Projecte de final de carrera

Memòria

Enginyeria Informàtica

Consultor : Javier Ferró Garcia

Generació automàtica d'una solució JEE (JPA + EJB + JSF) basada en el framework AndroMDA .

Carles Patiño Dominguez, Gener de 2009

1 RESUM

Tot i les simplificacions introduïdes en les darreres versions, la construcció de solucions JEE no és una tasca fàcil i requereix un bon coneixement de més d'un framework especialitzat en àrees com la persistència o les tecnologies de presentació.

Per reduir aquesta complexitat existeixen bàsicament dos corrents. L'una consisteix a crear marcs de treball de nivell superior que n'englobin d'altres més específics. L'altre opta per la generació automàtica de codi a partir de models.

En aquest projecte s'utilitzarà el framework AndroMDA per construir un programari que permeti generar automàticament a partir d'un model UML simple una aplicació JEE.

Es tractarà doncs d'especificar els elements UML amb els quals es dibuixaran els models. Després, es dissenyarà l'arquitectura JEE que haurà de ser generada. Finalment, s'implementarà el programari necessari per tal de convertir un model UML dibuixat segons les especificacions en el conjunt de classes, fitxers de configuració i altres elements de programari que conformen una solució JEE.

Àrea del PFC : JEE2

Paraules clau : AndroMDA, Enterprise Java Beans, Generació de codi, Model Driven Architecture.

2 INDEX

3 JUSTIFICACIO I PLA DE TREBALL

3.1 Introducció

És indubtable que JEE és una plataforma de referència dins del món del desenvolupament Web de tipus empresarial. Les successives versions l'han anat enriquint amb noves APIs o especificacions : JSP, JSF, JAAS, JPA, EJB, JPA, JMS etc... Aquesta riquesa, però, ha provocat que en moltes ocasions desenvolupar solucions JEE des de no res sol ser complex i pot suposar uns costos considerables per les empreses. Per aquesta raó, entitats independents de SUN, han proposat frameworks propis amb l'objectiu de simplificar i, doncs, de fer més rendibles els projectes de desenvolupament JEE. A tall d'exemple, podem citar els següents frameworks utilitzats abastament tant per millorar com per facilitar el desenvolupament : Acegi (Seguretat) , Spring (negoci) , Struts (vista web), hibernate (persistència), jUnit (testing).

L'èxit reconegut d'aquestes propostes ha fet que SUN hagi incorporat dins de JEE solucions similars (e.g. Java Server Faces) o bé, millores que s'hi inspiren (e.g. a diferència de les versions 1 i 2, els Enterprise Java Beans versió 3 utilitzen ara POJO, és a dir objectes Java clàssics; les noves APIs de persistència o JPA han tingut en compte tècniques emprades per Hibernate). D'aquesta forma, des del punt de vista de les empreses que construeixen programari, la utilització d'aquests frameworks agilitza i optimitza el desenvolupament de solucions empresarials en proposar peces de programari altament reutilitzables, esdevingudes estàndards de facto, provades, documentades, amb millores constants i a baix cost.

Si bé el conjunt de frameworks esmentat ha suposat un argument a favor per l'acceptació i l'evolució de JEE com a plataforma de desenvolupament, la interacció dels diferents frameworks necessaris per crear una solució completa no resulta sempre evident i requereix un cert grau de formació i experiència. Així doncs, s'han definit diferents estratègies amb l'objectiu de facilitar el desenvolupament, sistematitzant i automatitzant aspectes que es repeteixen en tots els projectes JEE. (e.g. implementació del patró Model-Vista-Controlador)

A nivell empresarial, l'objectiu de les esmentades estratègies és ben clar:

- Estandardització de processos de desenvolupament : regles de codificació, arquitectures.
- Evitar tasques de programació repetitives i que solen ser font d'errors. A més, resulten poc rendibles per l'empresa i poc atractives pels desenvolupadors.
- Reutilització d'arquitectures i no només de classes.
- Millora de la qualitat : el codi final produït no depèn tan de la inspiració o del nivell dels programadors. S'incrementa el control del codi generat.
- Es millora el control sobre els temps de desenvolupament. D'aquesta forma es pot avaluar amb més fiabilitat l'abast del projecte en termes de cost i temps.
- En automatitzar part de la implementació, s'incrementa la productivitat i la competitivitat.

D'un costat, trobem frameworks que cobreixen per sí sols la majoria dels aspectes que constitueixen una solució de programari empresarial : capa visual, capa de negoci i capa de persistència. Un exemple podria ser GRails. Aquest integra alguns dels frameworks esmentats anteriorment però facilita enormement la feina dels arquitectes i dels desenvolupadors. Darrerament, aquestes han assolit un punt de

maduració suficient per començar a ser preses en consideració en la concepció de programari empresarial. Si bé poden permetre un desenvolupament altament eficaç i efectiu, en alguns casos poden resultar insuficients quan es tracta d'implementar solucions especifiques o quan l'empresa es vol desmarcar de la competència, oferint solucions de valor afegit, diferenciades i més adaptades a situacions particulars.

De l'altre, trobem programari que, a partir de metadades que descriuen el model, generen el codi que implementa les funcionalitats volgudes (MDA: Model Driven Architectures). Aquí les solucions no són tan nombroses i, sovint, només consideren una part de la solució. Requereixen un coneixement més profund de les tecnologies utilitzades per la implementació però permeten un alt grau de flexibilitat i adaptabilitat a les necessitats.

Un dels objectius del present projecte de final de carrera serà doncs avaluar les possibilitats de la generació de codi per a crear solucions JEE. Es tractarà doncs d'implementar una sèrie de tècniques, basades en la generació de codi, que permetin automatitzar tasques de desenvolupament repetitives però bàsiques en qualsevol projecte.

3.2 Descripció del projecte

El projecte proposat vol estudiar una possible solució de construcció d'aplicacions JEE basada en la generació automàtica de codi. A partir d'una definició conceptual del model, dels requeriments i de les funcionalitats d'una aplicació donada es tractarà de crear els components de implementació necessaris per l'aplicació final i.e. base de dades, classes entitats, classes d'accés a dades, regles de negoci, tractament formularis. Les regles de transformació que permetran convertir la descripció conceptual en una implementació concreta vindran configurades per un conjunt de plantilles. Seran doncs les esmentades plantilles que definiran l'arquitectura de l'aplicació. Seguidament, un programari especialitzat serà l'encarregat de portar a terme la transformació. En alguns casos, els components generats automàticament no seran totalment funcionals i caldrà completar-los manualment. Finalment, es faran servir les eines habituals per convertir els fonts obtinguts en les llibreries i descriptors necessaris per la posada en producció.

Si bé es podrien construir des de no res els elements necessaris, s'ha trobat més judiciós partir d'una base ja existent i treballar només aquelles parts que es poden considerar més especifiques d'una solució JEE particular.

Concretament, pel projecte proposat, s'utilitzarà el framework AndroMDA. Bàsicament, es tracta d'una eina que permet generar qualsevol tipus de sortida textual a partir dels elements següents:

- fitxers de tipus XMI, estàndard XML per expressar models UML.
- un motor de transformació.
- plantilles definides per l'usuari que determinen el contingut dels elements generats.

Per la definició dels models UML en format XMI es poden utilitzar algunes de les eines de dibuix més usuals (entre elles MagicDraw). També, usualment, el motor de transformació utilitzat és l'Apache Velocity. Addicionalment, AndroMDA ja proporciona les plantilles (o cartridges utilitzant els termes de l'eina) per construir una aplicació JEE. Ara bé, si ens posem en la situació esmentada en la introducció, pot resultar interessant estendre o reescriure les plantilles existents per adaptar-les a les especificacions de projectes particulars. El procés queda resumit en el següent esquema :

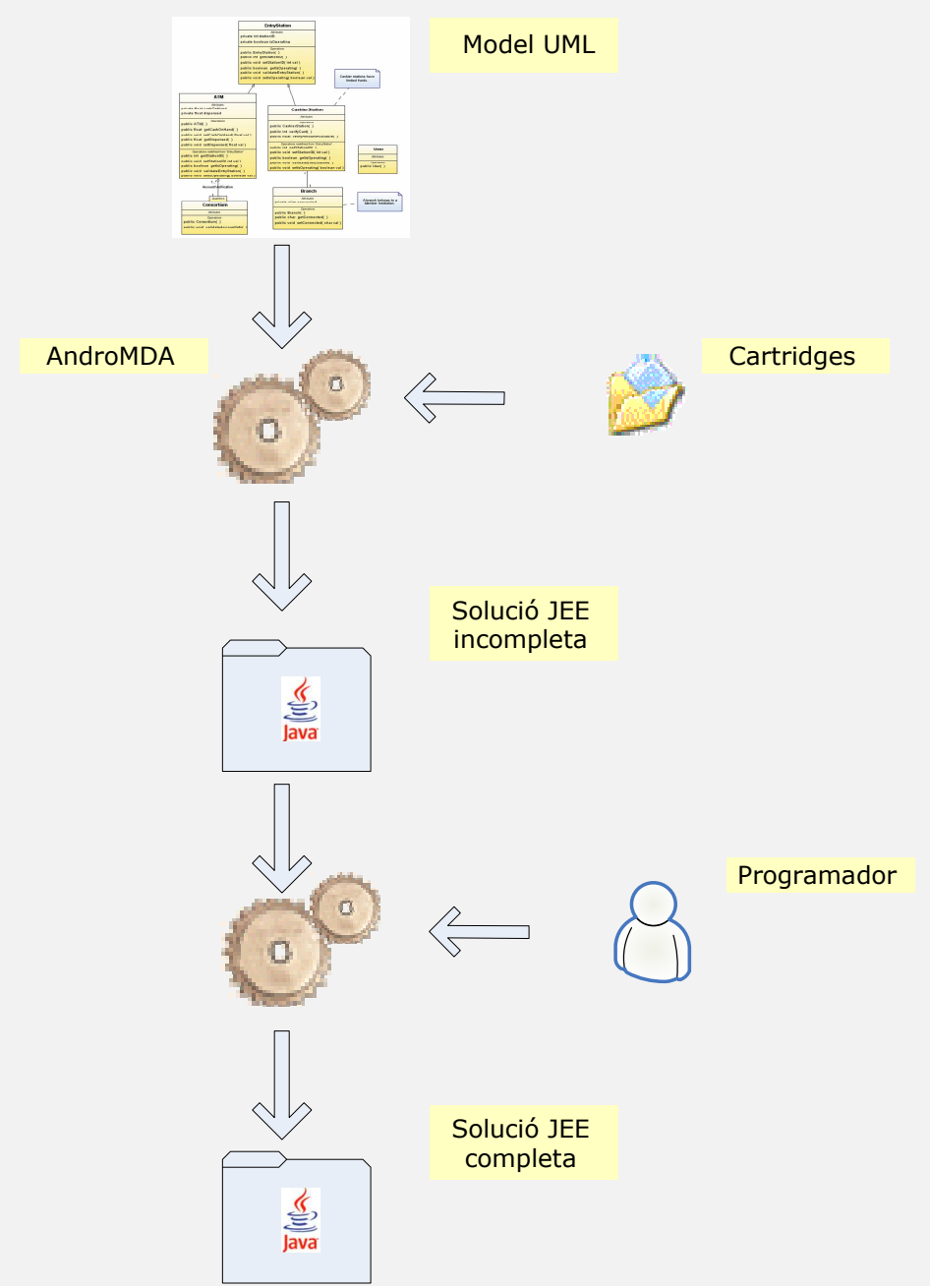


Figura 1 : Procés de generació de codi

Així doncs, el centre del projecte serà, d'una banda, la definició d'una arquitectura JEE completa, de l'altre, en el marc de AndroMDA, crear els estereotips UML i els cartridges necessaris per possibilitar la generació automàtica de l'arquitectura escollida. Per aquesta s'utilitzaran alguns dels frameworks més usuals dins del món JEE: Java Server Faces (JSF), Enterprise Java Beans (EJB) i Java Persistence API (JPA). Les versions o implementacions concretes triades són:

- Implementació IceFaces de JSF. En relació a la implementació de referència i a d'altres, integra tècniques Ajax per millorar el funcionament dels components JSF clàssics. A més, estan ben documentades, existeix una comunitat molt activa i s'integren amb la majoria d'entorns de desenvolupament : Eclipse, NetBeans,...

- EJB 3.0, darrera versió dels EJB que simplifica el model de la versió anterior. Si bé no és una solució indiscutible ofereix funcionalitats que no resulten possibles en d'altres tecnologies similars, com ara, el framework Spring. (e.g. permet accés remot a objectes)
- S'ha preferit utilitzar Hibernate conjuntament amb JPA en comptes de Hibernate únicament. En efecte, les possibilitats que ofereix aquesta combinació en el cas de treballar amb un servidor d'aplicacions semblen prou interessants (e.g. gestió de les transaccions per anotacions).

Esquemàticament, les diferents tecnologies que s'utilitzaran per definir l'arquitectura es mostren en la següent il·lustració:

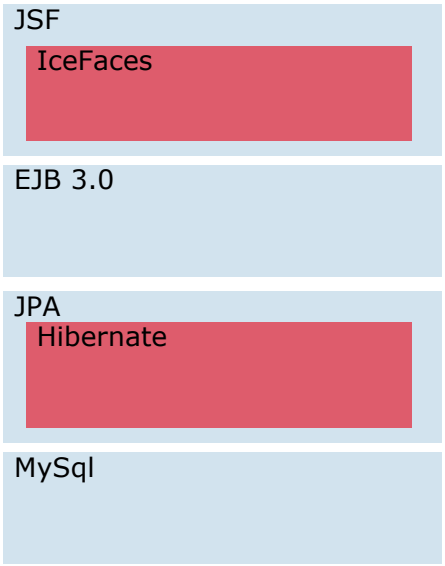


Figura 2 : Tecnologies utilitzades

3.3 Funcionalitats

Donat el tipus de projecte proposat, la funcionalitat principal que s'espera obtenir és la generació automàtica de les diferents capes que composen una solució JEE. Més concretament, s'espera obtenir un producte que, integrat dins del framework AndroMDA, generi de forma automàtica les següents parts de programari:

- Capa de persistència JPA/Hibernate:
  - Entitats : s'inclouen les classes d'entitats pròpiament dites així com la classes que modelen les associacions entre entitats.
  - Classes d'accés a dades (DAO) : modificació, creació, supressió, recerca.
  - Mapatge entre objectes i model relacional (ORM).
- Capa de negoci EJB 3.0:
  - Validació entitats.
  - Interfícies regles de negoci.
- Capa de presentació JSF/IceFaces:
  - Navegació.
  - Gestió:
    - Recerques simples i avançades
    - Presentació tabular de les dades.

    - Paginació.
    - CRUD : Modificació, creació, supressió
    - Objectes de transferència (DTO)
- Representació relació entre taules:
    - Relació 1 a molts : capçalera línies.
    - Relació molts a 1 : desplegables
- Crida a les funcionalitats de la capa de negoci.

3.4 Objectius

El primer objectiu consisteix en obtenir un coneixement suficient de cadascun dels frameworks escollits per la realització del projecte és a dir JSF i IceFaces; EJB 3.0; JPA i Hibernate. Si bé no és l'objectiu principal, es vol provar de comprendre quines són les seves virtuts i limitacions i, arribat el cas, poder comparar-les i situar-les respecte a altres frameworks amb funcionalitats similars.

Després, es tractarà d'integrar els frameworks esmentats dins d'una arquitectura adient. Degut a la complexitat de les tecnologies utilitzades, resulta molt necessari i útil concretar els aspectes arquitecturals que han de garantir la cohesió dels diferents components que entren en joc. Per aquest objectiu, es prendran en compte els patrons arquitecturals més usuals i clàssics dins de l'àmbit de JEE.

Seguidament, en comptes d'implementar una solució JEE a un cas concret, he pensat interessant introduir-me en aspectes relacionats amb la generació automàtica de codi per posar en pràctica l'arquitectura obtinguda. Es tractarà, per tant, d'estudiar els beneficis i limitacions de la construcció automàtica de codi. En el marc d'aquest projecte, es farà servir el framework de generació de codi AndroMDA que caldrà aprendre a utilitzar amb la profunditat suficient per permetre la generació del codi necessari per implementar una arquitectura altre que les facilitades per la pròpia eina.

Finalment, l'últim objectiu consistirà en validar i avaluar tant l'arquitectura com el procés de generació de codi definint un projecte tipus i aplicant les tècniques desenvolupades per implementar-lo.

De forma resumida els objectius que es volen assolir són:

- Aprendre a utilitzar els frameworks JEE JSF, EJB i JPA.
- Definir una arquitectura que integri els frameworks esmentats.
- Integrar dins del framework AndroMDA, la generació automàtica de solucions basades en l'arquitectura definida.
- Implementar un projecte tipus utilitzant els resultats dels punts anteriors.

3.5 Planificació temporal

L'estratègia seguida per desenvolupar el projecte consisteix essencialment en obtenir una arquitectura funcional i provada i, després, crear els components necessaris d'AndroMDA per reproduir-la automàticament. En efecte, abans de treballar la part corresponent a la generació, cal validar prèviament l'arquitectura dissenyada amb els casos típics considerats. En cas contrari, és a dir, si no es fa una validació prèvia, els resultats de la generació seran inevitablement incorrectes i serà molt difícil solucionar-los en aquest nivell.

Una vegada feta la validació, es podran implementar els components d'AndroMDA necessaris per reproduir automàticament l'arquitectura buscada.

D'aquesta forma, la divisió en tasques prevista queda detallada en el quadre següent:

Descomposició estructural d'activitats (WBS)		
Codi activitat	Nom activitat	Durada esperada (dies)
01	**Inici del projecte**	**15**
01.01	Recerca línies bàsiques del projecte a realitzar	
01.02	Definició del pla de treball	
02	**Anàlisi i disseny**	**35**
02.01	Instal·lació programari	1
02.01.01	*Eclipse*	
02.01.02	*JEE + Hibernate + IceFaces*	
02.01.03	*MagicDraw*	
02.01.04	*MySql*	
02.01.05	*Maven*	
02.01.06	*AndroMDA*	
02.02	Estudi frameworks escollits	10
02.02.01	*JPA- Hibernate*	
02.02.02	*EJB 3.0*	
02.02.03	*IceFaces*	
02.02.04	framework AndroMDA	
02.03	Anàlisi i disseny arquitectura	12
02.03.01	*Persistència : classes entitats, classes DAO i mapatge objectes / relacional.*	
02.03.02	*Negoci*	
02.03.03	*Presentació : prototip formularis*	
02.04	Anàlisi components AndroMDA per donar suport a l'arquitectura escollida : artefactes, estereotips, cartridges.	12
03	**Implementació**	**42**
03.01	Implementació i validació de l'arquitectura	15
03.01.01	*Definició d'un model d'exemple simple amb les estructures preses en compte en l'arquitectura.*	
03.02	Implementació components AndroMDA	20
03.02.01	*Implementació artefactes, estereotips,* cartridges.	
03.03	Validació components implementats	7
03.03.01	*Definició de projecte de prova d'una complexitat suficient per avaluar la generació de codi amb els elements implementats.*	
03.03.02	*Validació dels resultats obtinguts*	
04	**Fase final**	**28**
04.01	Memòria	20
04.02	Presentació virtual	8

El diagrama de Gantt resultant és el següent :

Nombre de tarea	Duración	Comienzo	Fin
Inici del projecte	**11 días**	**mié 17/09/03**	**mié 01/10/03**
Pla de treball	11 días	mié 17/09/03	mié 01/10/03
Entrega PAC1	0 días	mié 01/10/03	mié 01/10/03
Anàlisi i Disseny	**27 días**	**jue 02/10/03**	**mié 05/11/03**
Instal·lació programari	1 día	jue 02/10/03	jue 02/10/03
Estudi frameworks	8 días	jue 02/10/03	dom 12/10/03
Anàlisi i disseny arquitectura	10 días	lun 13/10/03	vie 24/10/03
Anàlisi i disseny components AndroMD	9 días	sáb 25/10/03	mié 05/11/03
Entrega PAC2	0 días	mié 05/11/03	mié 05/11/03
Implementació	**30 días**	**jue 06/11/03**	**mié 17/12/03**
Implementació i validació arquitectura	11 días	jue 06/11/03	jue 20/11/03
Implementació components AndroMDA	14 días	vie 21/11/03	mié 10/12/03
Validació components implementats	5 días	jue 11/12/03	mié 17/12/03
Entrega PAC3	0 días	mié 17/12/03	mié 17/12/03
Finalització	**20 días**	**jue 18/12/03**	**mié 14/01/04**
Memòria	15 días	jue 18/12/03	mié 07/01/04
Presentació Virtual	5 días	jue 08/01/04	mié 14/01/04
Entrega PAC4	0 días	mié 14/01/04	mié 14/01/04

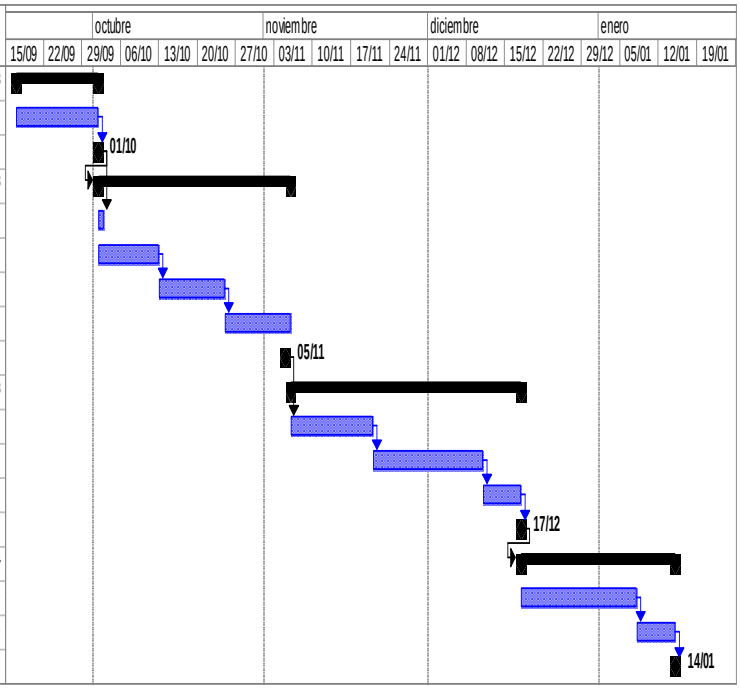


Figura 3 : Planificació

3.6 Entregables

Com a resultat de la fase 2, s'obtindran els elements d'anàlisi i disseny, tant de l'arquitectura desitjada com dels components d'AndroMDA que caldrà implementar per reproduir-la.

