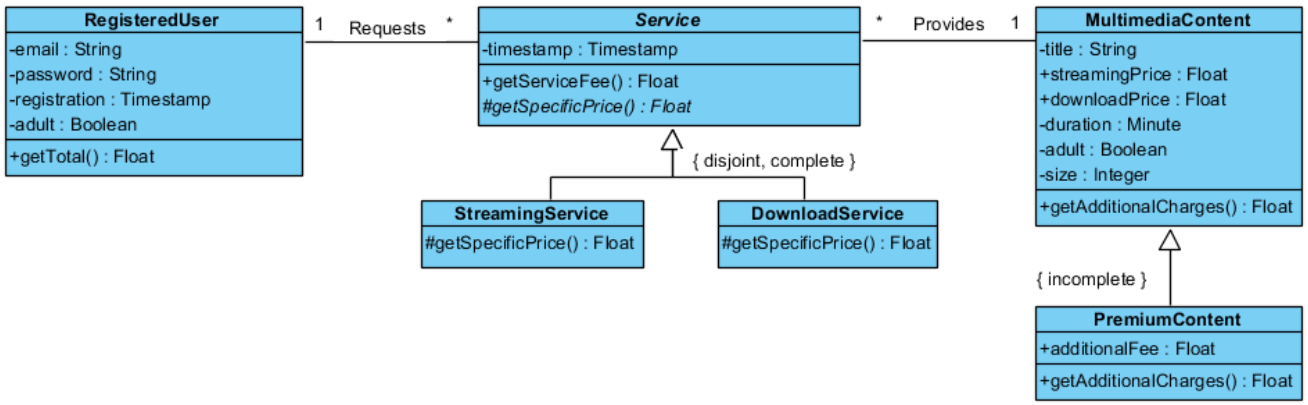
Exercici 2 (30%)

Proposa una solució alternativa (també en pseudocodi del mateix estil) que corregeixi els problemes de l'operació *getTotal* de *RegisteredUser* que has detectat en l'exercici anterior i que per tant compleixi qualsevol principi violat. Realitza tots els canvis que consideris necessaris a qualsevol de les classes del model de l'enunciat.

Solució:

```
public class RegisteredUser
{
      private List <Service> services;

      public float getTotal ()
      {
            float total = 0.0;
            foreach (Service s in services) total + = s.getServiceFee();
            // aquesta classe ja només "parla" amb Service,
            // qui sí es podrà comunicar amb MultimediaContent
            // a l'estar associada amb ella
            return total;
      }
}

public abstract class Service
{
      protected MultimediaContent content;

      // operació plantilla
      public float getServiceFee ()
      {
            return this.getSpecificPrice() + content.getAdditionalCharges();
      }

      protected abstract float getSpecificPrice();
      // operació abstracta que serà implementada a cada subclasse;
      // la decisió de quin preu aplicar depèn de la subclasse
}
```

```
public class StreamingService inherits Service
{
      protected override float getSpecificPrice ()
      {
            return content.streamingPrice;
      }
}

public class DownloadService inherits Service
{
      protected override float getSpecificPrice ()
      {
            return content.downloadPrice;
      }
}

public class MultimediaContent
{
      public float getAdditionalCharges ()
      {
            return 0.0;
            // operació polimòrfica;
            // per defecte, un contingut no té càrrec addicional
      }
}

public class PremiumContent inherits MultimediaContent
{
      private float additionalFee;
      public overrides float getAdditionalCharges ()
      {
            return additionalFee;
            // només els premiums tenen càrrec addicional, així que
            // sobreescrivim l'operació de la superclasse per així informar-lo
      }
}
```



Exercici 3 (30%)

Seguint amb els preus, ens adonem que si un contingut canvia de preu (per exemple, de streaming), necessitem que els serveis ja realitzats abans d'aquest canvi no canviïn els seus preus.

Ens proposem buscar una solució que ens permeti conèixer el preu de cada contingut en el moment en què es va fer la sol·licitud de servei així com en qualsevol moment passat.