En teoria, els entregables més interessants han de provenir de l'assoliment de la fase 3. Es tractarà de tots els components AndroMDA necessaris per la generació de codi : artefactes, estereotips, cartridges. En aquest punt del projecte i degut al meu coneixement mínim de AndroMDA em resulta bastant difícil donar una llista detallada d'aquests elements. Addicionalment, s'entregaran els fonts i executables d'un projecte creat de forma automàtica.

Finalment, s'entregaran la memòria i la presentació virtual del projecte.

4 REQUERIMENTS

4.1 Arquitectures dirigides pel model (Model Driven Architecture – MDA)

MDA és un enfocament en el disseny i implementació de programari on el model deixa de ser una etapa dins del procés per assolir un paper central. Usualment, el model serveix de base per dissenyar i després implementar una solució de programari. Si bé conceptualment totes tres etapes estan fortament lligades, a la pràctica, al llarg de la vida del projecte, es produeixen diferencies que fan que, per exemple, la implementació model no correspongui amb el model. Aquest desfasament fa perdre valor i fiabilitat al model i, a més, dificulta el posterior manteniment del programari.

MDA distingeix quatre models principals :

- El CIM (*computation independent model*) correspon al model del negoci del sistema. Modelitza el domini del negoci i en recull els requisits principals. Ha de ser totalment independent de l'arquitectura de programari.
- El PIM (*platform independent model*) continua tenint un nivell d'abstracció alt però es modelen a nivell d'arquitectura les funcionalitats del sistema. La seva representació és independent de qualsevol tecnologia de programari concreta.
- El PSM (*Platform especific model*) finalment modela la implementació definida en el PIM. Aquest cop es tracta d'un model dependent de la tecnologia de destí. Per exemple es podria tenir un PSM per cadascuna de les capes d'una aplicació JEE.
- Un punt clau de MDA és que es passa d'un model a un altre de forma automàtica. En el pas final, a partir del PSM es genera el model de codi que simplement consisteix en el codi, descriptors de desplegament etc... necessaris per implementar completament l'aplicació. Segons el cas, el codi generat s'haurà de completar manualment.

A nivell empresarial aquest enfocament és enormement interessant en molts aspectes:

- En automatitzar la generació de codi es fomenta la reutilització i el saber fer de la empresa.
- Es redueix el temps de desenvolupament però mantenint bons nivells de qualitat.
- El procés de creació de programari està més controlat : permet reduir els costos i ajustar l'estimació de l'esforç necessari pel desenvolupament de nous projectes.

4.2 Frameworks MDA i generadors de codi

No és l'objectiu d'aquest treball fer un anàlisi de les eines MDA disponibles i de les seves possibilitats. Tot i així, en les recerques inicials s'ha consultat les característiques principals d'algunes de les eines MDA que es poden trobar dins de https://www.codegeneration.net/generators.php.

Si bé l'abast, possibilitats i nivell de maduresa dels frameworks consultats són molt variats es podrien agrupar en dos tipus principals :

- Eines que generen el codi corresponent a certes parts de l'arquitectura. A més, el codi generat no es pot adaptar a les necessitats pròpies : per exemple, es podria

citar MagicDraw. Aquest pot generar el codi corresponent a les classes modelades o als diagrames de seqüència dibuixats.

- Eines que generen aplicacions senceres a partir del model. Sense dubte les més interessants. Generalment basen la generació de codi en plantilles però ofereixen solucions Out of the Box. Bons exemples són : Taylor, (http://taylor.sourceforge.net/index.php), JAG (http://jag.sourceforge.net/), AndroMDA (http://www.AndroMDA.org/).

Inicialment, encara que fora de l'àmbit de MDA, es van prendre en consideració frameworks com Openjavax o Grails que simplifiquen notablement el procés de desenvolupament.

Finalment, i seguint la recomanació del consultor, m'he decantat per utilitzar AndroMDA.

4.3 AndroMDA

Parteix de models UML en format XMI (Extensió de XML per representar Models UML) dibuixats amb qualsevol eina CASE compatible.

En base a aquest model utilitza els anomenats cartridges per generar codi per una plataforma en particular. Bàsicament, un cartridge és un conjunt de diagrames i de templates que permeten generar codi de l'aplicació. Per exemple, trobem :

- *Bpm4struts* : Genera pàgines Struts a partir d'un model UML.
- *JSF* : Capa de presentació JSF.
- *EJB* : Genera EJBs d'entitat i de sessió.
- *Hibernate* : Permet generar una capa de persistència Hibernate.
- *Spring* : En conjunció amb el cartridge Hibernate, genera les capes de persistència i de negoci tipus Spring. La capa de presentació es pot generar amb el cartridge Bmp4Struts.

4.4 Requeriments

Amb els cartridges disponibles, AndroMDA permet la generació d'aplicacions per un bon nombre de frameworks, tots ells d'àmplia utilització. Així doncs, una possible orientació del projecte seria la implementació d'un projecte utilitzant els cartridges disponibles. Ara bé, el meu interès en aquest tipus d'eina és la de comprovar fins a quin punt són eficaces per la realització de projectes amb arquitectures especifiques. En efecte, usualment, les empreses que creen programari, a banda de la possible arquitectura de grà gruix imposada pel framework de torn, també especifiquen una arquitectura pròpia de grà fi.

Així doncs, la meva proposta consisteix en dissenyar i implementar un cartridge que permeti generar una arquitectura pròpia utilitzant frameworks ja existents.

Més concretament, els requeriments que el cartridge hauria de satisfer serien :

Requeriments sobre els diagrames

- Reduir el nombre de diagrames necessaris : es preveu un diagrama pel model i un segon per modelar els elements de presentació i serveis.

- Es descartaran diagrames de tipus dinàmics com diagrama d'estats o de seqüència. Per exemple, en el cas dels serveis es preferirà la implementació manual (molt més rica) a la possible generació automàtica.
- Minimitzar dins dels models els aspectes arquitecturals o d'implementació : per exemple, s'hauria d'evitar la necessitat d'afegir representacions dels objectes de transport.
- Els diagrames no estan orientats a EJB 3.0 ni a cap tecnologia en particular.
- Anotar el model d'entitats amb informació que pugui ser útil tant en la capa de persistència com en la capa de presentació. (constraints , validadors ...)
- Facilitar l'especificació de consultes de forma descriptiva, sense utilitzar OCL ni diagrames particulars.

Requeriments d'arquitectura

- El mateix cartridge ha de permetre generar totes les capes d'una aplicació : persistència, negoci i presentació.
- En relació al codi generat s'ha decidit que la plataforma de destí sigui EJB 3.0 (persistència i negoci) i IceFaces (Presentació)
- Els diagrames utilitzats no tenen per que ser una imatge de l'arquitectura de destí.

Requeriments d'arquitectura de grà fi

- Amb els diagrames s'ha de poder descriure l'estructura de les pàgines de la capa de presentació : per exemple agrupació de dades o pestanyes.
- La interfície d'usuari ha de correspondre al funcionament següent :

  - Inicialment, es mostra a l'usuari una llista de dades. Opcionalment, s'afegeix un requadre amb diferents camps que permeten a l'usuari filtrar la informació mostrada. També opcionalment, existirà la possibilitat d'ampliar el nombre de criteris per fer la recerca.

  - D'una banda, l'usuari disposa de botons per executar operacions de tipus CRUD, més concretament, creació, modificació. L'opció de modificació s'aplica a una fila en concret i, per tant, es mostra per cadascuna de les files. Al contrari l'opció de creació es mostra fora de la taula.

  - De l'altre, l'usuari pot tenir accés a mètodes de negoci. Com en el cas anterior, alguns d'ells s'aplicaran a una fila en concret i, per tant, es mostraran dins de la mateixa taula, d'altres es representaran en la seva part superior.
- Aquest workflow no ha de ser èxplicit en el diagrama.

5 DISSENY DE L'ARQUITECTURA

En aquest punt, es tracta de definir l'arquitectura de grà gruix del sistema. No s'entrarà doncs en les especificats introduïdes per cadascuna de les tecnologies concretes que s'utilitzaran. Efectivament, aquestes tecnologies permeten sovint adaptar-se a més d'un tipus d'arquitectura i es corre aleshores el risc de fer-la dependent dels elements que la composen.

5.1 Organització en capes

Un dels primers objectius de l'arquitectura és reduir la complexitat del sistema que es vol desenvolupar. Amb aquest objectiu, l'estratègia consistent en organitzar el sistema en un conjunt de capes relacionades ha donat bons resultats no només en l'àmbit de les aplicacions empresarials.

Més concretament, ens referim al patró d'arquitectura *Layer*. L'aplicació d'aquest patró consisteix en dividir l'estructura del sistema en vàries capes lògiques. L'objectiu seguit és aplicar un dels principis bàsics de la orientació a l'objecte, i.e. alta cohesió i baix acoblament. Així doncs, en una estructura en capes, cadascuna d'aquestes ha de tenir responsabilitats clarament definides i diferenciades de la resta. L'acoblament només es produeix entre la capa immediatament superior i la immediatament inferior.

El nivell de detall, el nombre de capes i les seves responsabilitats així com la seva denominació depèn dels diferents autors encara que totes les propostes són bàsicament molt similars. En aquest treball, s'utilitzarà la divisió clàssica utilitzada per molts autors (e.g. *Patterns of Enterprise Application Architecture* de Martin Fowler) :

- Capa de lògica de negoci o domini
- Capa de presentació
- Capa d'integració

Usualment, les capes esmentades es consideren dins d'una mateixa capa anomenada Middle Layer que es situa entre una capa client, constituïda per exemple per un navegador i una capa de recursos (EIS) constituïda principalment pel SGBD.

Ens interessarem sobre tot a la capa de negoci degut a que per la resta de capes s'han arribat a estàndards de facto i les possibles decisions de disseny semblen més clares.

5.1.1 Lògica de negoci o domini

És dins d'aquesta capa, i només en aquesta, on s'implementa la lògica de negoci. En una arquitectura de dues capes client-servidor, aquest capa no existeix de forma explicita. De fet, es troba dispersa entre la part visual i la mateixa base de dades on part de la lògica s'implementa en forma de vistes, procediments emmagatzemats disparadors etc...

Domain Layer

Usualment, en un model orientat a l'objecte, una part de la lògica del negoci es defineix gràcies a un conjunt de classes relacionades que constitueixen el model de domini. Aquesta estratègia resulta, però, insuficient per definir completament les

regles de negoci. Com indica Fowler en el llibre mencionat anteriorment, són possibles varies estratègies

Una consisteix en completar el model anterior implementant la lògica del negoci dins dels mètodes de les classes. Així doncs, la capa de presentació podria treballar directament amb els objectes del negoci.

Domain Layer i Service Layer

Una altra possible solució consisteix en introduir dins de la capa de negoci una sub-capa de serveis de forma que la capa de negoci queda formada pel model de domini (Domain Model) i una capa de serveis (*Service Layer*). Aquesta aplica el patró Façade i actua com a una interfície que simplifica l'accés al model de domini i n'abstreu les complexitats. Entrant més en detall, l'atribució de responsabilitats entre aquestes es pot fer aleshores de dues formes diferents.

En una primera possibilitat, el model de domini està composat per un conjunt de classes (únicament els seus atributs i no els mètodes) i les associacions entre elles. La capa de serveis es composa d'una sèrie de mètodes que completen la lògica de negoci.

En la segona possibilitat, Fowler diferencia entre la lògica de negoci i la lògica de l'aplicació (o worflow d'aplicació). La primera correspon a les regles de funcionament elementals, bàsiques i parcials del model mentre que la segona té una visió global de l'aplicació i és la responsable d'integrar tots els elements del model per assolir les funcionalitats de l'aplicació de forma completa.

La service layer també serà la responsable d'implementar la lògica d'aplicació. La decisió és força discutible però, a la pràctica si es consideren funcionalitats que prenen en compte més d'una classe, resulta més clar i fàcil definir la lògica corresponen dins d'un altre nivell que creant fortes dependències dins del model.

En el marc del present projecte, s'escollirà aquesta última estratègia. En efecte, tenir una capa de serveis permet ocultar part de la complexitat del model a la capa de presentació. A més, evita que la capa de presentació accedeixi directament a la lògica implementada en el model de domini i que s'hi implementi part de la lògica de l'aplicació. Addicionalment, serà ideal per implementar altres tipus de serveis com la gestió de les transaccions o de la seguretat.

5.1.2 Presentació

Per la capa de presentació s'optarà pel ja clàssic patrò Model-Vista-Controlador (MVC). En aquest cas, la decisió resulta bastant clara atès que per la capa en qüestió es tracta d'un patró recomanat per Sun dins de les Java-BluePrints i les tecnologies de presentació disponibles dins de la plataforma EE5 la implementen.

Breument, el patró consta de tres parts :

- El model representa l'estat de la vista.
- La vista interactua directament amb l'usuari i mostra visualment la informació emmagatzemada dins del model. També recull les peticions fetes per l'usuari.
- El controlador és el responsable d'actualitzar la vista quan canvia el model i de tractar les peticions d'usuari fent les crides adients a la lògica de negoci.

5.1.3 Transferència de dades entre capes

Amb una capa de serveis, entre la presentació i el model, apareix la necessitat de trobar una forma de traspassar la informació entre les dues. En aquestes situacions, una possible solució és la utilització d' objectes de transferència, o Data Transfert Objects si s'utilitza la denominació usual. La idea és definir objectes amb un nivell de detall inferior al del model de domini però sense definir cap mena de lògica. Actuen com a simples representants del model de domini. Segons les necessitats o les decisions de disseny poden ser una rèplica dels objectes de domini però sense cap lògica de negoci o oferint només mètodes accessors o bé una versió simplificada del model i adaptada a les necessitats de la capa de presentació.

La utilització del patró Data Transfert Objects (DTO) és molt discutible i varia segons els autors. Dins de les especificacions anteriors a EJB3, els objectes de transferència (o value objects) eren abastament utilitzats. En efecte, en una arquitectura distribuïda i donat que els EJBs d'entitat no eren serialitzables, els DTOs permetien traspassar la informació relativa al model entre capes. També, en no tenir la complexitat del model permetien un guany en rendiment.

Ara bé, amb EJB 3.0 la mateixa Sun considera la utilització del DTO com un anti-pattern. D'altres autors com Christian Bauer i Gaving King dins del llibre *Java Persistence with Hibernate* (2008), aconsellen treballar directament amb el model.

Tot i això, d'altres autors n'aconsellen la utilització en algunes situacions. Per exemple, Fowler aconsella la seva utilització en arquitectures distribuïdes quan es tracta de passar múltiples instàncies d'objectes en una sola crida: *Use a Data Transfer Object whenever you need to transfer multiple items of data between two processes in single method call.*

L'autor Floyd Marinescu dins del seu llibre *EJB Design Pattern*, distingeix tres variants per implementar la comunicació entre capes:

- *Domain Data Transfert Objects* : Es tracta de crear per cada objecte del model un objecte de transport amb les mateixes propietats però només oferint mètodes accessors i cap lògica de negoci. Cada objecte DTO en la capa de presentació és una còpia d'un objecte del model. Així doncs, l'estructura de DTOs que en resulta és dedueix directament del model i facilita les operacions de traspàs de dades en ambdós sentits. No obstant, entre d'altres, presenta l'inconvenient que acobla fortament la presentació amb el model i complica el manteniment del sistema en ser necessari tenir sincronitzades les dues estructures. A més també caldrà preveure la construcció de DTO a partir d'objectes del model i viceversa. Tant Fowler com Marisnescu aconsellen aleshores aplicar el patrò *DTO Factory* per centralitzar la lògica de creació dels objectes esmentats
- *Custom Data Transfert Objects* : Amb aquesta variant, ja no existeix una relació directa entre model i DTO. Els DTOs es defineixen segons les necessitats de la presentació. D'aquesta forma, model i presentació queden desacoblats. En relació a la variant precedent, el seu principal inconvenient rau en el traspàs de dades en sentit presentació – model en operacions d'escriptura. En només ser una visió parcial, la informació dins d'un DTO pot no ser suficient per crear o actualitzar correctament una entitat del model, en teoria més complexa. Com en el cas anterior, serà necessari aplicar el patró *DTO Factory*.
- *Data Transfer HashMap* : Amb models de domini complexes, la capa responsable del tractament dels DTO pot ser un autèntic problema. Es pot aconseguir reduir la complexitat d'aquesta capa o simplement eliminar-la

completament utilitzant la variant on la transferència de dades entre capes es fa mitjançant col·leccions de tipus *map* que associen el nom d'una propietat a un valor concret. Aquesta tècnica és més fàcil d'implementar i facilita enormement el manteniment. En efecte, cadascuna de les entitats del model pot ser responsable de crear una representació pròpia en forma de mapa que relaciona el nom d'una propietat amb el seu valor. La capa de presentació només haurà d'escollir els valors que hagin de ser tractats al seu nivell. L'avantatge principal és la desaparició de la capa de factoria de DTOs i ja no es necessari ni tan sols crear un complex conjunts de DTOs per representar el model. No obstant, entre d'altres, presenta l'inconvenient de ser necessari establir clarament la nomenclatura dels noms de propietat. Efectivament, amb aquesta tècnica es perden alguns dels avantatges del tipatge fort (e.g. control de tipus durant la fase de compilació).

En aquest punt no es pren cap decisió sobre quina variant es farà servir. Al moment de dissenyar els aspectes relacionats amb la generació de codi s'escollirà aquella que millor s'adapti.

5.1.4 Integració

La responsabilitat de la capa d'integració és fer persistents els objectes del model de domini. El conjunt de patrons que s'apliquen en aquest nivell és important i d'una certa dificultat. Donat que els problemes que s'hi resolen són comuns i no depenen de casos particulars sembla més judiciós utilitzar frameworks de persistència ja existents. Degut a que de moment les bases de dades relacionals són les més abundants, aquests bastidors es basen en el mapatge objecte – relacional (O/R Mapping). Bàsicament consisteix en relacionar les entitats del model de domini amb una taula de la base de dades i cada propietat amb una columna.

El lligam amb la capa de negoci s'efectua aplicant el patró Data Access Object. (DAO Pattern). Aquest permet separar el model de domini dels mecanismes relacionats amb l'accés a les dades (selecció, inserció, modificació, supressió, gestió de les transaccions). En general es sol definir un objecte DAO per cada entitat del model de domini. Finalment, en funció de la solució escollida caldrà refinar l'arquitectura escollida.

5.2 Especificitats de l'arquitectura proposada amb EJB3.0 - JPA

5.2.1 Persistència EJB3/JPA

Entitats

Dins d'una aplicació JEE el més usual és utilitzar la Java Persistence API (JPA) per gestionar la persistència de les entitats del model de negoci. Es tracta d'una especificació i no d'una implementació. En cas d'utilitzar un servidor d'aplicacions la tecnologia concreta d'implementació dependrà del contenidor. En aquest treball s'utilitzarà JBOSS que, per defecte, fa servir Hibernate.

Concretament, la tècnica utilitzada per fer persistents les entitats consisteix en la utilització d'anotacions Java. Així, el mapping objecte-relacional es resol anotant les classes i propietats de les classes entitat. Per exemple, es pot definir quina taula de la base de dades correspon a una entitat o a quina columna correspon una propietat donada.

Durant la fase d'implementació s'entrarà en detall de com es reflecteixen conceptes ORM imprescindibles com poden ser les claus de negoci, claus compostes, relacions, herència, etc.. en la implementació de les classes d'entitats i les anotacions corresponents.

Per tant, dins de les classes d'entitat trobarem :

- Anotacions a nivell de classe.
- Propietats persistents anotades.
- Mètodes accessors (i.e. getters i setters)
- Mètodes de negoci i de validació. La seva inclusió dins de les entitats respon a un model objecte tradicional i queda ben entès de que en cap cas efectuen operacions directament amb la capa de persistència.

Accés a dades

Per accés a dades s'entén el conjunt d'operacions CRUD i de recerca que afecten a cada entitat i que són imprescindibles per fer persistents les entitats. Les funcionalitats requerides són :

- Recuperació de la base de dades de les entitats que satisfan una sèrie de requisits imposats per les regles de negoci.
- Dins de la base de dades, inserció, actualització, supressió de les entitats del model objecte.

Amb JPA, aquestes operacions es tradueixen dins de les classes d'accés a dades per crides a mètodes a la *EntityManager.* L'objecte *EntityManager* és central per la gestió de la persistència donat que és la responsable de tractar amb els objectes en memòria, els caches i la demarcació de les transaccions entre d'altres. És important notar que la instanciació de l'objecte *EntityManager* és responsabilitat del contenidor raó per la qual es recomana que dins de les classes d'accés a dades la referència s'obtingui per injecció de dependències.

A nivell d'arquitectura, aplicarem el clàssic patró de disseny Data Access Object. Per cada entitat, s'implementa una classe que encapsula totes les operacions anteriors i és la responsable de fer passar les entitats del model objecte al model relacional.

Sovint, alguns autors condicionen l'aplicació del patró DAO a la del patró DAO Factory. Aquest té com a objectiu desacoblar el model objecte de la implementació concreta de la persistència. Ara bé, amb la tecnologia utilitzada en el present projecte, la factoria ja no és necessària en la majoria dels casos. En efecte, primer JPA permet independència respecte al framework de persistència concret utilitzat, i, després, aquest framework independitza el model del motor de base de dades realment utilitzat. Podríem trobar interès en utilitzar DAO Factory en cas de que l'aplicació hagués de ser compatible amb bases de dades amb estructures diferents.

Finalment, doncs, per l'accés a dades, l'arquitectura proposada només tindrà en compte el patró Data Access Object. En el contexte de EJB3 , cada classe DAO serà un EJB de sessió sense estat on la referència a l'objecte *EntityManager* s'obté per injecció de dependències.

Capa de serveis EJB3

Per la capa de serveis l'opció més natural és la utilització d'EJBs de sessió. Es tracta de classes POJO amb anotacions ad hoc. Dins d'aquetes classes serà necessari obtenir instàncies d'objectes DAO. Aquí també, aquestes instanciacions seran efectuades automàticament pel contenidor gràcies a la injecció de dependències.

També, serà en aquest nivell i sempre amb les anotacions adients que s'implementarà la demarcació de les transaccions.

Finalment, a diferència de les versions anteriors de EJB no serà necessari aplicar el patró *Service Locator* per permetre a la capa de presentació fer referència als serveis. Simplement, la nova especificació de EJB permet fer el lookup de serveis, una altre vegada, fent ús de la injecció de dependències a través d'anotacions.

Tots aquests aspectes s'estudiaran més en detall durant la fase d'implementació.

5.2.2 Presentació Java Server Faces

Els principals elements del framework JSF són:

- Formularis Web : S'utilitzen els tags propis de JSF. Es defineixen els components JSF que formen part del formulari.
- Backing-bean : Es tracta d'una classe Java que compleix la norma javaBean. Pot controlar un o més formularis Web. Implementa propietats que corresponen a l'estat de la vista així com els mètodes d'accés corresponents, mètodes per manejar les accions i els events iniciats en la vista.
- Faces-config.xml : S'hi declaren els backing Beans indicant el seu nom, la classe Java corresponent i el tipus d'accés que es vol utilitzar (aplication, session o request). També s'hi defineixen les regles de navegació de l'aplicació.

Java Server Faces aplica el patró Model-View-Controller de la següent forma:

- *View* : Bàsicament, està composada per l'arbre de components JSF, renderers, convertidors, validadors.
- *Controller* : Es composa del FacesServlet, del fitxer de configuració Faces-config.xml, i dels manejadors d'accions i events. De forma resumida, FacesServlet rep les peticions fetes des del navegador Web i, gràcies al fitxer Faces-config.xml les redirigeix al bean adient. Es criden els mètodes escoltadors d'events i d'accions implementats dins dels beans.
- *Model* : Dins dels mètodes escoltadors s'accedeix al model de negoci. La forma d'accedir-hi depèn de l'arquitectura escollida per implementar el model de negoci. En el cas del projecte presentat, donada l'arquitectura escollida, l'accés al model es fa a través d'una capa de serveis implementada amb EJBs de sessió sense estat i pel traspàs de dades entre la capa de presentació i serveis s'utilitzen objectes de transferència.

El diagrama següent resumeix l'explicació :

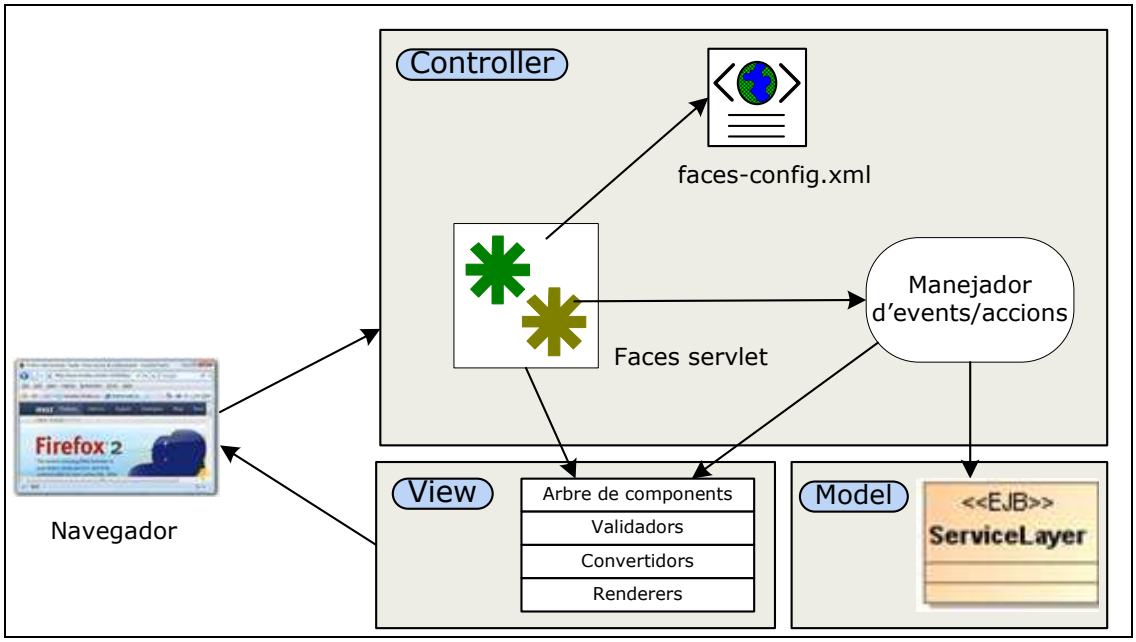


Figura 4 : Model-Vista-Controlador aplicat a JSF

5.2.3 Diagrama de seqüència

El diagrama següent mostra de forma resumida les interaccions principals entre capes.

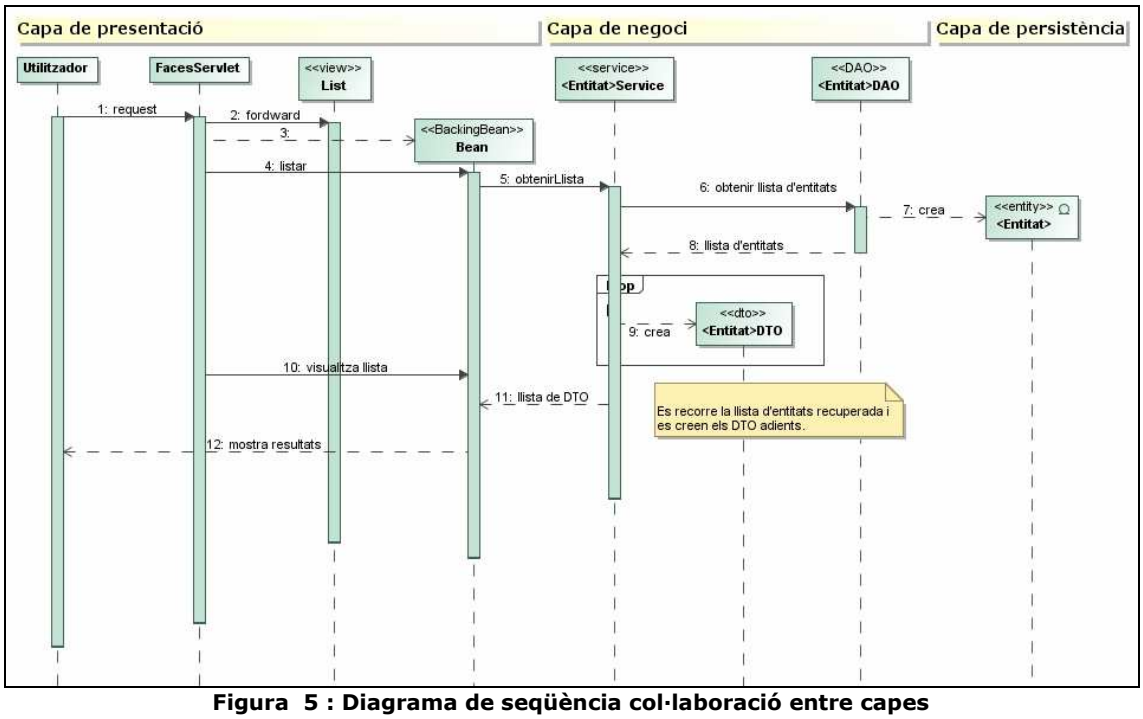


Figura 5 : Diagrama de seqüència col·laboració entre capes

6 DISSENY DELS DIAGRAMES

Una vegada establerta l'arquitectura, es tracta de definir els diagrames a partir dels quals caldrà construir l'aplicació. En aquest punt es busca que els diagrames siguin el més independents possibles de la tecnologia final d'implementació.

6.1 Diagrama del model de domini

Bàsicament, un model de domini es composa d'un conjunt de classes, propietats, mètodes de negoci i associacions. Naturalment, el diagrama UML adient per representar-lo és un diagrama estàtic de classes. Aquest, però, resulta insuficient per expressar completament un model de domini. Per aquesta raó el diagrama de classes bàsic es completarà amb estereotips, mecanisme d'extensió d'UML. Aquest consisteix en adjuntar a qualsevol artefacte d'UML, (classe, propietat, associació...) una anotació entre els caràcters « i ». Addicionalment, un estereotip es pot completar amb un o més tags associats a valors concrets.

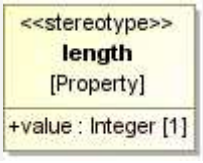
6.1.1 Entitats

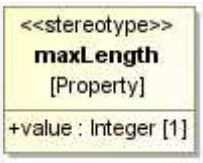
Les entitats representen els objectes de negoci. De forma usual és representen com classes amb l'estereotip «Entity» Generalment, una classe Entity correspondrà a una taula en la base de dades. Per facilitar la introducció del model, per defecte, el nom de la taula correspondrà al nom de la classe. Usualment, però, els gestors de persistència permeten a l'usuari especificar un nom de taula concret. Així doncs, opcionalment, «Entity» es podrà completar amb els tags *tableName* i *tableSchema*. Si bé aquests no tenen cap sentit dins d'un model de domini permetran canviar si necessari el valor per defecte donat a les taula i a l'esquema on ha de ser creada.

6.1.2 Propietats

A nivell del model, les propietats de les classes d'entitat es caracteritzen per la seva visibilitat i pel seu tipus. Els diferents tipus d'estereotips previstos aplicables a les propietats són els següents:

«required»	
Objectiu	La propietat a qui s'aplica ha de estar sempre informada.
Tipus compatibles	Tots els tipus
Estereotip	

«length»	
Objectiu	Especificar la llargada d'una cadena
Tipus compatibles	String
Estereotip	
Tag	**Descripció**
Value	Valor que correspon a la llargada.

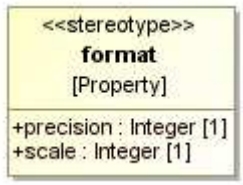
«maxLength»	
Objectiu	Especificar la llargada màxima d'una cadena
Tipus compatibles	String
Estereotip	
Tag	**Descripció**
value	Valor que correspon a la llargada.

«upperCase»	
Objectiu	La cadena ha d'estar en majúscules.
Tipus compatibles	String
Estereotip	

«lowerCase»	
Objectiu	La cadena ha d'estar en minúscules.
Tipus compatibles	String
Estereotip	

«pattern»	
Objectiu	La cadena verifica un patró especificat per una expressió regular.
Tipus compatibles	String
Estereotip	
Tag	**Descripció**
regExpr	Expressió regular

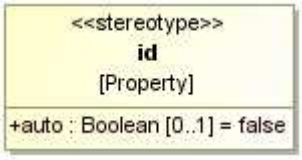
«range»	
Objectiu	Indicar el interval vàlid.
Tipus compatibles	Integer, Decimal
Estereotip	
Tag	**Descripció**
min	Valor mínim
max	Valor màxim

«format»	
Objectiu	Indicar el nombre de dígits enters i decimals.
Tipus compatibles	Integer, Decimal
Estereotip	

Tag	Descripció
precision	Nombre de dígits de la part entera.
scale	Nombre de dígits de la part decimal.

6.1.3 Claus de negoci

Si bé la noció de clau de negoci no s'utilitza dins del model objecte, serà necessari que dins de la representació d'una entitat s'indiqui quina o quines propietats corresponen a la clau primària de la taula que correspon a l'entitat.
Per aquest propòsit es defineix l'estereotip «id» aplicat a la o les propietats que corresponen a una clau de negoci dins del model objecte i que es traduiran per una clau primària en el model relacional.
Pels casos on no és pugui fer servir cap clau natural com a clau de negoci es podrà demanar al gestor de persistència de generar-la automàticament. En aquest cas només existirà una propietat amb l'estereotip «autoid» i serà del tipus Integer.

«id»	
Objectiu	Indicar que una o varies propietats composen la clau de negoci de l'entitat.
Tipus compatibles	Tots els tipus. Si el tag auto val true, el tipus ha de ser Integer
Estereotip	

Tag	Descripció
auto	Si true el valor de la propietat serà calculat automàticament per gestor de persistència.

6.1.4 Mètodes de negoci

Com s'ha dit en l'apartat corresponent a l'arquitectura, la lògica de negoci resideix dins de les mateixes entitats. La forma natural de representar-la serà a través de mètodes de classes. El diagrama ens permetrà indicar el valor de retorn, nom del mètode i signatura :

+Nom del mètode(parametre : *Tipus[0..*]*) : *Tipus retorn*

Addicionalment, per indicar que el mètode pot llençar una excepció, es pot associar el mètode amb una classe amb l'estereotip «exception». Com es mostra en la taula següent, s'afegeix un tag que permet definir el missatge associat a l'excepció.

«exception»	
Objectiu	Indicar que un mètode de negoci pot generar una excepció.
S'aplica	Mètodes d'entitats

Estereotip	

Tag	Descripció
defaultMessage	Missatge en clar associat a l'excepció.

6.1.5 Recuperació d'instàncies d'entitats de negoci

Normalment, dins de la capa de negoci seran necessaris mètodes responsables de recuperar col·leccions d'instàncies d'entitats fetes persistents a la base de dades. Si bé des d'un punt de vista arquitectònic aquests mètodes s'implementaran dins de les classes DAO, per no carregar el diagrama, aquests es dibuixaran relacionats amb les entitats.
Així doncs, amb una dependència es podran associar les entitats amb una classe estereotipada amb «finder».

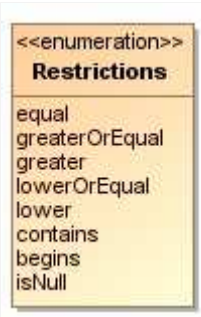
«finder»	
Objectiu	Especificar un servei de recuperació d'una col·lecció d'entitats persistents a la base de dades.
S'aplica	Classes entitats
Estereotip	

Tag	Descripció
orderBy	Noms de la propietat o propietats separat per ',' serveix de com a criteri d'ordenació.

L'associació indica que existeix un servei que permet recuperar una col·lecció amb totes les instàncies de l'entitat amb la que es fa l'associació. Addicionalment, es poden afegir propietats a la classe «finder» per tal de definir paràmetres i restriccions per acotar la recerca. Aquestes propietats tindran l'estereotip «param» amb el que es podran definir restriccions a les propietats de l'entitat.

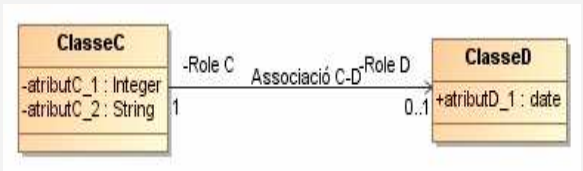
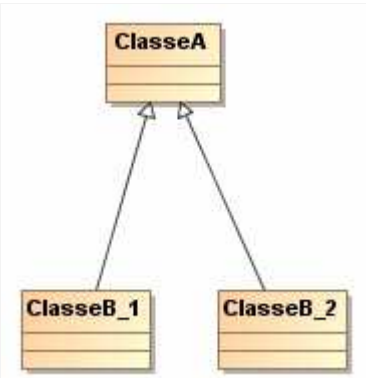
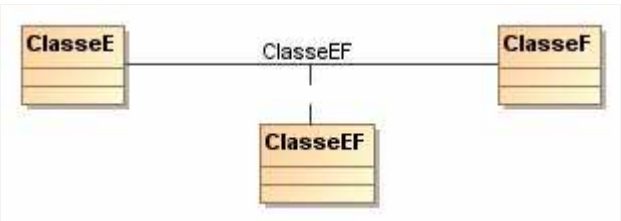
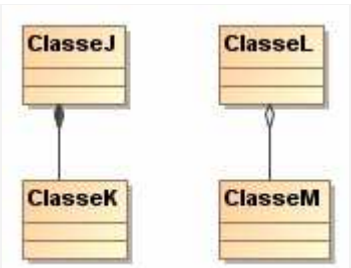
«param»	
Objectiu	Especificar un servei de recuperació d'una col·lecció d'entitats persistents a la base de dades.
S'aplica a	Propietats de classes «finder»
Estereotip	

Tag	Descripció
Bind	Propietat de l'entitat del model on s'aplica la restricció.

Restriction	Restricció aplicada a la recerca per un camp concret. Les possibles restriccions poden ser : igual, superior o igual, superior, inferior o igual, inferior, conté (aplicable només a les propietats de tipus text) , comença per (aplicable només a les propietats de tipus text), té valor null.	

6.1.6 Associacions

Les possibles associacions que es poden utilitzar dins de un diagrama UML són:

Associacions	
Associacions	**Herència**
	
Entitats associatives	**Composicions i agregacions**
	

6.2 Diagrama de les capes de presentació i serveis

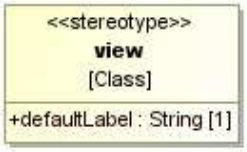
6.2.1 Gestió de la seguretat

La gestió de la seguretat es basa en l'atribució de rols als usuaris. Dins del diagrama, cada rol es representa amb el símbol següent :

Com es veurà en els apartats següents, les classes amb certs estereotips podran tenir una relació de dependència amb cap, un o més d'un actor. Per facilitar la lectura i confecció del diagrama, quan per les esmentades classes no existeixi cap rol associat no s'aplicarà cap restricció de seguretat.

6.2.2 Vista

És l'element inicial per la definició d'una vista. Aquest es dibuixarà com una classe sense propietats i amb l'estereotip «view». Els tags possibles es mostren en la taula següent:

«view»	
Objectiu	Representar una vista
S'aplica a	
Seguretat	Sí
Estereotip	

Tag	Descripció
defaultLabel	Text associat a la vista.

En relació amb els tags definits, la representació visual que s'ha d'obtenir és :

Es tracta simplement d'una pàgina amb el títol deduït del tag corresponent. De la classe de tipus «view» penjaran la resta de classes que compondran la vista complerta.

6.2.3 Llistes de dades

Una llista es modelarà a través d'una classe amb l'estereotip «dataList». Cadascuna de les seves propietats, a les que s'aplicarà l'estereotip «field», representaran una columna. Els tags definits es mostren en les taules següents:

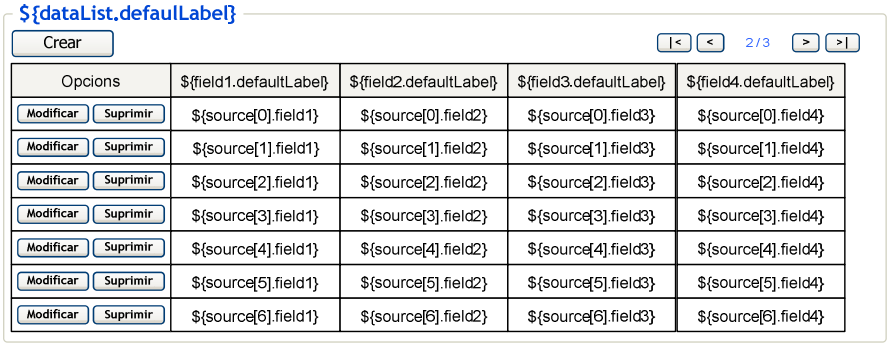
«dataList»	
Objectiu	Representar una llista de dades.
S'aplica a	
Seguretat	Sí
Estereotip	

Tag	Descripció
defaultLabel	Text associat a la llista.

«field»	
Objectiu	Representar una columna
S'aplica a	Classes «dataList», «dataDetail»
Seguretat	No
Estereotip	

Tag	Descripció
defaultLabel	Text associat al camp.
bind	Enllaça el camp amb la propietat d'una entitat.
readOnly	Dins d'un «dataList» no té cap efecte. Dins d'un «dataDetail» indica que el camp només és en lectura. Per defecte, val false.
Invisible	Indica si el camp és invisible. Per defecte, val false.

Cada fila de la taula correspon a una instància d'una entitat. Per associar un camp de la taula a una propietat de l'entitat s'utilitza el tag *bind* de l'estereotip «field». El seu valor és el nom de la propietat de l'entitat.
Per simplificar la introducció del model, el valor del tag *bind* podrà estar a blanc. En aquest cas, el seu valor serà el mateix que el nom de la propietat. Cal notar que les propietats «field» no tenen cap tipus de dada en particular. En efecte, aquest es deduirà del tipus de la propietat de l'entitat a la que s'enllaça.
Finalment, es pot condicionar la representació de la llista dibuixant dependències amb actors.
En relació amb els tags definits, la representació visual que s'ha d'obtenir és :



El text associat a la taula s'obté del valor donat al tag *defaultLabel* de la classe «dataList».
Els títols de les columnes venen donats per cadascun dels labels de les propietats amb l'estereotip «field». També, el contingut de cada cel·la s'obté accedint a cadascuna de les entitats de la llista. Al seu torn, per cada línia, s'accedeix a la propietat adient.
En principi, totes les taules disposaran d'un paginador i no s'ha previst cap tag per la seva gestió.

S'han previst dos tipus bàsics de fonts de dades per alimentar el contingut de les llistes :

- un resultat d'una recerca : La «dataList» s'associa a una classe «search» on es defineixen una sèrie de criteris de recerca i les restriccions corresponents. El resultat és una col·lecció d'entitats que la llista mostrarà. Aquest tipus de font es tracta més en detall en l'apartat *Serveis de recerca.*
- una propietat de tipus llista d'una entitat del model de negoci : En algunes situacions, el contingut d'una llista pot dependre del element seleccionat en una altra llista. En aquest cas la segona llista dependrà no d'un servei en concret sinó d'una propietat de tipus col·lecció de la instància seleccionada en la primera llista. Dins del diagrama, aquest cas es modelarà amb una dependència amb l'estereotip «dataSource». Admetrà el tag *property* que permetrà indicar quina propietat de tipus llista és la font de dades.