Voldríem, doncs, que en lloc d'un senzill atribut per a cada preu (streaming i descàrrega), de cada contingut multimèdia poguéssim saber-ne els preus en qualsevol moment del passat i suposeu que per a això disposem d'una nova classe *Price* per representar preus i que ofereix un mètode *getValue* per obtenir-ne el valor de tipus float.

a) Descriu breument com solucionaries aquest problema. En cas que proposis l'aplicació d'un patró, explica quin i raona la seva conveniència.
b) Proposa una solució detallada en forma de diagrama de classes. Mostra només les classes que canviïn respecte al diagrama de l'enunciat.
c) Indica com seria el pseudocodi resultant per què un contingut multimèdia retorni l'últim preu de streaming que està en vigor.

Solució:

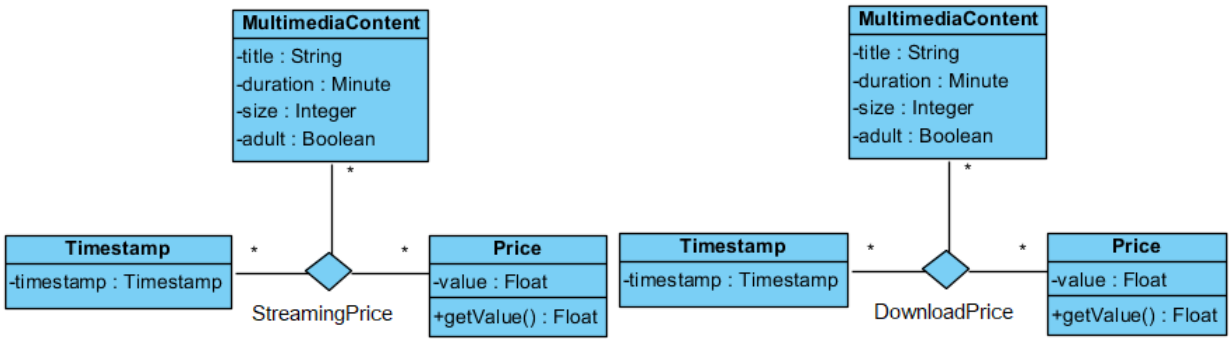
a)

Ara mateix el preu d'un contingut multimèdia és un atribut d'aquest i, per tant, de cada contingut només en sabem el preu actual. Tot i que es tracta d'un atribut i no d'una associació, el patró que ens permet resoldre aquesta problemàtica és el patró Associació Històrica.

Aquest patró ens permet transformar una associació (o un atribut, com és el cas) en un històric dels valors que ha pres l'associació o atribut al llarg del temps. Aquest és exactament el problema que volem resoldre.

b)

Com que el que tenim és un atribut, necessitem aplicar la variació en què primer extraiem l'atribut com una associació. El resultat final és:

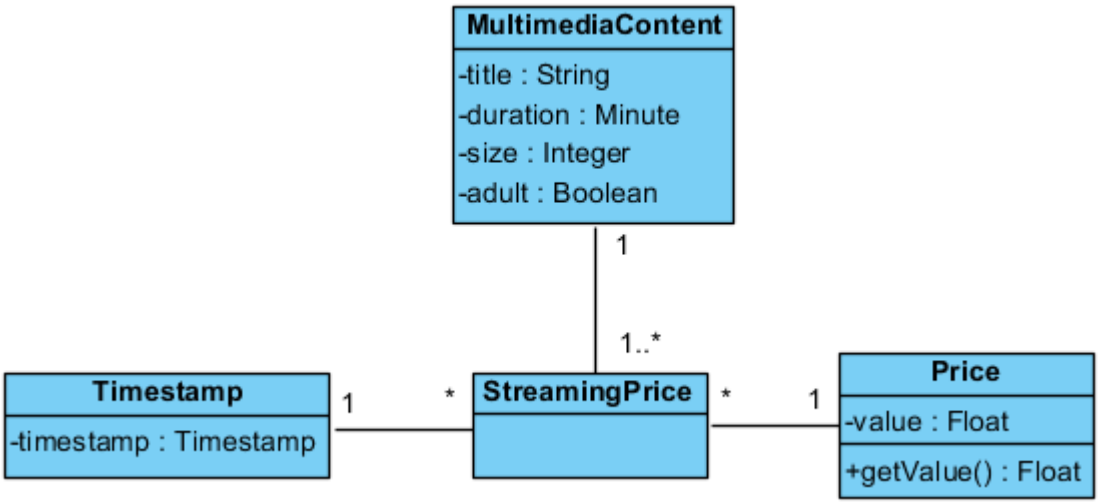


Aquest diagrama reflecteix una restricció que abans no teníem: donat un contingut i una data-hora (instant o timestamp) concreta, només podem tenir un preu (per a cada tipus de servei). És a dir, un contingut només pot tenir un canvi de preu a un instant donat.

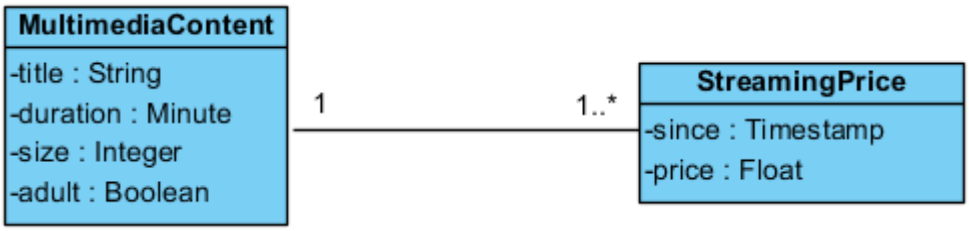
En aquesta solució assumim que el preu d'un contingut és l'actual fins que apareix una associació amb un *Timestamp* posterior, moment en què aquell preu passa a ser històric i ja no és l'actual. Per tant:

- Estem ja reflectint que un contingut multimèdia no pot tenir dos preus del mateix servei al mateix temps, ja que sempre assumim que si hi ha dues associacions cadascuna representa un preu que deixa de ser vàlid quan comença el següent.
- Estem assumint que un contingut NO pot tenir períodes de temps en què no té cap preu per a cada servei.

D'altra banda, amb els llenguatges orientats a objectes actuals no se sol permetre utilitzar associacions ternàries. Es pot fer modelant l'associació històrica com si tingués una classe associativa i després fent una petita transformació. Per exemple, amb *StreamingPrice*:

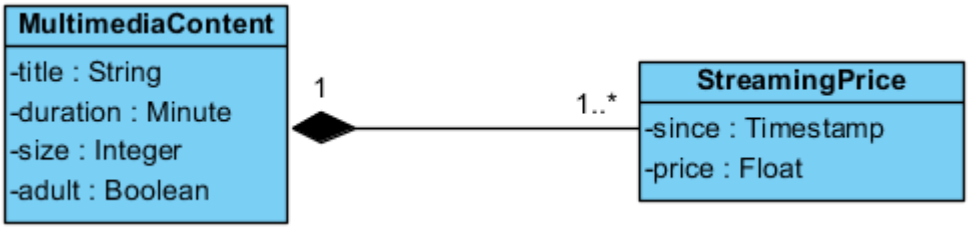


De fet, com *Timestamp* i *Price* els podríem considerar tipus de dades, que normalment posaríem com a atributs, aquesta solució la podem transformar en:



Finalment, podem veure que tant *StreamingPrice* com *DownloadPrice* són components d'un contingut multimèdia, ja que pertanyen a un i només un contingut multimèdia, no té sentit que ho

canviem de contingut i si destruïm un contingut haurem de destruir el seu històric de preus. Per tant, podem modelar com:



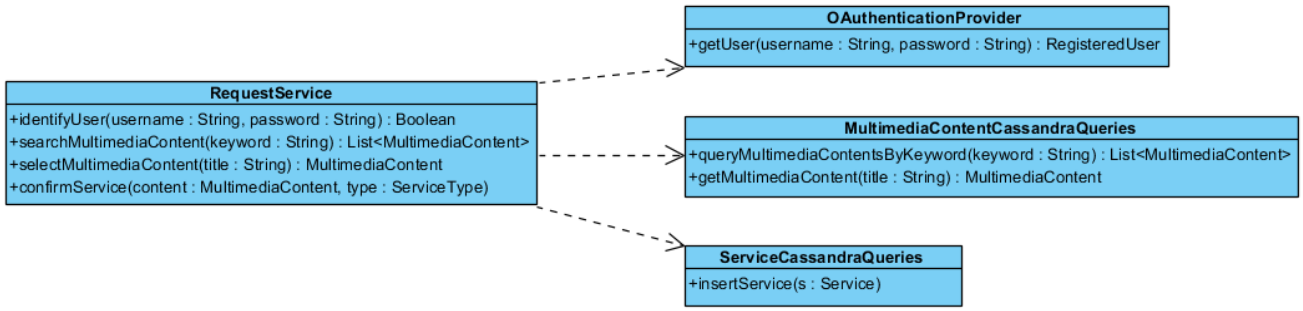
c)

```
public class MultimediaContent
{
	private List <StreamingPrice> streamingHistory;

	public float getCurrentStreamingPrice ()
	{
		if (streamingHistory.isEmpty()) return null;
		return streamingHistory.last().price;
	}
}

public class StreamingPrice
{
	public Timestamp since;
	public float price;
}
```

Exercici 4 (20%)

Comencem ara a dissenyar l'arquitectura del programari del sistema i hem decidit utilitzar una arquitectura amb 3 capes: presentació, domini i serveis tècnics.

Durant el disseny dels primers elements ens trobem en un punt en què disposem d'una classe *RequestService* de la capa de domini que ens permet anar donant resposta, a nivell d'abstracció de la lògica, al cas d'ús pel qual un usuari registrat s'autentica, busca entre el contingut multimèdia disponible, selecciona un del seu gust i sol·licita reproduir-lo per streaming o realitzar-ne una descàrrega.



A mesura que es van realitzant les diferents interaccions del cas d'ús, es van executant els mètodes de RequestService:

- *identifyUser:* Invocada per un subsistema de seguretat que informa quin és l'usuari que està utilitzant el sistema. La classe *RequestService* pot buscar les dades de l'usuari utilitzant *OAuthenticationProvider.* Si és efectivament obtingut des d'*OAuthenticationProvider*, el cas d'ús pot continuar.
- *searchMultimediaContent:* Obté un llistat de continguts multimèdia disponibles a la plataforma que compleixen els criteris de cerca en el títol. La classe *RequestService* pot buscar les dades dels continguts multimèdia utilitzant *MultimediaContentCassandraQueries*.
- *selectMultimediaContent:* Obté un contingut multimèdia a partir de la selecció de títol que realitza l'usuari sobre el llistat obtingut en l'operació anterior. L'obtenció de l'objecte concret es realitza utilitzant *MultimediaContentCassandraQueries*.
- *confirmService:* Crea el servei a partir del contingut multimèdia escollit i el tipus de servei desitjat (streaming o descàrrega), i l'enregistra a la base de dades mitjançant *ServiceCassandraQueries.* També, obté els binaris del contingut multimèdia en el format òptim per al reproductor de la nostra plataforma, si ha seleccionat streaming, o bé obté el fitxer descarregable, si ha seleccionat descàrrega; però no representem això últim per tal de simplificar el model.