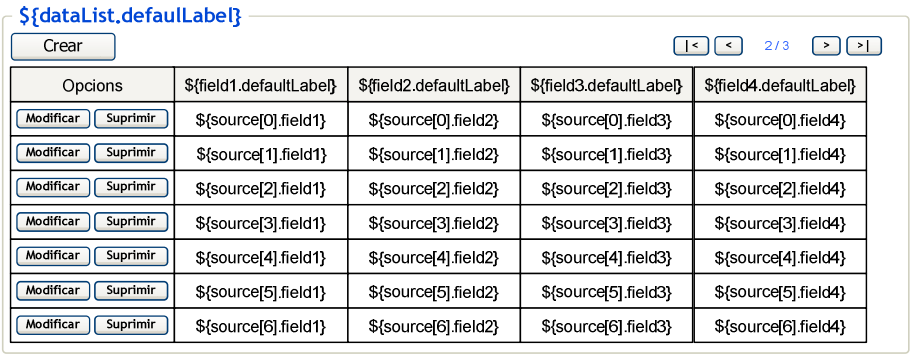
6.2.4 Serveis accessibles des de la llista

Es prendran en compte tres tipus de serveis molt usuals en les aplicacions empresarials : els serveis CRUD, els serveis d'usuari i els serveis de recerca.

Serveis CRUD

Corresponent als serveis bàsics de creació, recuperació, modificació i supressió. Dins de la llista es materialitzen amb botons que donen accés a les opcions adients. Per que la creació i la modificació estiguin disponibles caldrà que per sota de la llista existeixi una classe amb l'estereotip «form».

La representació visual que s'espera obtenir és la següent:



Com es pot comprovar, a deferència de la creació, les opcions de modificació i supressió s'apliquen a cada línia de la llista.
Per controlar l'accés als serveis CRUD, es crearà una relació de dependència amb els rols desitjats. Per poder associar rols diferents a les quatre operacions s'ha definit els estereotips «creation», «edit», «delete», «access». En una associació amb un rol, s'indicarà quines operacions s'apliquen.
En el cas de que l'usuari no tingués dret de creació, modificació o supressió, el botó corresponent no es pintaria.

Serveis d'usuari

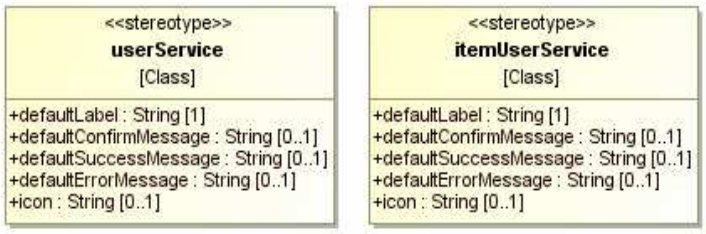
Aquests tipus de serveis corresponen a funcionalitats que l'usuari pot executar directament des de la capa de presentació, usualment, fent click sobre un botó. Així doncs es farà correspondre a cada servei d'usuari un botó amb una etiqueta identificadora. Generalment, en aquestes situacions el sistema demana a l'usuari

que confirmi que realment vol executar la funcionalitat. Un cop executada avisa l'usuari indicant si l'execució ha estat correcte o no.

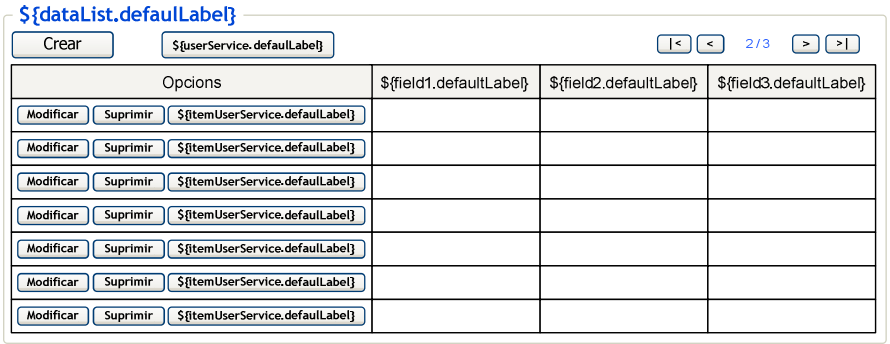
Es distingiran dos categories de serveis d'usuari segons si aquests s'apliquen a una instància d'una entitat («itemUserService») o a un procés no lligat a cap instància en concret («userService»).

Per definir la seva lògica interna seria possible utilitzar diagrames de seqüència, estats o altres. Si bé la idea és temptadora, crec que a la pràctica, el sistema no és prou flexible per prendre en compte la complexitat de la lògica de negoci. Així doncs es preferirà que sigui el programador que la implementi. Tot així s'implementarà de forma automàtica la gestió dels missatges associada.

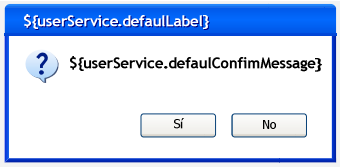
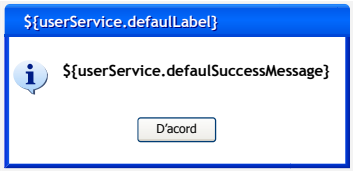
Per modelar els serveis d'usuaris es defineixen els estereotip «userService» i itemUserService». Els tags definits es mostren en la taula següent:

«userService» / «itemUserService»	
Objectiu	Modelar els serveis accessibles des d'una llista.
S'aplica a	Classes «dataList»
Seguretat	Sí
Estereotip	
Tag	**Descripció**
defaultLabel	Text associat a la llista.
defaultConfirmMessage	Missatge presentat a l'usuari per confirmar l'execució del servei.
defaultSuccesMessage	Missatge presentat a l'usuari indicant que la execució del missatge ha estat correcte.
defaultErrorMessage	Missatge presentat a l'usuari indicant que la execució del missatge ha estat incorrecte.
Icon	A més de la etiqueta es podria associar una icona. (es preveu, però no s'implementarà)

En relació amb els tags definits, la representació visual que s'ha d'obtenir és :



Els diferents missatges previstos són (només es mostra el cas corresponent a userService):

Missatges		
Confirmació	**Execució correcte**	**Execució errònia**
		

Serveis de recerca

Una recerca es modelarà a través d'una classe amb l'estereotip «search». Per cadascuna de les seves propietats s'aplicarà l'estereotip «searchField». Els tags definits es mostren en el diagrama següent:

«search»	
Objectiu	Modelar un servei de recerca
S'aplica a	Associació amb classes «dataList»
Seguretat	No
Estereotip	

Tag	**Descripció**
baseEntity	Entitat sobre la que s'aplica la recerca.
orderBy	Llista de propietats de l'entitat base o de les entitats relacionades en el model lògic que serveix per ordenar els resultats obtingut.

Per cada «dataList» es poden definir fins a dos tipus de recerca, una bàsica l'altre avançada. Amb el tag *baseEntity* s'indica una entitat del model de negoci i serà l'entitat sobre la que s'aplicaran els arguments de recerca.

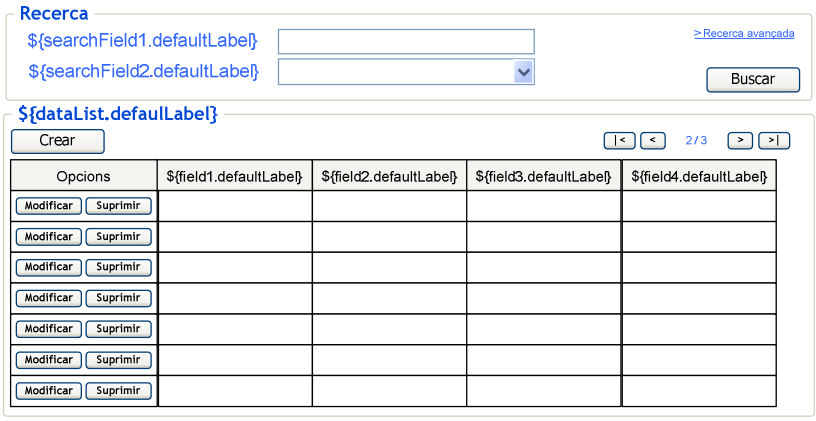
«searchField»	
Objectiu	Representar els camps i arguments d'una recerca.
S'aplica a	Propietats de classes «search»
Seguretat	No
Estereotip	

Tag	**Descripció**
Bind	Propietat de l'entitat del model on s'aplica la restricció.

restriction	Restricció aplicada a la recerca per un camp concret. Les possibles restriccions poden ser : igual, superior o igual, superior, inferior o igual, inferior, conté (aplicable només a les propietats de tipus text) , comença per (aplicable només a les propietats de tipus text), té valor null.	
defaultLabel	Text associat al camp de recerca.	

El tag *bind* permet indicar la propietat de l'entitat base amb la qual està enllaçat el camp de la secció de recerca. Serà sobre aquesta propietat on s'aplicarà la restricció definida dins del tag *restriction*.

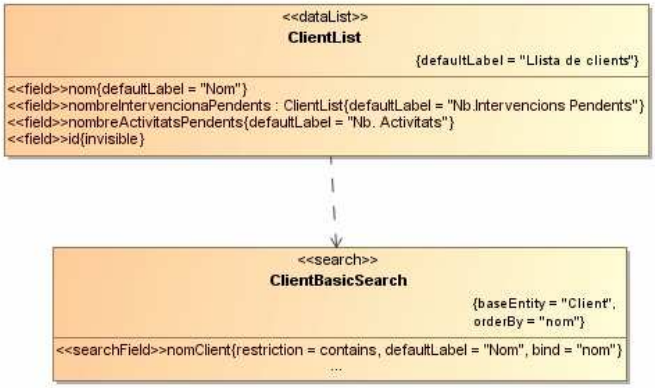
En relació amb els tags definits, la representació visual que s'ha d'obtenir és :



El resultat obtingut correspon al de l'apartat anterior, però ara s'ha afegit la secció de recerca. Es pot observar que aquesta inclou un lligam a la secció de recerca avançada.

Exemple

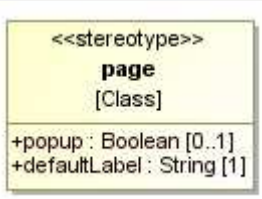
Es completa l'exemple anterior amb la definició d'una recerca bàsica.



La recerca consisteix en un camp nom i un camp tipus de client. Si estan informats es filtraran els clients el nom del qual contingui el text introduït dins del camp de recerca Nom.

6.2.5 Pàgines (pages)

Una pàgina es modelarà a través d'una classe amb l'estereotip «page». Aquesta no tindrà cap propietat. Els tags definits es mostren en el diagrama següent:

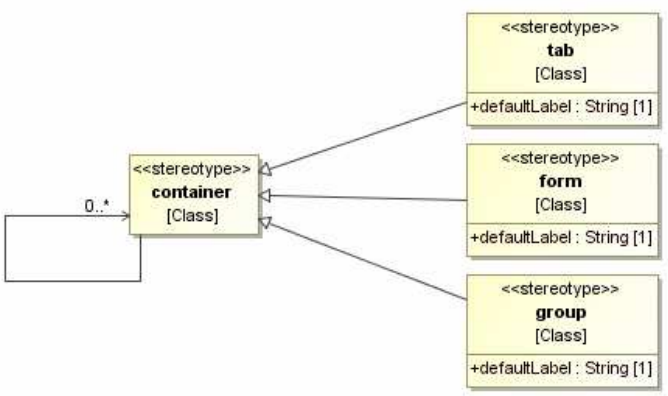
«search»	
Objectiu	Representa una pàgina del navegador.
S'aplica a	
Seguretat	Sí
Estereotip	
Tag	**Descripció**
Popup	Si val true, la pàgina es mostrarà com una pàgina de tipus popup.
defaultLabel	Text associat a la pàgina.

A partir d'una llista, l'usuari pot accedir a una pàgina amb el detall de la línia seleccionada o bé una pàgina que permeti accedir al formulari de creació. Aquesta pàgina pot ser normal, és a dir, reemplaça la pàgina en curs del navegador o de tipus *pop-up* és a dir, es visualitza per sobre de l'actual.

6.2.6 Contenidors (tab,group i form)

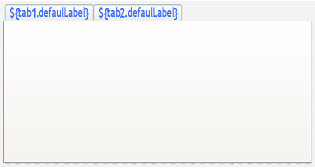
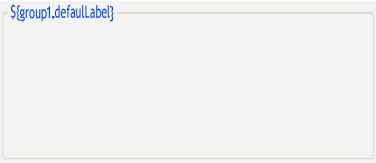
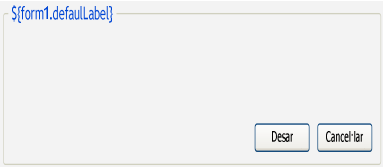
Els estereotips que hereten de «container» representen objectes visuals que poden contenir d'altres objectes de tipus «container». També, com mostra l'associació reflexiva de «container», un contenidor pot 'contenir-ne' d'altres. També, encara que en el diagrama següent no es mostri, també s'hi poden incloure llistes. Accepten relacions de dependència amb actors per modelar la seguretat associat a cada contenidor.

En el diagrama de classes, la relació entre un contenidor i el seu contingut es representa com una composició.

«container»	
Objectiu	Modelar components visuals contenidors.
S'aplica a	
Seguretat	Sí
Estereotip	
Tag	**Descripció**

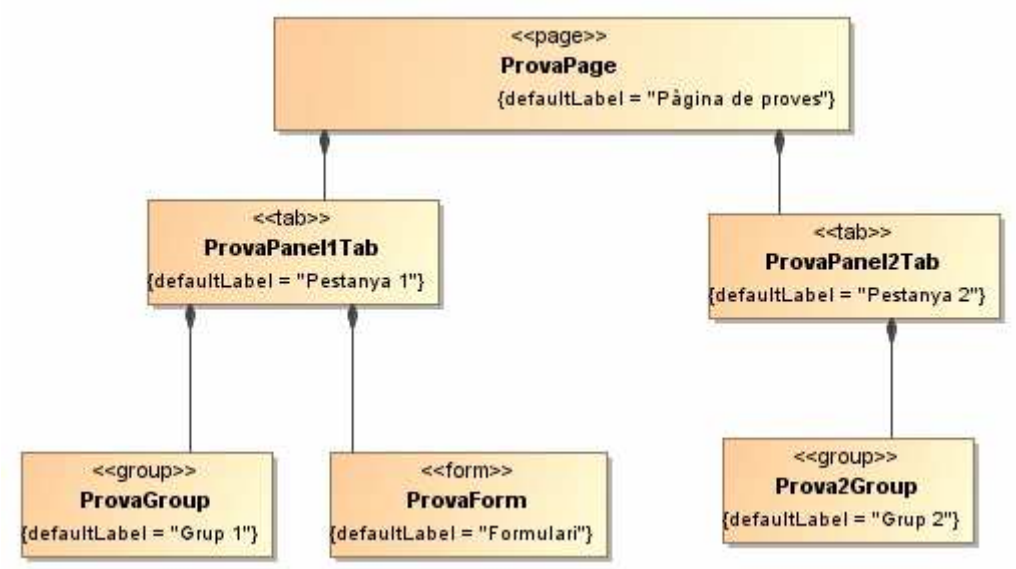
defaultLabel	Text associat al contenidor.

Per aquest treball només s'han considerat tres tipus de contenidors:

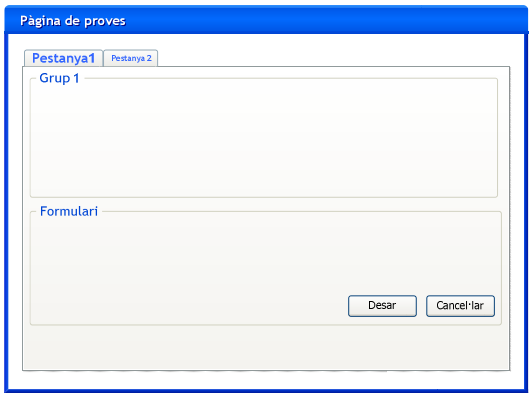
Missatges		
Tab	**Group**	**Form**
Pestanyes que permeten incloure dins d'una mateixa pàgina gran quantitat d'informació	Requadre amb un títol que permet organitzar la presentació de la informació.	El valor dels camps d'entrada inclosos dins d'un contenidor Form s'enviaran al servidor dins de peticions GET o POST
		

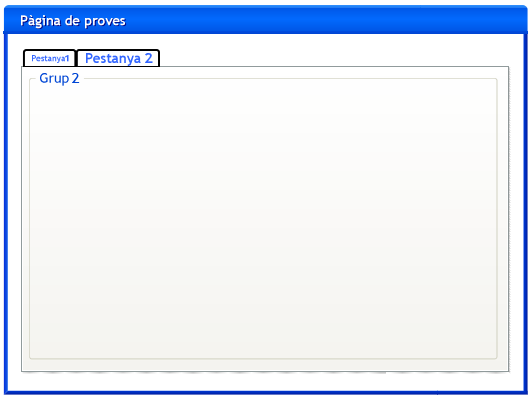
Exemple

Posem per cas el següent diagrama parcial :



La representació que s'espera obtenir és :

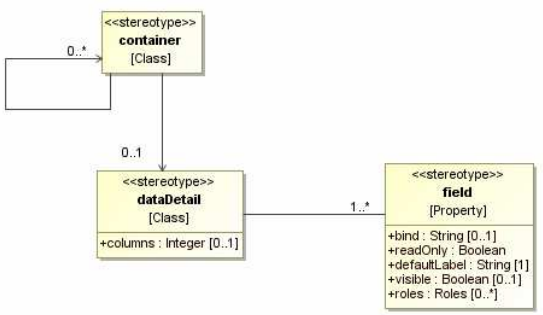




6.2.7 Detall de les dades (dataDetail)

Dins dels contenidors es podran mostrar dades provinents del model. Aquestes es podran mostrar en forma tabular, cas de les «dataList» i ja explicat; o com un conjunt de camps distribuïts dins d'un formulari, cas del «dataDetail».

El detall de les dades es modelarà a través d'una classe amb l'estereotip «dataDetail». Cada propietat d'aquesta classe, a la que s'aplicarà l'estereotip «field», correspondrà a un camp del formulari. Els tags definits es mostren en el diagrama següent:



«dataDetail»	
Objectiu	Representar la presentació de dades com un conjunt de camps.
S'aplica a	
Seguretat	Sí
Estereotip	

Tag	Descripció
Columns	Nombre de columnes a utilitzar per tal de distribuir la informació en pantalla.

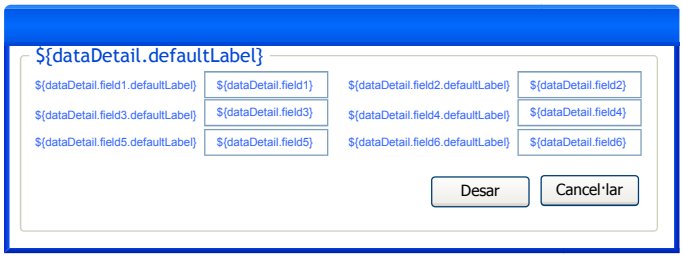
Aplicat a «dataDetail», l'estereotip «field» segueix el mateix funcionament que el detallat en paràgrafs anteriors. Aquí, però, el tag *readOnly* té sentit i permet visualitzar en sortida camps dins de contenidors amb l'estereotip «form».

També, com en el cas de «searchField», es pot crear una dependència entre una propietat «field» i un classificador de tipus «combo» per tal de que el camp corresponent es representi en forma de combo.

Quan una classe «dataDetail» s'inclou dins d'un «form» els camps corresponents estaran en entrada/sortida i les modificacions efectuades seran enviades al servidor en prémer el botó *Desar*.

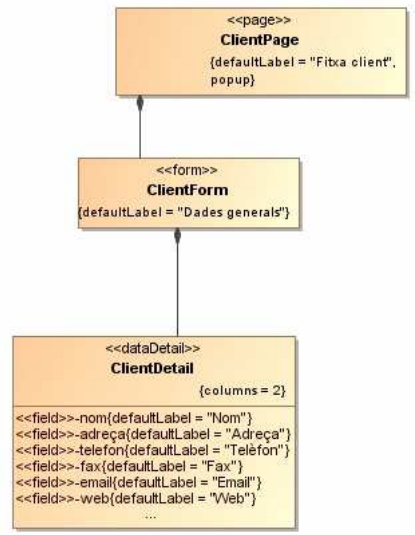
En cas contrari, és a dir dins d'un «group» o «tab», els camps estaran sempre en sortida.

En relació amb els tags definits, la representació visual que s'ha d'obtenir és :



Exemple

En aquest exemple es mostra com es podria modelar un formulari d'edició de les dades d'un client.



6.2.8 Combos

En moltes situacions resulta convenient que l'usuari pugui introduir una dada a partir d'una llista amb tots els valors possibles. Per tant, també s'ha previst la possibilitat de representar quadres combinats. Dins del diagrama es representa com una classe sense propietats i amb l'estereotip «combo».

Tant les propietats «field» de les classes «dataDetail» com les propietats «searchField» de les classes «search» podran definir una dependència amb classes «combo» de forma que els camps es representaran en forma de combos.

Els tags definits es mostren en el diagrama següent:

«combo»	
Objectiu	Representa un quadre combinat.
S'aplica a	Propietats «field» i «searchField»
Seguretat	No
Estereotip	

Tag	Descripció
loadForm	Indica l'entitat les instàncies de la qual constitueixen el contingut de la llista.
value	Una vegada feta la selecció, propietat el valor de la qual s'emmagatzema dins de la propietat «field»
description	Propietat de l'entitat que s'ha de mostrar a l'usuari.

7 IMPLEMENTACIO

7.1 Generació de codi

Per la implementació del generador de codi s'ha utilitzat el framework AndroMDA. A més d'incloure generadors de codi (o cartridges segons terminologia d'AndroMDA) per arquitectures concretes (EJB, Hibernate, etc...), aquest permet construir cartridges propis.

Per implementar el cartridge volgut , s'han seguit els passos recomanats inclosos dins del tutorial *10 steps de write a cartridge*.

En una primera fase, cal modelar tant el PIM (o model independent de la plataforma) com el PSM (o model específic a la plataforma). Aquest modelatge s'efectua utilitzant notació UML i una eina de dibuix compatible tipus MagicDraw.

Després, es tracten els models creats utilitzant els cartridges Java i Meta d'AndroMDA. S'obté un esquelet de classes que cal acabar de implementar per tal de definir totalment el PSM. Finalment, el model de codi es concreta amb la implementació de templates Velocity que corresponen a les classes i fitxers de configuració etc.. de l'arquitectura a generar.

En els punts següents es mostraran els resultats de les fases principals.

7.1.1 PIM (platform independent model)

El PIM especifica el conjunt d'elements UML a partir del qual s'efectuarà el modelatge i que serà la base per tot el procés de generació de codi. Aquí es tracta de classes amb estereotips, associacions i dependències.

Amb AndroMDA, el PIM s'implementa a través d'un conjunt de classes. Per facilitar aquesta tasca, es pot representar el model corresponent al PIM amb un diagrama de classes i executar els cartridges *Java* i *Meta* per crear el codi font necessari (classes i interfícies).

Es tracta de classes amb l'estereotip «metafacade» i associades segons l'estructura del PIM volgut. Abans de fer la generació, AndroMDA podrà comprovar si un model concret satisfà el PIM definit. També cal remarcar que per cada classe s'ha definit un mètode *transformToX()*. Si bé AndroMDA genera l'estructura de classes, una de les tasques efectuades és la implementació dels mètodes de transformació. Aquests permeten passar del model dibuixat per l'usuari als elements concrets de l'arquitectura de destí.

A partir de l'anàlisi efectuat en l'apartat 6, el diagrama de classes és el següent :

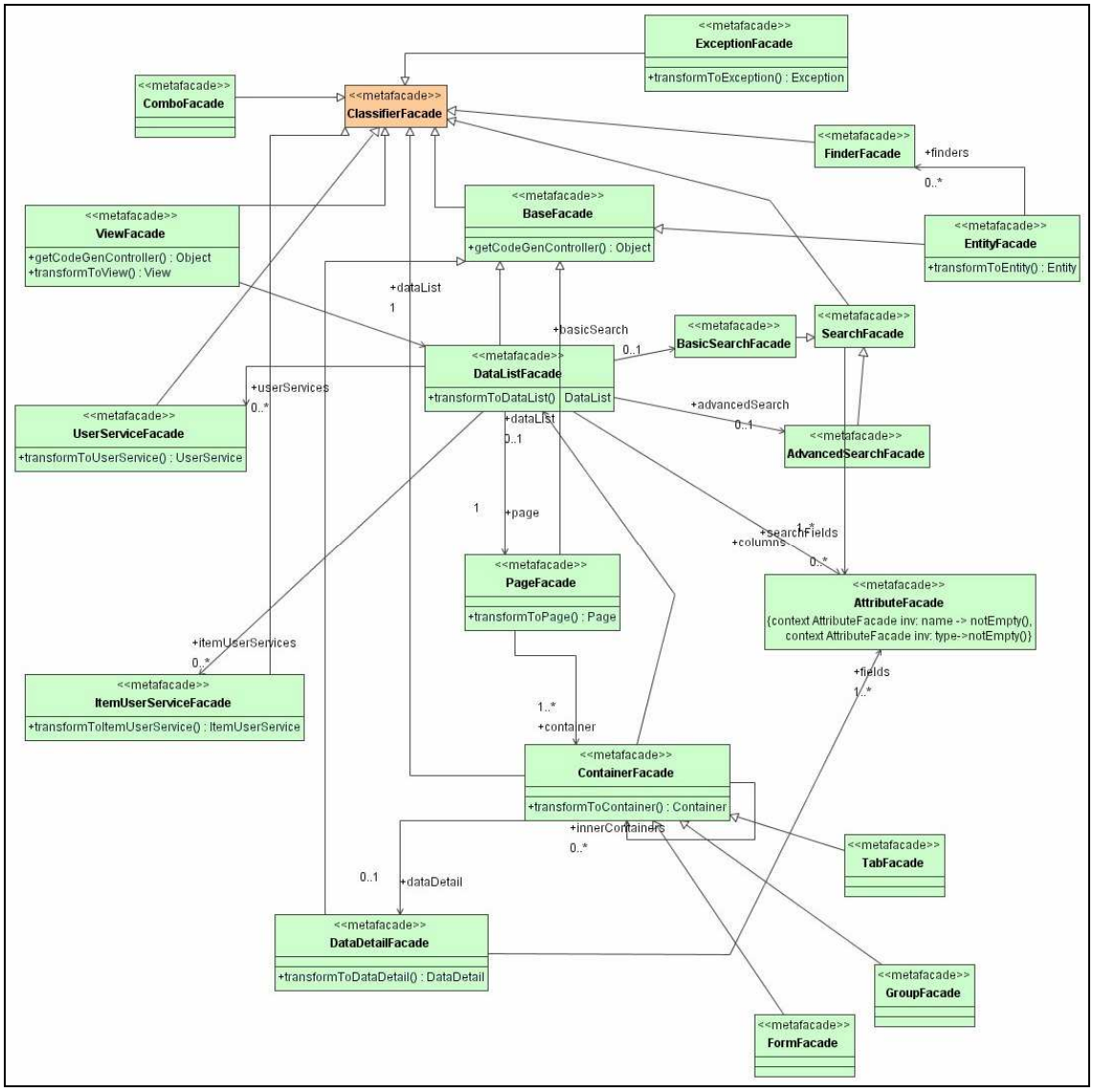


Figura 6 : Diagrama de classes del PIM

7.1.2 PSM (platform especific model)

Per definir el PSM, cal en primer lloc identificar els elements conceptuals principals que composen el conjunt d'artefactes a generar. En el cas del present treball i segons l'anàlisi efectuada en l'apartat 6 tenim :

- D'una banda, el model s'expressa a través de classes Java estàndards anotades segons l'especificació JPA. El conceptes claus que cal considerar són doncs : entitat de model, propietat, operació i associació. El diagrama complet és el següent :

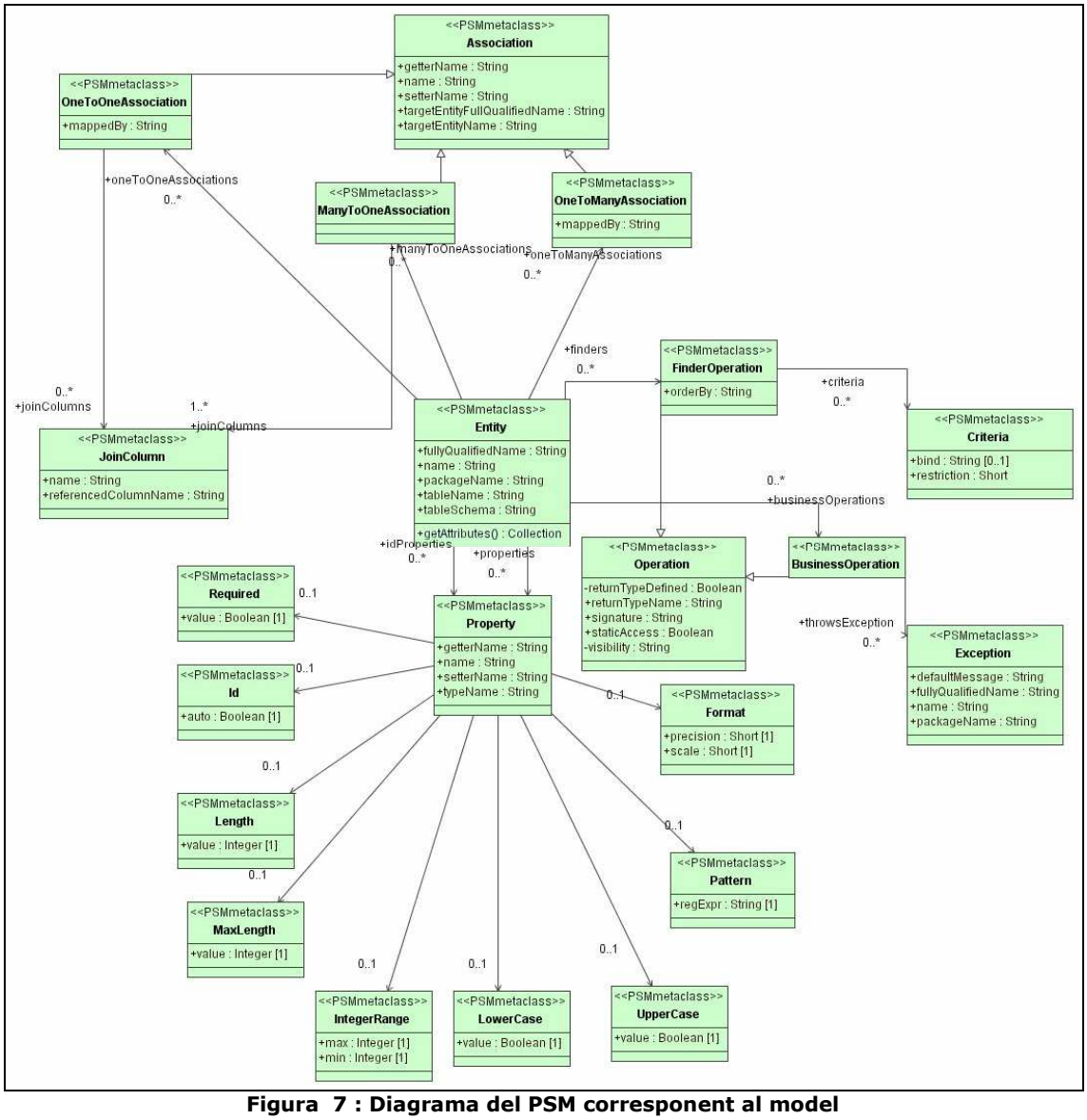


Figura 7 : Diagrama del PSM corresponent al model

Com marca AndroMDA, les classes definides han de tenir l'estereotip «PSMclass»

- Pel PSM corresponent a la capa de presentació i serveis els conceptes clau són vista, servei d'usuari, llista, pàgina, contenidor, detall, combo. El diagrama complet és el següent :

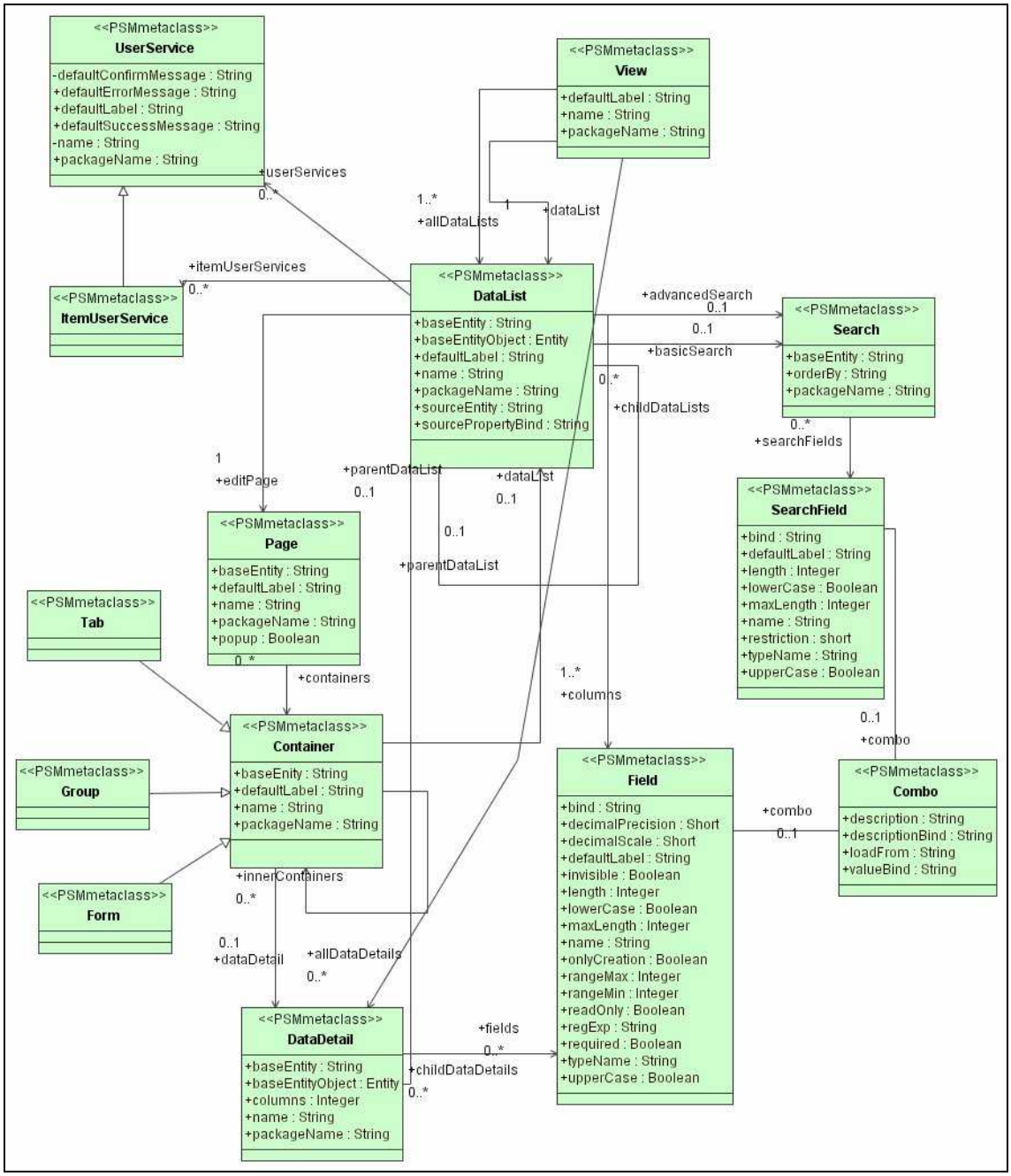


Figura 8 : Diagrama del PSM corresponent a la definició de la vista

Com es pot comprovar a diferència del PIM, cal entrar en detalls relatius a l'arquitectura de destí com ara el noms de paquets. Cal dir que el PSM presentat està relativament simplificat. Un PSM més estricte hauria de considerar altres elements addicionals com per exemple propietats on emmagatzemar els noms de mètodes getters i setters o les signatures dels diferents mètodes; definir la representació del les classes DAO, beans, EJB etc.. és a dir una imatge exacte de l'arquitectura de destí. En aquest projecte, s'ha optat doncs per simplificar el PSM i enriquir els templates. Així, els classificadors del PSM recullen els elements imprescindibles a partir dels quals, i dins dels templates, es podran crear els components de programari.

A tall d'exemple, a partir de les diferents instancies de la classe Entity del PSM es deduiran les classes DAO corresponents. També, a partir de les classes DataList o

DataDetail es podran generar els EJB, beans o interfícies necessaris. (Aquests aspectes es veuran amb més profunditat en l'apartat dedicat a l'arquitectura generada).

7.1.3 Templates

El codi a generar pròpiament dit es defineix dins de templates Velocity. Aquest framework d'Apache té un llenguatge propi que permet combinar les metadades obtingudes del PIM i PSM amb fragments de codi d'implementació per acabar generant el codi de les classes, interfícies i fitxers de configuració.
En el marc del present projecte, els diferents templates implementats són els següents :

Persistència	
Ubicació	...\AndroMDA-pfcjee5-cartridge\src\templates\jee5
Objectiu	Generar el codi de les classes que constitueixen la capa de persistència JPA
Fitxer de macros	jpa-render.vm
Template	**Descripció**
EntityClass.vsl	Permet generar el codi corresponent a una entitat JPA.
PkClass.vsl	Classe amb el codi que representa una clau composta.
EntityDAOBean.vsl	Implementació classes DAO.
EntityDAOInterface.vsl	Interficie locals de les classes DAO.
Exception.vsl	Classes excepcions definides per l'usuari en el model.
Manifest.mf.vsl	Fitxer manifest mòdul EJB
Orm.xml.vsl	Fitxer configuració persistència orm.xml.
Persistence.xml.vsl	Fitxer configuració persistència persistence.xml.
EntityDAOImpl.vsl	Deprecated
EntityFacade	No utilitzada en la versió presentada del projecte.
EntityFacadeLocal	No utilitzada en la versió presentada del projecte.
EntityFacadeRemote	No utilitzada en la versió presentada del projecte.
EntityFacadeRemoteBean	No utilitzada en la versió presentada del projecte.

Negoci	
Ubicació	...\AndroMDA-pfcjee5-cartridge\src\templates\jee5\business
Objectiu	Generar el codi de les classes que constitueixen la capa de negoci EJB 3
Fitxer de macros	Cap
Template	**Descripció**
ViewFacade.vsl	Interfície amb els serveis que composen la capa de serveis.
ViewFacadeBean.vsl	Bean local.
ViewFacadeLocal.vsl	Interfície local EJB3
ViewFacadeLogic.vsl	Implementació lògica dels serveis oferts pel facade.
ViewFacadeRemote.vsl	Interfície remota EJB3
ViewFacadeRemoteBean.vsl	Bean remot.
ListController.vsl	Interfícies amb els serveis del controlador de llista.
ListControllerBean.vsl	Bean local controlador de llista.
ListControllerLocal.vsl	Interfície local EJB3
ListControllerLogic.vsl	Implementació lògica serveis necessaris pel funcionament de la llista de dades.
DetailController.vsl	Interfícies amb els serveis del controlador de detall.
DetailControllerBean.vsl	Bean local controlador de detall.
DetailControllerLocal.vsl	Interfície local EJB3
DetailControllerLogic.vsl	Implementació lògica serveis necessaris pel funcionament del detall de dades.

Presentació	
Ubicació	...\AndroMDA-pfcjee5-cartridge\src\templates\jee5\web
Objectiu	Pàgines JSF/Icefaces i fitxer de configuració de JSF.
Fitxer de macros	icefaces-render.vm, jsf-config.vm
Template	**Descripció**
\beans	
DataListBean.vsl	Backing-bean que gestiona una llista de dades. Hereta de DataListLogic i permet a l'usuari incloure codi propi.
DataListLogic.vsl	Lògica funcionament llista de dades (paginació, recerques, detall)
DataDetailBean.vsl	Backing-bean que gestiona el detall de dades. Hereta de DataListLogic i permet a l'usuari incloure codi propi.
DataDetailLogic.vsl	Lògica funcionament detall dades (workflow CRUD, navegació)
\cfg	
Faces-config.xml.vsl	Fitxer de configuració de JSF. Declara els backing-beans i defineix la navegació.
\pages	
DataListPage.vsl	Pàgina Icefaces que representa una pàgina amb una llista de dades. El backing-bean corresponent es genera amb el template DataListBean.vsl.
Page.vsl	Pàgina Iceface que representa el detall de les dades. El backing-bean corresponent es genera amb el template DataDetailBean.vsl

7.1.4 Configuració del procés de generació de codi

El lligam entre el PIM, PSM i els templates s'estableix principalment a través dels fitxers de configuració de AndroMDA *...\AndroMDA-pfcjee5-cartridge\src\META-INF\AndroMDA\metafacades.xml* i *...\AndroMDA-pfcjee5-cartridge\src\META-INF\AndroMDA\cartridge.xml.*

Dins del fitxer *metafacades.xml* es mapeja cadascuna de les classes del PIM amb un estereotip. Quan el framework tracta el model, cada cop que troba un classificador amb estereotip busca la classe facade corresponent i la instancia. Per exemple, en el cas de l'estereotip «entity», tenim:

```
<metafacade
       class="org.AndroMDA.cartridges.jee5.metafacades.EntityFacadeLogicImpl"
       contextRoot="true">
       <mapping class="org.omg.uml.foundation.core.UmlClass$Impl">
            <stereotype>entity</stereotype>
    </mapping>
</metafacade>
```

Dins del fitxer *cartridge.xml* s'associa a cada facade (indicada dins del tag <modelElement>) un o varis templates. A més d'indicar la localització del template, es configura el nom del fitxer generat (*outputPattern on {0} indica el camí de sortida per defecte i {1} el nom de l'element UML donat al model*), es controla si s'ha de sobreescriure en cas de que el fitxer resultant ja existeixi (*overwrite*), si es genera un únic fitxer (*outputToSingleFile*) o si s'ha de generar inclús quan queda buit (*outputOnEmptyElements*).
Per exemple, el següent fragment XML, indica que per cada *entityFacade*, i doncs per extensió segons l'exemple anterior per cada classificador amb l'estereotip «entity», s'aplicarà el template *EntityClass.vsl* i es crearà un fitxer amb l'extensió *.java*. Aquest fitxer es tornarà a crear de nou cada cop que es posi en marxa el procés de generació.

```
<template     path="templates/jee5/EntityClass.vsl"
              outputPattern="{0}/entities/{1}.java"
```

```
            outlet="entityclass"
            overwrite="true"
            outputToSingleFile="false"
            outputOnEmptyElements="false">
        <modelElements>
            <modelElement variable="entityFacade">
                <type name="org.AndroMDA.cartridges.jee5.metafacades.EntityFacade"/>
            </modelElement>
        </modelElements>
</template>
```

El següent fragment XML mostra com es controla la generació del fitxer de configuració faces-config.xml. En aquest cas, només es crea un sol fitxer per tots les facades *viewFacades.*

```
<template    path="templates/jee5/web/cfg/faces-config.xml.vsl"
             outputPattern="faces-config.xml"
              outlet="faces-config"
             overwrite="true"
             outputToSingleFile="true"
             outputOnEmptyElements="false">
      <modelElements>
            <modelElement variable="viewFacades">
            <type name="org.AndroMDA.cartridges.jee5.metafacades.view.ViewFacade"/>
            </modelElement>
       </modelElements>
</template>
```

Finalment, dins dels templates, es recupera la instancia de la classe *facade* i es crida el mètode *transformToXXX* corresponent. D'aquesta forma, es construeixen els objectes del PSM pertinents.
L'exemple següent correspon al template que permet generar les classes d'entitat JPA. El framework injecta dins del context de Velocity una instància de entityFacade. Com mostra el diagrama de classes del PIM, el mètode *transformToEntity* retorna una instància de *Entity* del PSM. A partir de *entity* es recuperen diferents atributs (paquet, nom, etc...) i es crea el codi pròpiament dit.

```
##-----------------------------------------------------
## Template per la generació de classes entitat
##-----------------------------------------------------
#set ($entity = $entityFacade.transformToEntity())
package ${entity.packageName}.entities;
import java.util.Collection;

## table name es opcional, m
@javax.persistence.Entity
@javax.persistence.Table
##(name="$entity.tableName",schema="$entity.tableSchema")
public class ${entity.name} implements pfcjee5.common.entities.BaseEntity{

#renderPK($entity)
## render propietats
#renderProperties($entity)
## render de les propietats corresponents a associacions
#renderOneToManyProperties($entity)
#renderManyToOneProperties($entity)
#renderOneToOneProperties($entity)
## Render constructor sense paràmetrs
#renderConstructor($entity)
## Render getter/setter pk
#renderPKGetterSetters($entity)
## Render getter/setter propietats
#renderPropertiesGetterSetters($entity)
```

```
## Render getter/setter one-to-many
#renderOneToManyGetterSetters($entity)
## Render getter/setter many-to-one
#renderManyToOneGetterSetters($entity)
## Render getter/setter one-to-one
#renderOneToOneGetterSetters($entity)

##mètodes de negoci
#renderBusinessOperations($entity)
}
```

7.2 Arquitectura generada (o model de codi)

7.2.1 Consideracions generals

Per aprofitar les possibilitats del framework EJB s'ha pensat interessant que el resultat del procés de generació fossin dos projectes. L'un implementarà la capa de presentació l'altre la lògica de negoci, model i persistència. El segon projecte donarà accés als seus serveis a través de EJB de sessió remots (una de les característiques que diferencien EJB de Spring). Més concretament, a cada *View* del model correspondrà un EJB remot al qual s'accedirà des de la capa de presentació per accedir als serveis CRUD i de negoci. Aquest actua com a façana de la capa de negoci i n'amaga la complexitat a la capa de presentació. El fet de definir una única façana per *View* permet reduir la complexitat de l'arquitectura i hauria de permetre un acoblament menor entre presentació i lògica.