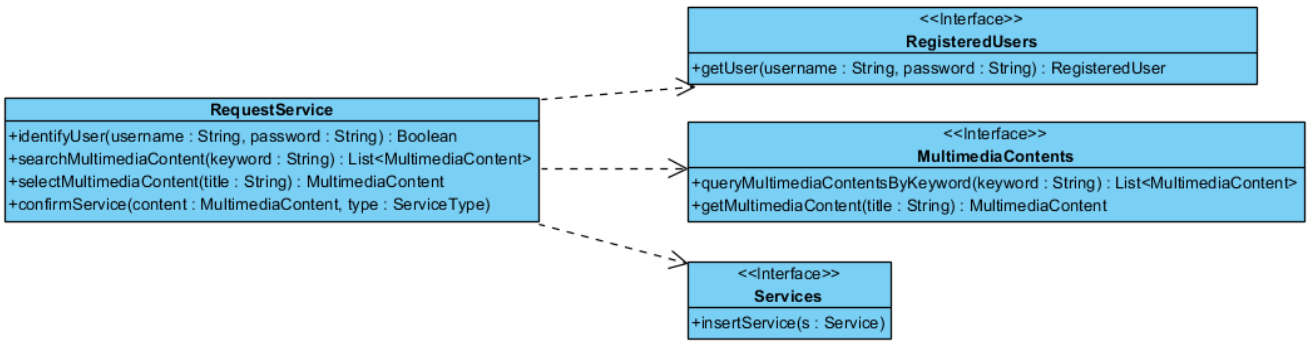
Les classes *OAuthenticationProvider*, *MultimediaContentCassandraQueries* i *ServiceCassandraQueries*, com podeu deduir, són de la capa de serveis tècnics.

Creus que aquesta classe compleix tots els principis de disseny que hem vist? En cas afirmatiu, justifica per què creus que és així. En cas negatiu, indica quins principis creus que no compleix i indica quins canvis introduiries en el diagrama de classes per complir-los.

Solució:

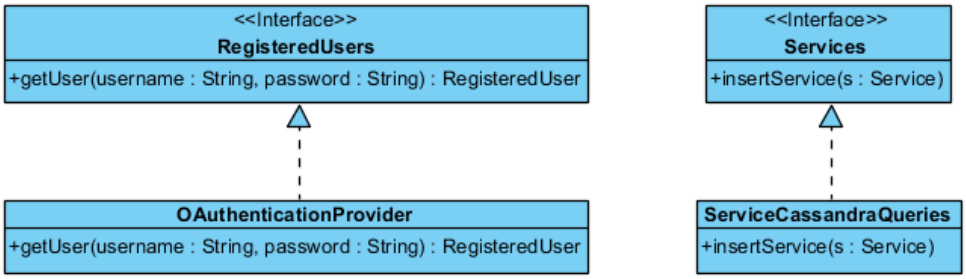
Aquesta solució es pot qüestionar si satisfà més o menys alguns dels principis de disseny. Però un principi que incompleix clarament és el principi d'Inversió de Dependències, ja que *RequestService,* que està al nivell d'abstracció de la lògica del domini, depèn de classes que tenen un nivell d'abstracció molt més baix, ja que tenen el nivell d'abstracció de la infraestructura; de fet, el d'un fabricant concret de bases de dades no relacionals i el d'un mètode popular d'autenticació.

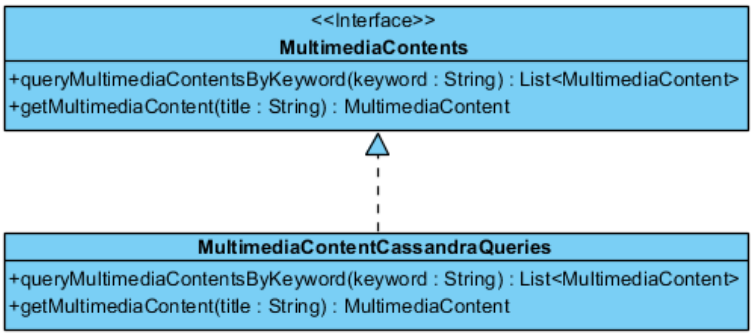
A nivell de diagrama de classes la solució consisteix a fer que *RequestService* depengui d'abstraccions i que els detalls també depenguin d'aquestes abstraccions. Una manera d'aconseguir-ho és crear una abstracció per a cada classe massa concreta:



En aquest cas hem creat 3 interfícies que representen, al nivell d'abstracció del domini, el conjunt d'usuaris, continguts multimèdia i serveis sol·licitats que tenim al sistema. Ara *RequestService* no depèn de detalls sinó d'una altra abstracció que està al seu mateix nivell d'abstracció.