Dins de l'arquitectura generada s'han de considerar tres tipus principals de classes:

- Classes base : són classes que es troben dins del paquet *pfcjee5* i que no són generades. Són part important de l'arquitectura i implementen funcions bàsiques (funcionalitats CRUD, localització de serveis, etc...) comunes a tots els projectes generats. La seva utilització facilita la tasca de generació de codi. Sovint permeten evitar la creació de noves classes utilitzant mecanismes més senzills (cf. creació DTO d'entitat).
- Classes generades en cada procés : Generalment, implementen lògica de funcionament depenent del model. Es generen novament per cada execució del procés de generació de codi.
- Classes d'usuari : Són classes generades automàticament únicament si no existeixen. Hereten de classes generades i en no ser creades cada vegada permeten a l'usuari modificar el comportament per defecte o implementar la lògica de negoci.

7.2.2 Exemple generat

Per explicar l'arquitectura, em basaré en un exemple que ha estat generat pel cartridge implementat i que permetrà mostrar les característiques bàsiques del codi generat. (el diagrama complet es troba en la carpeta pfcjee5-template\src\uml)

Captures de pantalles

- La pantalla següent correspon a la llista de clients. Es poden efectuar recerques bàsiques i avançades.

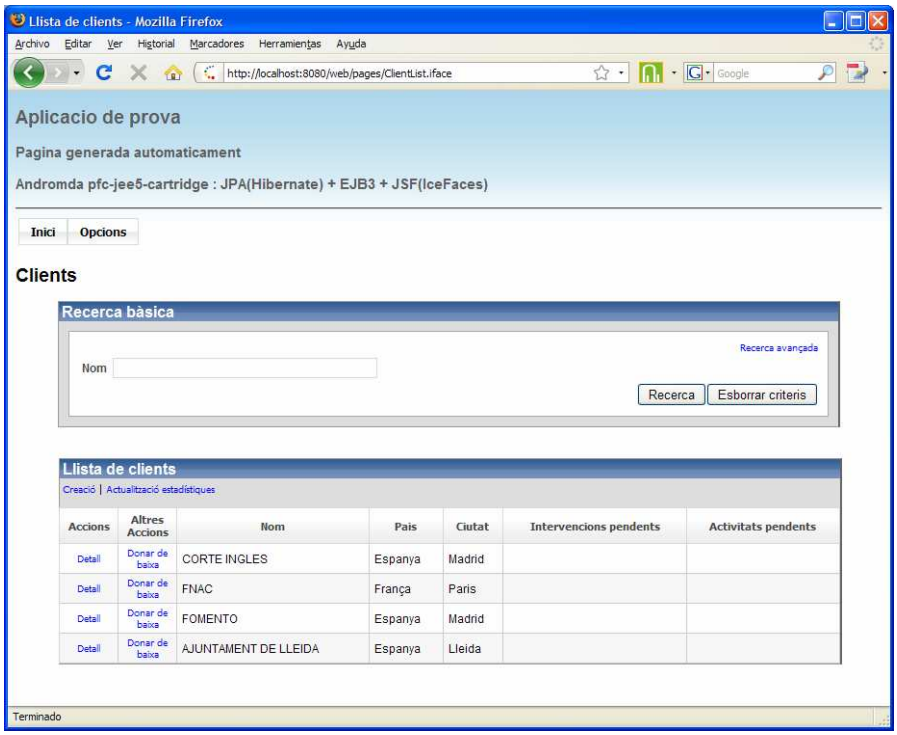


Figura 9 : Llista de clients

- S'emplena la zona de recerca i es prem el botó *Recerca*

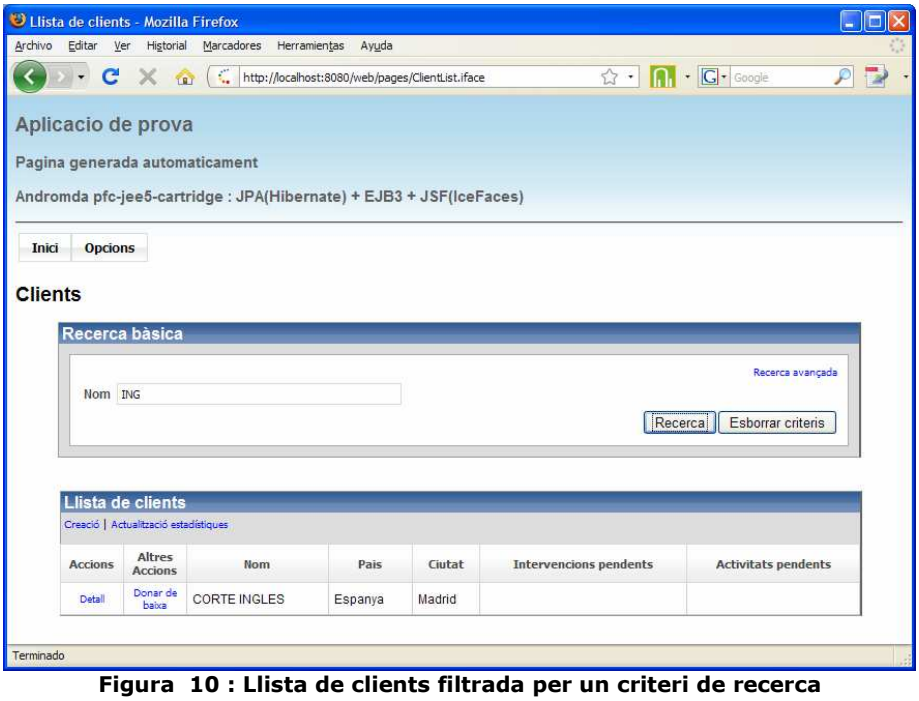


Figura 10 : Llista de clients filtrada per un criteri de recerca

- Es passa al mode de recerca avançada seguint el link *Recerca avançada.* Una vegada en mode de recerca avançada, es pot tornar al mode bàsic seguint el link *Recerca bàsica.*

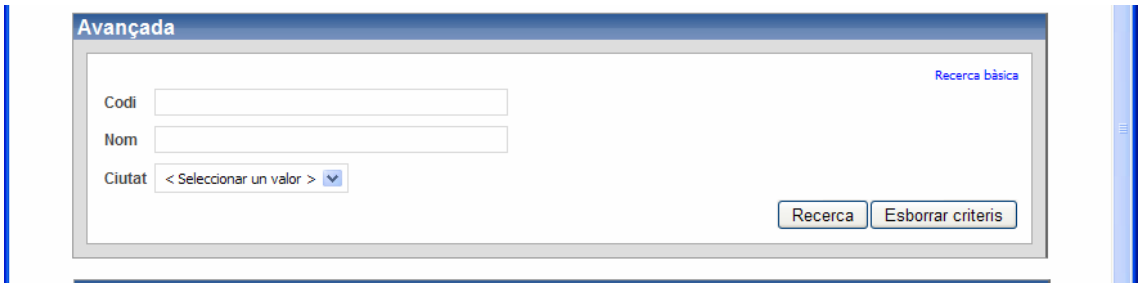


Figura 11 : Recerca avançada

- Es fa una recerca sobre el nom de la ciutat i s'obtenen els resultats esperats.

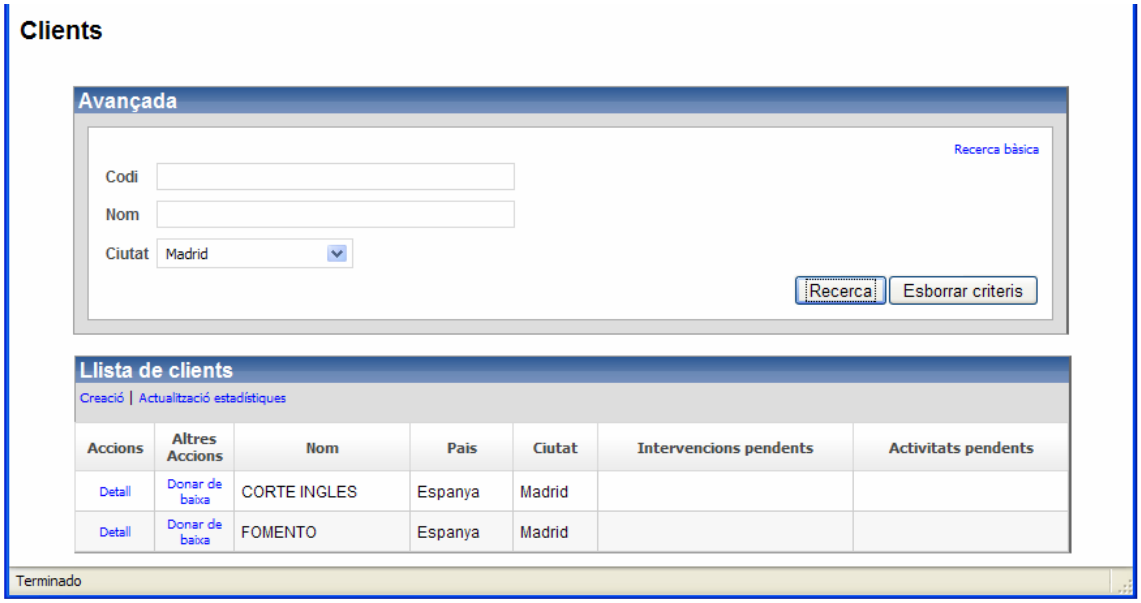


Figura 12 : Llista de clients filtrada per criteris de la recerca avançada

- Es segueix el link *Detall* corresponent al client amb el nom CORTE INGLES. S'accedeix a la pàgina de detall corresponent.

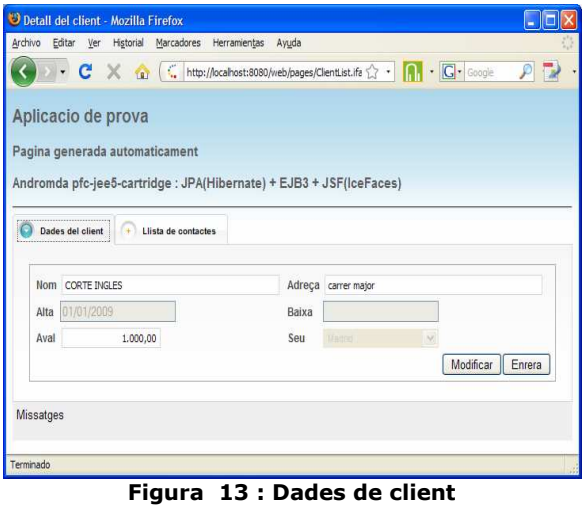


Figura 13 : Dades de client

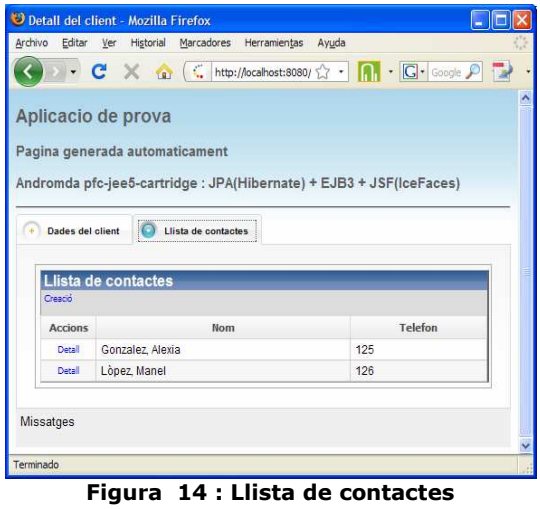
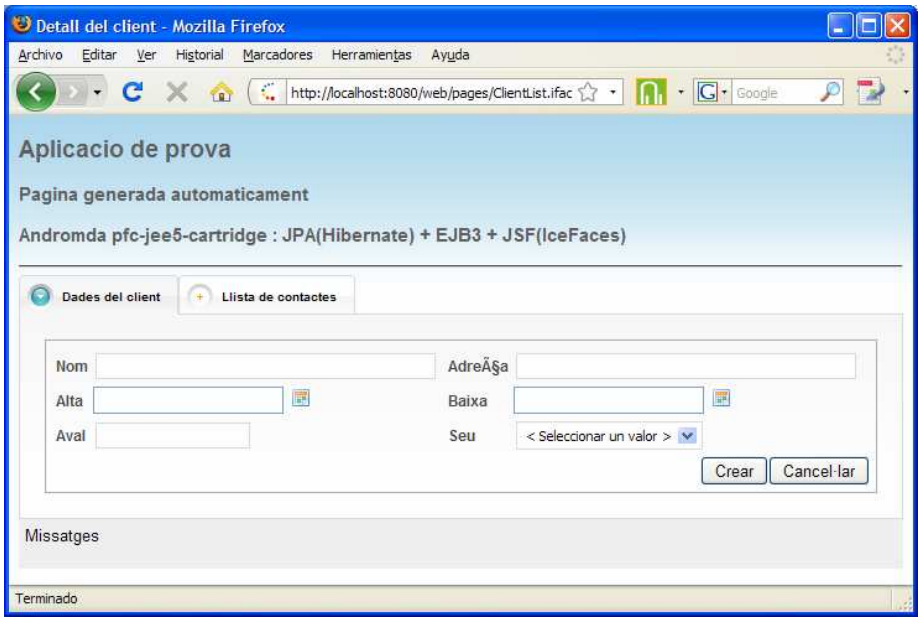


Figura 14 : Llista de contactes

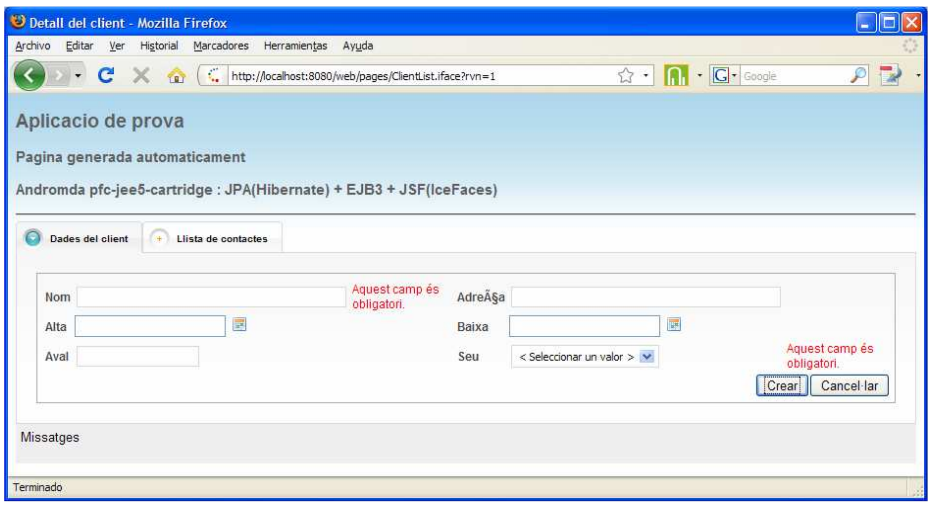
Les pàgines de detall segueixen un flux ben determinat. En obrir-se, la pàgina està en mode *Read.* En aquest mode els camps no es poden editar i l'usuari només pot fer *Enrera* per tornar a la llista anterior o *Modificar* per modificar les dades presentades. En aquest cas, els camps esdevenen editables i l'usuari pot modificar-ne el contingut. Les dades es poden validar fent *Desar* o anul·lar i tornar a mode *Read* fent *Cancel·lar*.

- L'usuari pot crear un client nou seguint el link *Creació.*



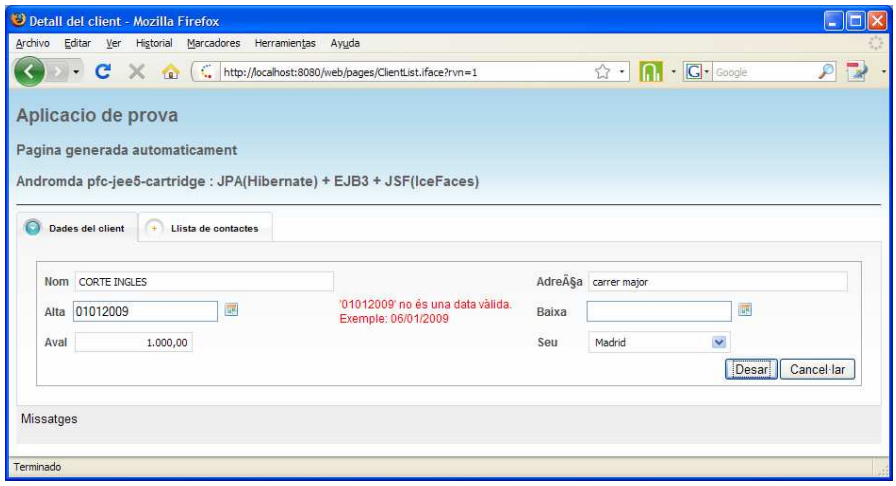
En fer *Crear* o *Desar* segons el mode, s'efectuen la validacions indicades en el model :

- Control nom obligatori:

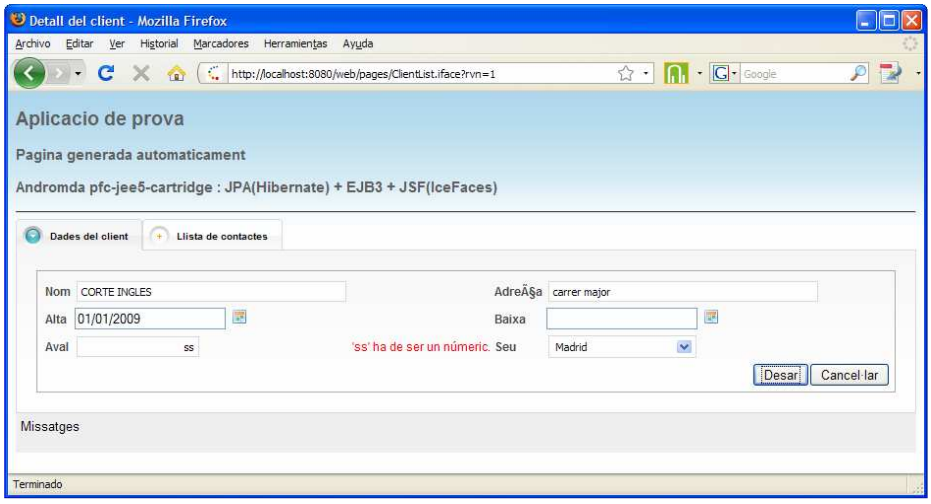


A més, com indica el model, el nom de client es posa automàticament en majúscules.

- Control data correcte :



- Control aval és un número:



Es pot afegir un nou contacte al client

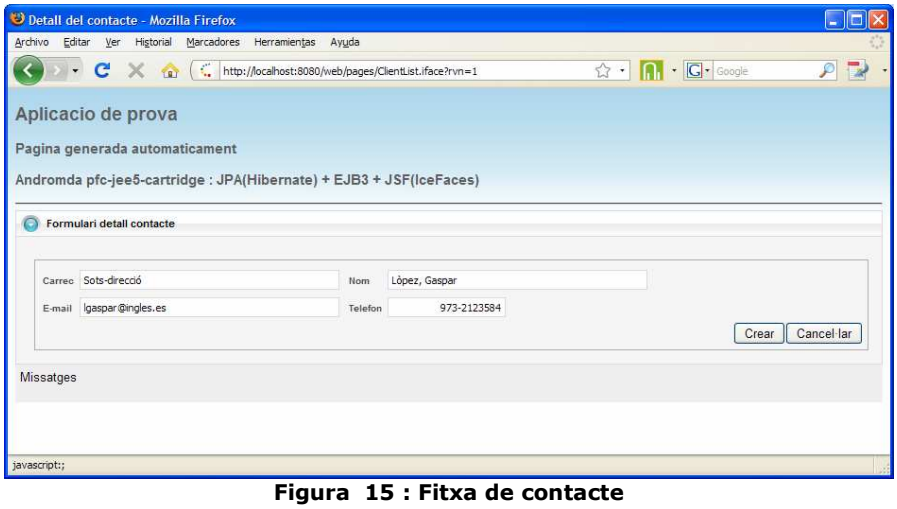


Figura 15 : Fitxa de contacte

Després de validar una creació, la llista visualitzada té en compte la fila creada.

- L'usuari crida a la funció corresponent a l'actualització de les estadístiques. Com indica el model, es mostra la pantalla de confirmació adient:

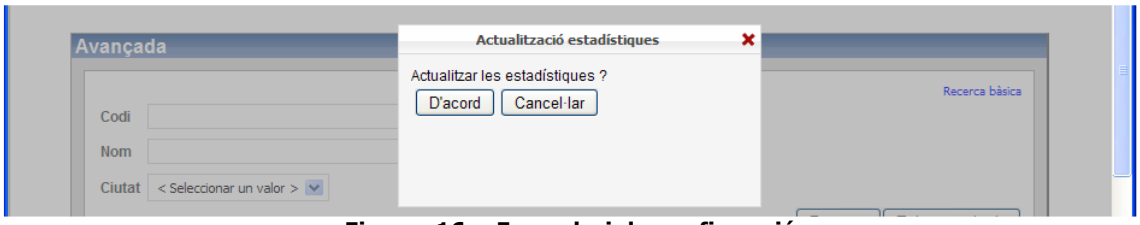


Figura 16 : Formulari de confirmació

- Si l'actualització s'efectua correctament, apareix el missatge de confirmació.

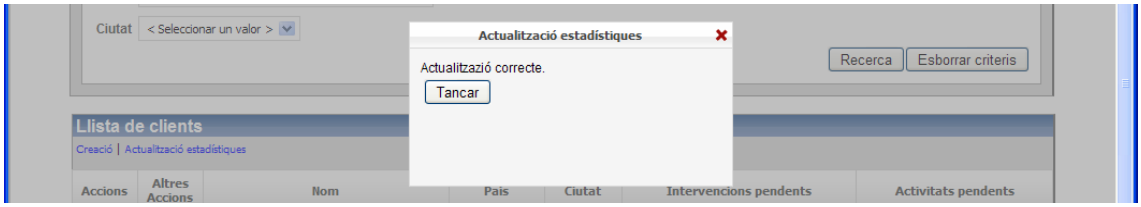


Figura 17 : Formulari operació correcte

7.2.3 Transferència entre capes

Una de les decisions que cal prendre quan es dissenya una arquitectura multi-capa és quina tècnica utilitzar per traspassar les dades entre capes. En la part de disseny, no es va prendre cap decisió al respecte esperant a avançar en l'estudi de la generació de codi per escollir la tècnica més adient.
Si bé la utilització de qualsevol de les dues solucions exposades en l'apartat 5.1.3 (*Custom DTO* o *Hash Map DTO*) és convenient, després d'unes primeres proves he preferit utilitzar els Hash Map DTO. Efectivament, d'entrada, simplifica enormement la generació de codi en no ser necessari construir ni els DTOs corresponents a cada llista o detall ni, sobre tot, les classes assembladors per cada DTO creat. Facilita la utilització d'un conjunt de classes base comunes i genèriques.
Així doncs el conjunt de classes DTO i la classe *Assembler* es mostren en el següent diagrama :

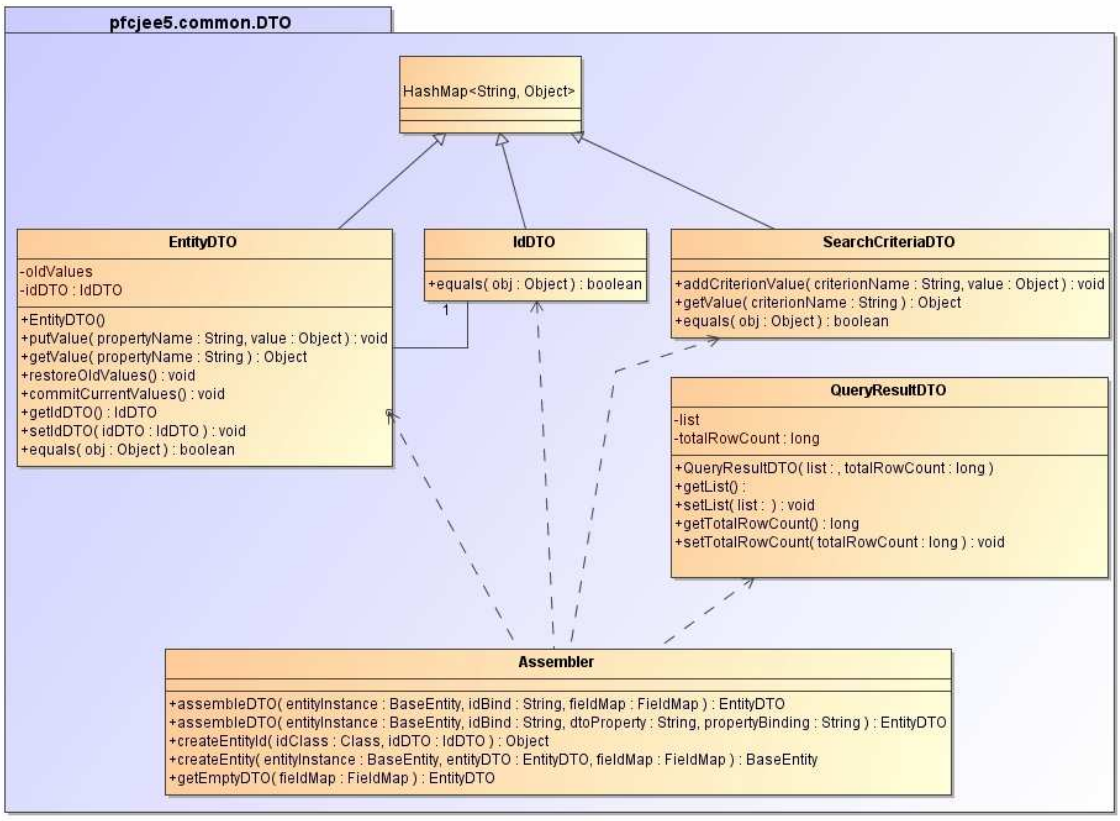


Figura 18 : Classes base gestió DTO

Com es pot comprovar s'han definit més d'un tipus concret de DTO segons el tipus d'informació que transporten. Tot i així, totes les classes hereten de la classe *HashMap* i el seu funcionament bàsic correspon a un *HashMap DTO* clàssic.
El seu funcionament dins de l'arquitectura proposada es pot comprovar en els diferents diagrames de seqüència explicats més endavant.

EntityDTO i IdDTO

La classe *EntityDTO* permet transportar les dades relatives a les entitats. La clau del *HashMap* és el mateix nom d'atribut indicat en el model per especificar les columnes de les llistes o els camps de les pàgines de detall. El valor de retorn correspon a la dada pròpiament dita.
Per fer genèric el tractament d'aquestes DTO, cada classe controladora de llista (hereta de *DataListController*) o de detall (hereta de *DataDetailController*), utilitza una instància de la classe *FieldMap* (classe que simplement hereta de *LinkedHashMap*) per mantenir el lligam entre una propietat i el binding corresponent. Aquest lligam es fa efectiu en el constructor de les classes controladores i el codi corresponent es genera automàticament. Per exemple, per la classe *ClientListControllerLogic* es pot trobar el següent fragment de codi :

```
//- Camps de la taula
this.addField("nom","nom");
this.addField("nombreIntervencionsPendents","nombreIntervencionsPendents");
this.addField("nombreActivitatsPendents","nombreActivitatsPendents");
this.addField("id","id");
this.addField("nomCiutat","seu.id.ciutat");
this.addField("nomPais","seu.id.pais");
```

Amb aquesta tècnica la classe AssemblerDTO esdevé genèrica aprofitant la informació aportada pels bindings i les possibilitats de la invocació dinàmica del llenguatge Java. D'aquesta forma, passant per paràmetre una instància d'una entitat donada i un FieldMap, el mètode assemblerDTO de la classe Assembler retorna un objecte *EntityDTO* correctament creat. Inversament, el mètode *createEntity* permet reconstruir una instància de entitat a partir d'un objecte *EntityDTO*.
Cal notar que per fer possible aquest funcionament ha estat necessari definir la classe *IdDTO*. Per cada instància de *EntityDTO* es té un objecte *IdDTO* que permet identificar cada *EntityDTO* per la seva clau. El fet de crear un DTO especial únicament per l'identificador en comptes d'utilitzar un atribut més dins de la *HashMap* de *EntityDTO* permet gestionar d'una forma més neta i genèrica les entitats amb claus úniques compostes (cf. entitat *Ciutat* del model de prova).

SearchCriteriaDTO i QueryResultDTO

D'una forma similar a *EntityDTO*, la classe *SearchCriteriaDTO* és una *HashMap* que per cada camp inclòs en una recerca guarda el valor introduït per l'usuari. També, de forma similar s'ha definit la classe *SearchCriteria* que dins de les classes controladores permet associar a cada camp de recerca el binding corresponent i el tipus de restricció a aplicar. Per exemple, per la classe *ClientListControllerLogic* es pot trobar el següent fragment de codi :

```
//- Criteris recerca bàsica
this.getBasicSearch().addCriterion("nomClient",
      pfcjee5.common.DAO.Restrictions.contains,"nom");
//- Criteris recerca avançada
this.getAdvancedSearch().addCriterion("codiClient",
      pfcjee5.common.DAO.Restrictions.equal,"id");
this.getAdvancedSearch().addCriterion("nomClient",
      pfcjee5.common.DAO.Restrictions.contains,"nom");
this.getAdvancedSearch().addCriterion("seuId",
      pfcjee5.common.DAO.Restrictions.equal,"seu.id");
```

Les dues classes esmentades seran utilitzades principalment per la classe *HqlBuilder* per construir la sentència Hql adient. Finalment, el resultat de la recerca es retorna a la capa visual dins d'una instància de *QueryResultDTO*. Bàsicament, es tracta d'una llista d'instancies de *EntityDTO* i un enter amb el nombre total de files corresponents a la consulta sense considerar la paginació. Aquesta informació servirà per presentar de forma paginada els resultats de les recerques efectuades per l'usuari. (cf. **Construcció de sentències HQL** dins de l'apartat **Model de negoci i persistència**)

7.2.4 Arquitectura capa de presentació

En una estructura estàndard JSF les pàgines Web de faces són controlades per *Backing Beans*. En l'arquitectura proposada, es defineix un *Backing Bean* per cada llista (classificador amb estereotip «dataList») i detall (classificador amb estereotip «dataDetail») definit en el model. L'accés a la capa de negoci es fa a través d'un únic EJB remot que actua com a façana respecte a la complexitat de la lògica de negoci. En relació al model existirà un EJB remot d'aquest tipus per cada classificador amb l'estereotip «view» .

Diagrama de seqüència

En el diagrama de seqüència següent, es mostra com col·laboren les diferents classes de la capa de presentació en resposta a un seguit d'accions d'usuari usuals:

l'usuari emplena criteris de recerca i prem *Recercar* (cf. missatge 1); després selecciona una fila de la taula i fa *Detall* (cf. missatge 9); els sistema mostra el detall de la fila seleccionada i l'usuari prem el botó *Modificar (*cf. missatge 19), pot modificar la informació presentada i fer *Guardar* (cf. missatge 23) per emmagatzemar-la dins de la base de dades; finalment torna a la llista fent *Enrera* (cf. missatge 30).

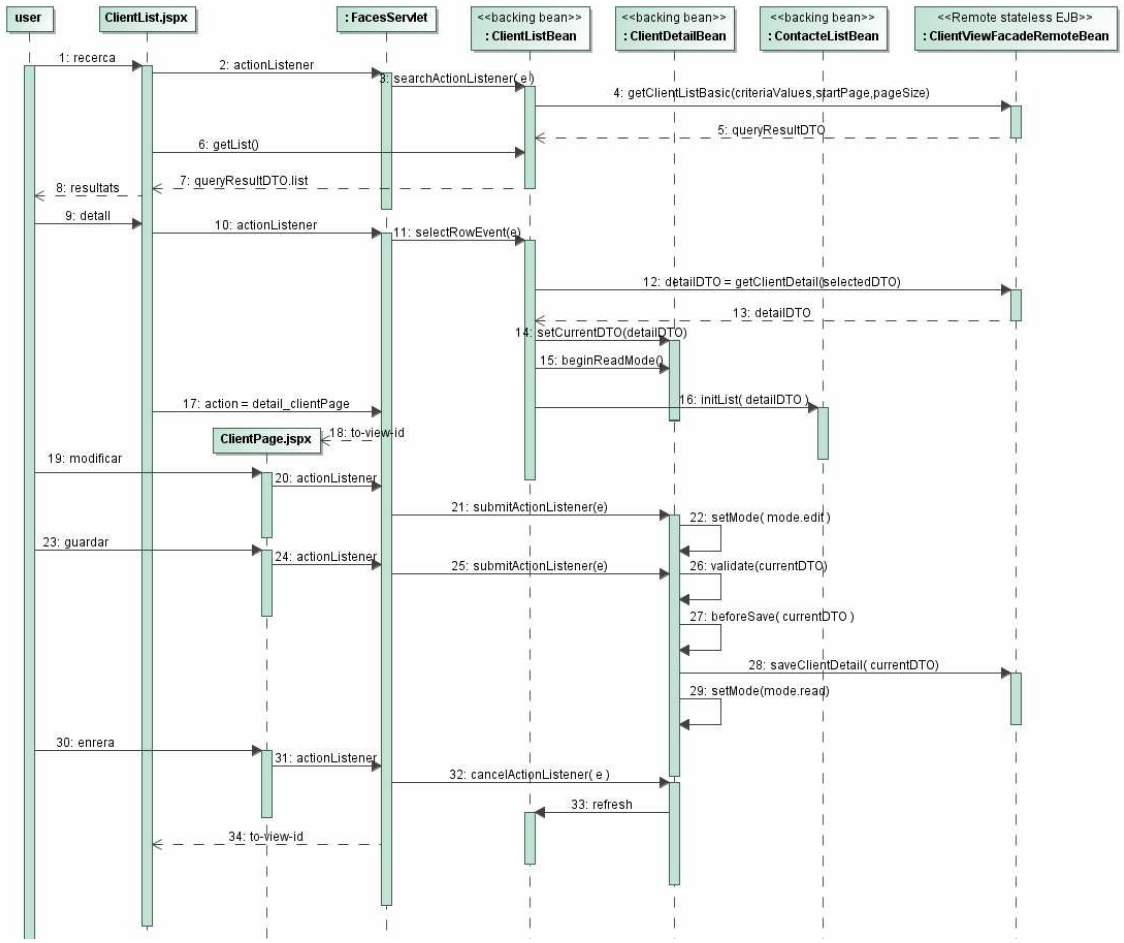


Figura 19 : Diagrama de seqüència capa de presentació

En aquest altre, es mostra la seqüència de missatges necessària per implementar una crida a una funcionalitat de negoci, en aquest cas actualitzar les estadístiques.

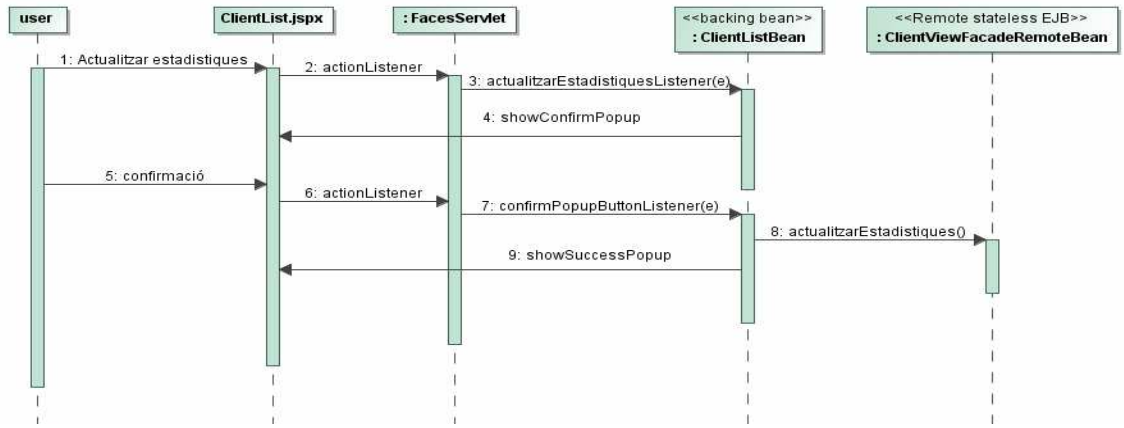


Figura 20 : Diagrama de seqüència accés serveis d'usuari

Diagrama de les classes base

El funcionament bàsic dels *backing beans* de les llistes es troba dins de les classes *BaseListBean* i *DataDetail*. La classe *SelectList* permet gestionar el contingut dels combos mentre que *EntityDTOConverter* és el converter que permet controlar la visualització i recuperació de l'element seleccionat dins del combo.

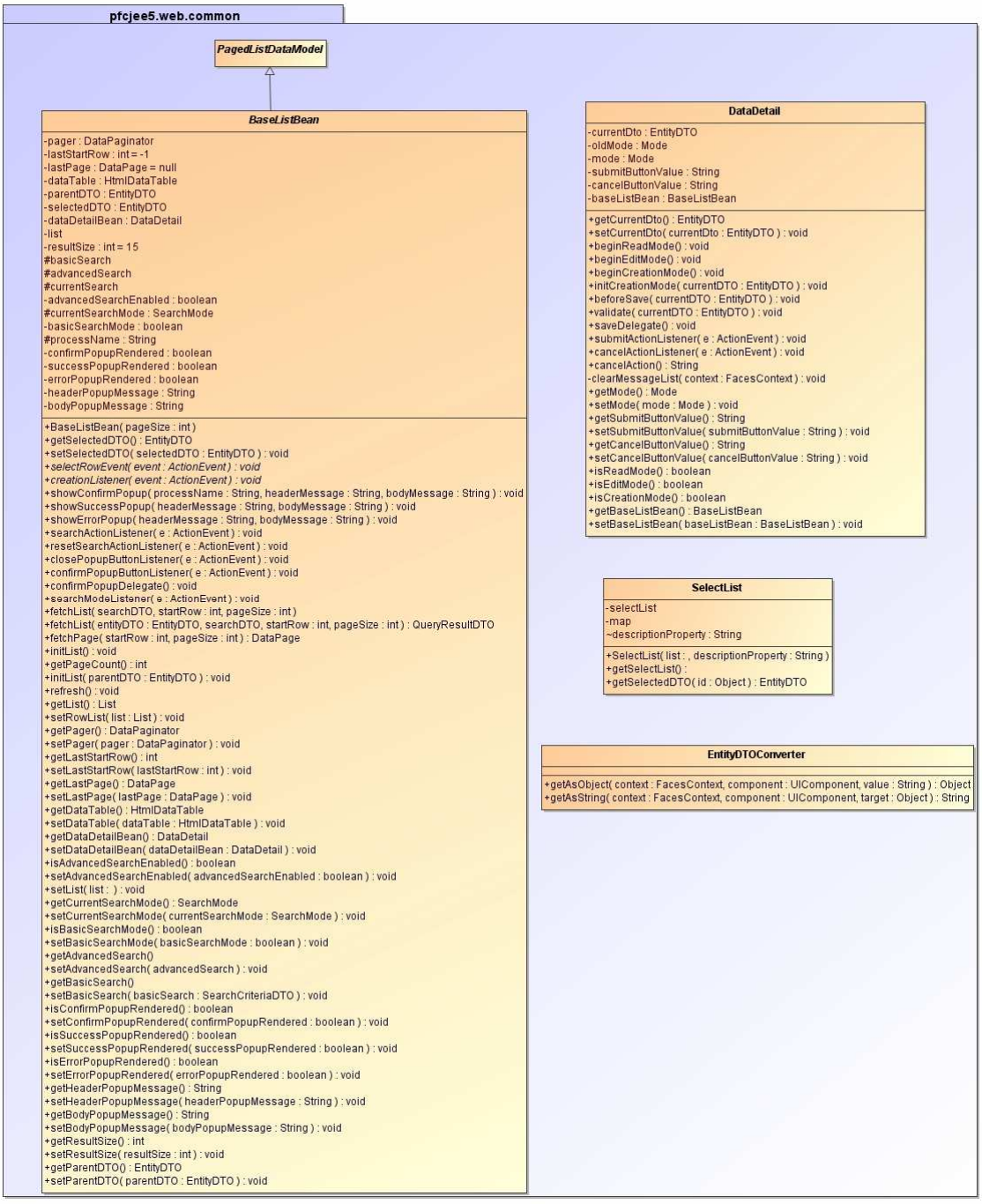


Figura 21 : Classes base capa de presentació

Diagrama de les classes generades

A banda de les classes base, es generaran automàticament les classes que es mostren en el següent diagrama :