Llavors, els detalls, que ja teníem, passen a dependre de les abstraccions. Una forma de dependència vàlida és la d'implementació d'interfície:


<<Interface>>
RegisteredUsers
+getUser(username : String, password : String) : RegisteredUser
OAuthenticationProvider
+getUser(username : String, password : String) : RegisteredUser
<<Interface>>
Services
+insertService(s : Service)
ServiceCassandraQueries
+insertService(s : Service)


<<Interface>>
MultimediaContents
+queryMultimediaContentsByKeyword(keyword : String) : List<MultimediaContent>
+getMultimediaContent(title : String) : MultimediaContent
MultimediaContentCassandraQueries
+queryMultimediaContentsByKeyword(keyword : String) : List<MultimediaContent>
+getMultimediaContent(title : String) : MultimediaContent

Exercici 5 (10%)

En l'exercici anterior hem plantejat una qüestió de disseny.

Vegem un fragment del pseudocodi de *RequestService*:

```
class RequestService
{
	...
	public MultimediaContent selectMultimediaContent (string title)
	{
		// ups! no sabem d'on obtenir la instància de
		// MultimediaContentCassandraQueries que necessitem:
		MultimediaContentCassandraQueries queries = ???;
		MultimediaContent mc = queries.getMultimediaContent (title);
		return mc;
	}
	...
}
```

Vist l'snippet de codi anterior, no sabem d'on obtenir una instància de *MultimediaContentCassandraQueries.* Podríem convertir totes les operacions d'aquesta classe en operacions amb àmbit de classe (estàtiques) i usar-les directament des *RequestService.* Però ens preocupa que potser més endavant volguem suportar també la base de dades ElastiCache (i poder decidir en cada instal·lació si fem servir una o una altra base de dades); en aquest cas tindríem *RequestService* acoblat a la classe *MultimediaContentCassandraQueries* i seria complicat poder decidir la base de dades en el moment de la configuració del programari.

Proposa una solució que permeti desacoblar *RequestService* de les classes específiques de Cassandra. Aquesta solució ha d'estar en consonància amb la solució que has proposat per a l'exercici anterior. La solució ha d'explicar, a més, com *RequestService* aconsegueix la instància de

la classe que utilitzarà per consultar un contingut multimèdia a partir del seu nom, resolent així el dubte sorgit al pseudocodi de l'exercici anterior.

Solució:

En la solució de l'exercici anterior, ja hem modelat que *RequestService* només faci servir (invoqui) mètodes d'abstraccions, interfícies pertanyents a la classe de domini, com *MultimediaContents*, *RegisteredUsers,* i *Services*.

En algun moment, però, ens cal obtenir una instància d'aquest tipus. Si *RequestService* instancia ella mateixa les instàncies que necessita es tornaria a acoblar a les classes concretes de la capa de serveis tècnics:

```
MultimediaContents contents = new MultimediaContentCassandraQueries();
```

Volem, doncs, escollir la implementació de *MultimediaContents* (i altres) a utilitzar sense acoblar *RequestService* a aquesta implementació. Aquest és exactament el problema que resol el patró d'arquitectura Injecció de Dependències.

La solució consisteix a associar una instància de *MultimediaContents,* una de *RegisteredUsers* i una de *Services* a *RequestService*. Baixant al nivell de codi, *RequestService* tindrà un atribut per a cadascuna d'aquestes instàncies. Llavors, en lloc d'inicialitzar la mateixa *RequestService* aquests atributs, li injectarem aquestes dependències des de fora. Proposem utilitzar la variant en què s'injecten a través del constructor:

```
class RequestService
{
      private MultimediaContents contents;
      private RegisteredUsers users;
      private Services services;

      public RequestService (
            MultimediaContents cs,
            RegisteredUsers us,
            Services ss)
      {
            this.contents = cs;
```

```
            this.users = us;
            this.services = ss;
      }

      public void identifyUser (string email)
      {
            RegisteredUser ru = users.getRegisteredUser (email);
            ...
      }
}
```

Hem mostrat el disseny resultant i com es faria servir una de les abstraccions des d'una de les operacions de la classe.