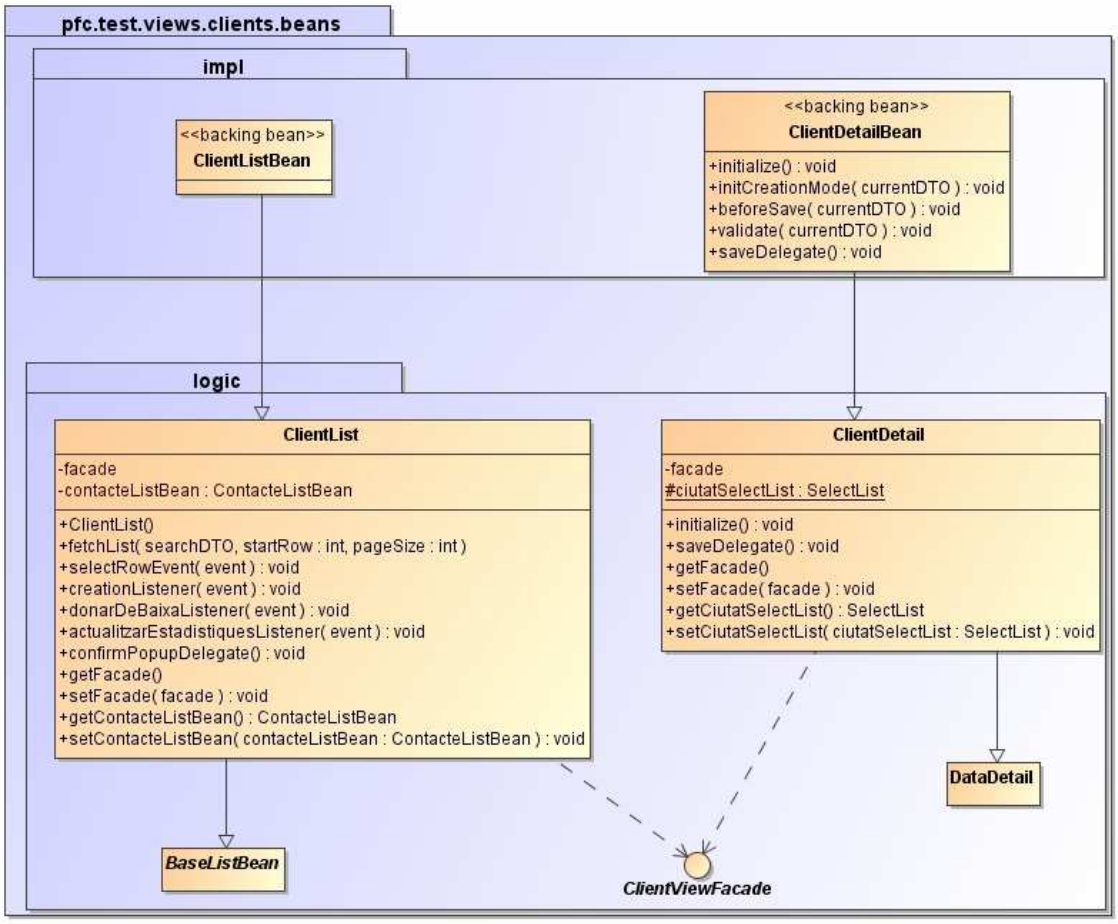


Figura 22 : Classes de la capa de presentació del model de proves

Dins del paquet *Logic,* trobem les classes *ClientList* i *ClientDetail* que implementen tota la lògica que es pot deduir del model. Les classes *ClientListBean* i *ClientDetailBean* que hereten de les anteriors no contenen cap implementació i tenen com a objectiu permetre a l'usuari modificar el funcionament de la lògica tornant a definir els mètodes adients. Per exemple, pot implementar el mètode *initCreationMode* per inicialitzar el valor dels camps dels formularis durant la creació. Cal dir que mentre que *ClientList* i *ClientDetail* es destruiran i tornaran a crear cada cop que s'executa el procés de generació de codi, les classes *ClientListBean* i *ClientDetailBean* només es crearan si no existeixen.

Altres elements generats

També de forma automàtica es generen les pàgines de *faces* necessàries. Donat que en el present treball s'ha treballat la estructura i no aspectes relacionats amb l'estil de les pàgines generades aquestes només es generaran una primera vegada per permetre a l'usuari millorar el disseny.
Per un altra banda, la instanciació dels diferents beans així com el control de la navegació es fa a partir del fitxer de configuració *faces-config.xml*. Aquest es generarà automàticament en cada nova generació.

7.2.5 Lògica de negoci

Dins del marc d'una aplicació EJB3, per la implementació de la lògica de negoci s'utilitzen EJB de sessió locals. D'una banda, s'implementa la lògica de negoci dins de l'EJB remot façana (en l'exemple tractat *ClientViewRemoteBean*) i dins dels EJB locals *ClientListController*, *ClientDetailController, ContacteListController* i *ContacteDetailController.*
Per cada llista (classificador amb estereotip «dataList») i detall (classificador amb estereotip «dataDetail») definit en el model s'implementarà una classe controladora amb els mètodes deduïts del model.

Diagrama de seqüència

A continuació es mostren tres diagrames de seqüència que corresponen a les tres accions d'usuari detallades en el punt anterior, però ara, des del punt de vista de la capa lògica.
En la primera acció, la façana envia un missatge *find* al bean controlador de llista per tal de recuperar la llista de clients corresponents.

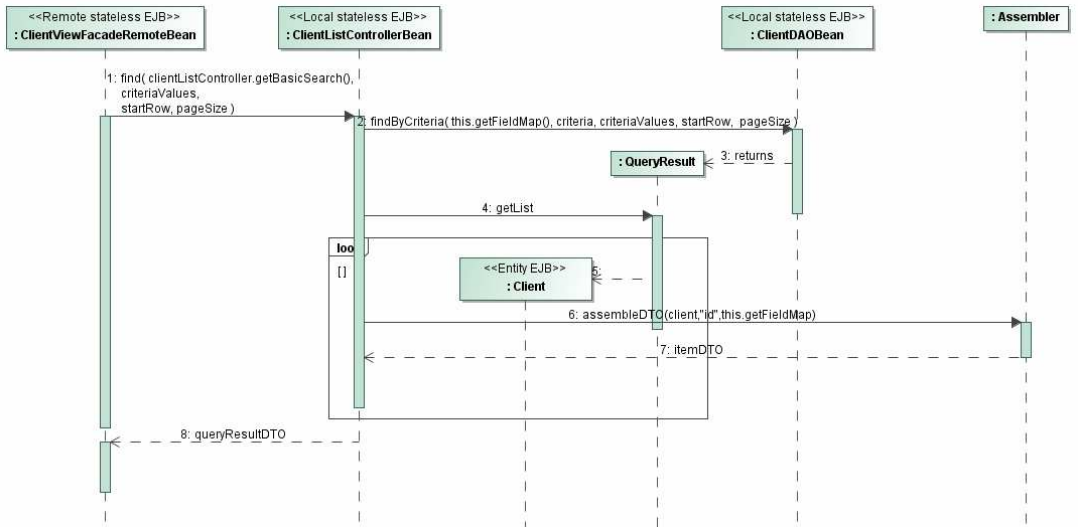


Figura 23 : Diagrama de seqüència recerca bàsica

En la segona, la mateixa façana envia un nou missatge *getDetail* per tal d'obtenir totes les dades relatives a l'entitat seleccionada.

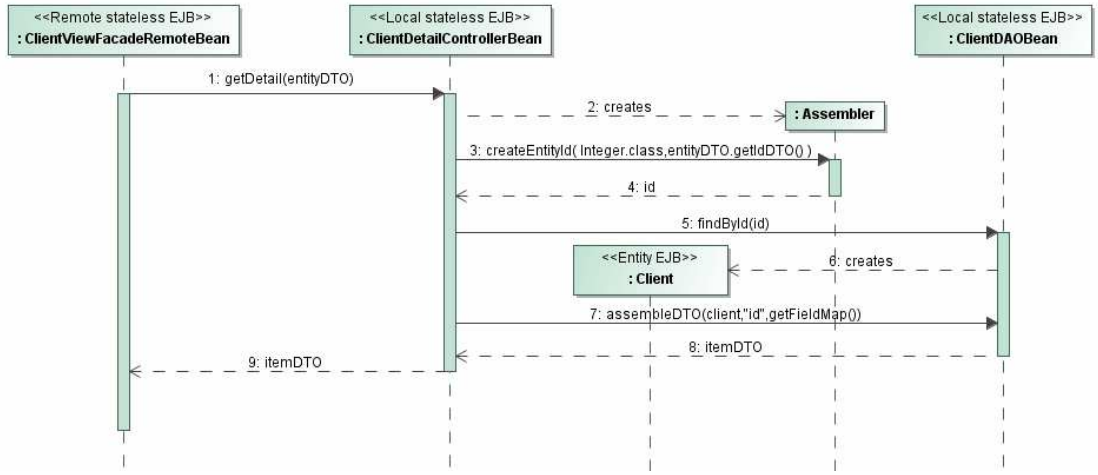


Figura 24 : Diagrama de seqüència detall de client

Finalment, el missatge *save* rebut per bean controlador inicia les crides necessàries dins de la capa de persistència per emmagatzemar efectivament l'entitat modificada.

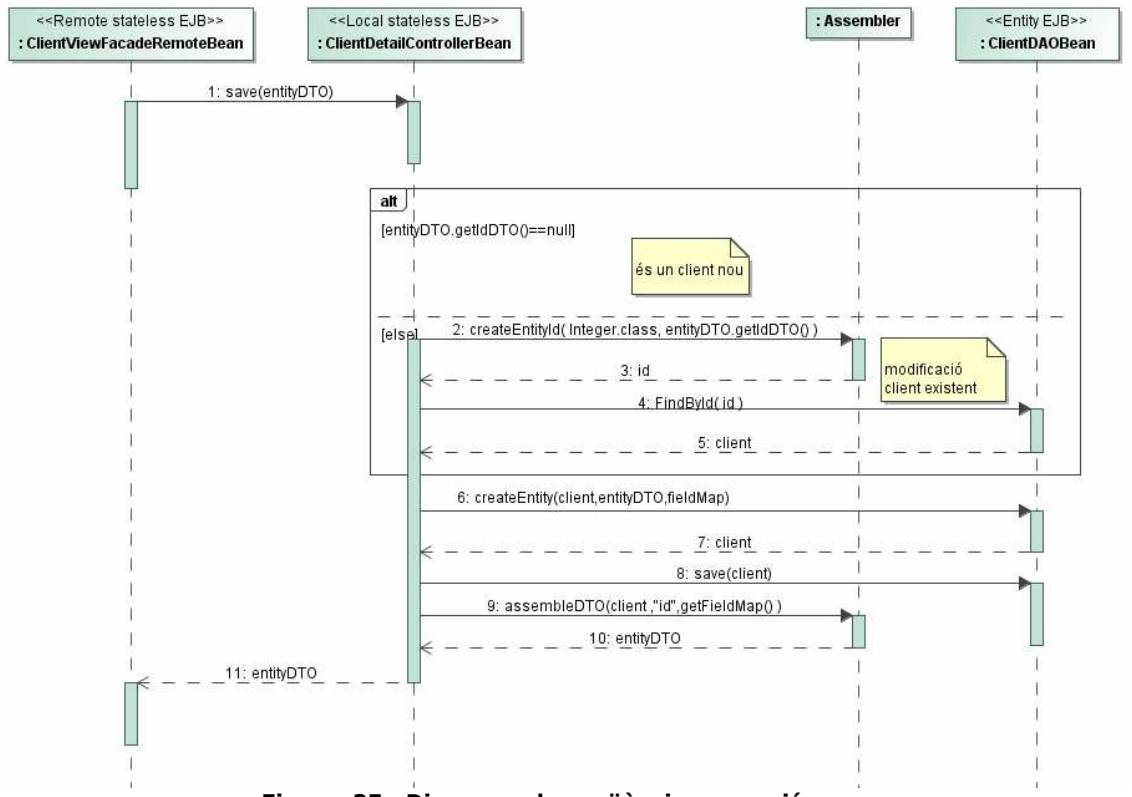


Figura 25 : Diagrama de seqüència operació save

Diagrama de les classes base

Les classes controladores de llistes i detalls implementen el codi comú necessari per tractar les operacions de recerca (bàsica i avançada) , de persistència.
La classe *FieldMap* és un hashmap que permet definir per cada controladora la relació existent entre els camps de la vista i les propietats de les entitats persistents. Aquesta serà utilitzada per la classe *Assembler* per construir el *HashMapDTO* que es passarà a la capa de presentació.
De la mateixa manera, la classe *SearchCriteria* permet configurar les diferents recerques disponibles de les classes controladores de llista.

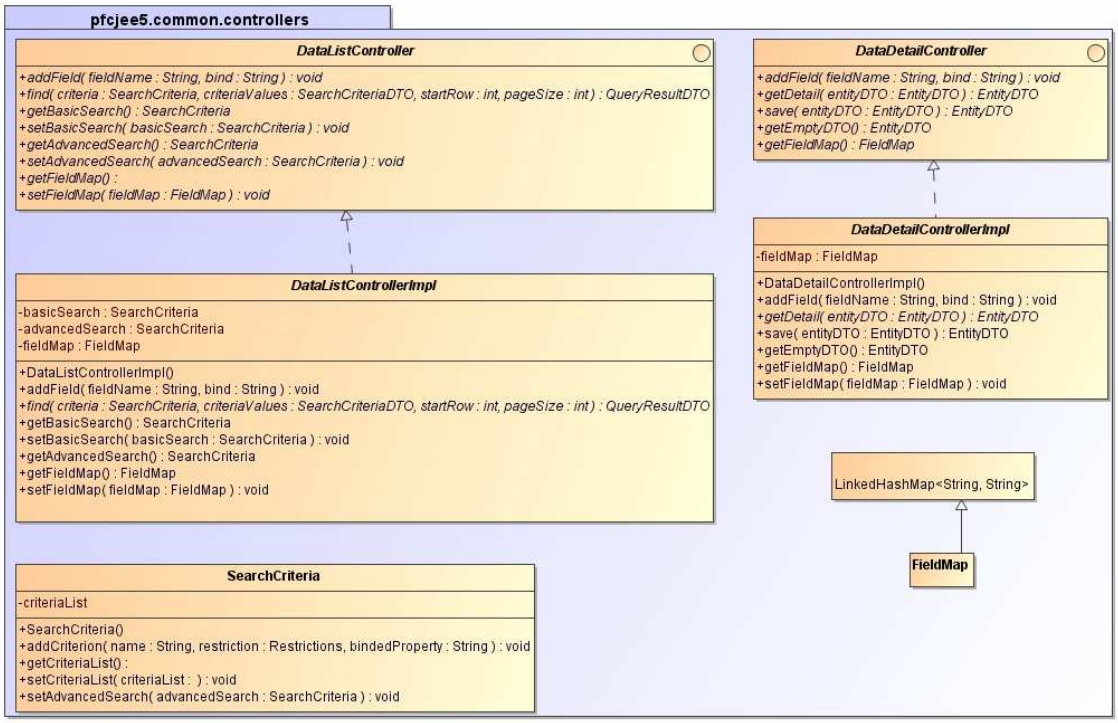


Figura 26 : Classes base gestió controladors

Diagrama de les classes generades

En aquest primer diagrama es mostra les classes i interfícies generats. La classe *ClientViewFacadeLogic* implementa la lògica bàsica de les funcionalitats necessàries per gestionar la vista de clients. Com en el cas de la presentació, també es genera una classe que permet a l'usuari implementar els serveis declarats.

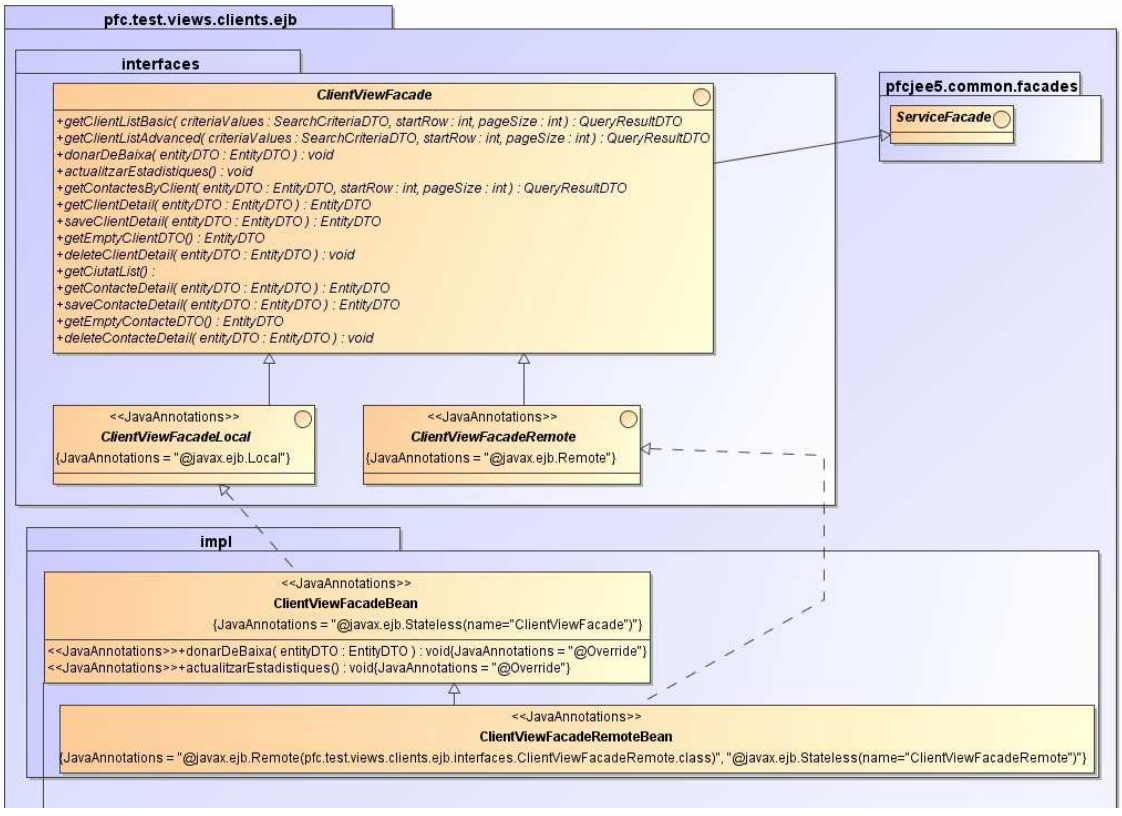


Figura 27 : Façana ClientViewFacade

En aquests altre diagrama es mostra les classes controladores corresponent a clients. (per claredat només s'ha inclòs la part corresponent a clients)

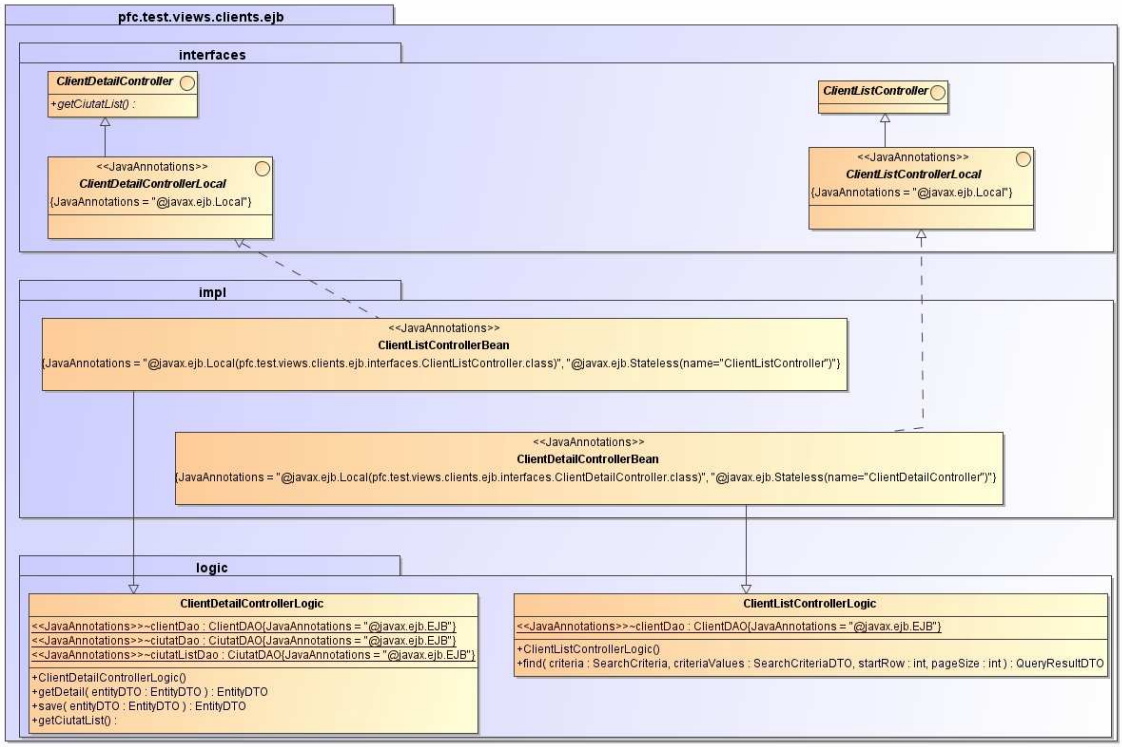


Figura 28 : Classes controladores del model d'exemple

7.2.6 Model de negoci i persistència

Segons el model, es generaran les classes DAO corresponent a cadascuna de les entitats que conformen el model. Totes aquestes hereten de la classe *EntityDAOBean*. Aquesta implementa les funcionalitats bàsiques CRUD.

Construcció de sentències HQL

La classe *HqlBuilder* permet la creació automàtica de les sentencies HQL.

L'automatisme s'aconsegueix de la següent manera:

- A partir del diagrama del model i dins del constructor del controlador de la llista, el generador pot especificar els criteris de recerca. En el cas de la recerca avançada de clients tenim :

```
//- Criteris recerca avançada
this.getAdvancedSearch().addCriterion("codiClient",
      pfcjee5.common.DAO.Restrictions.equal,"id");
this.getAdvancedSearch().addCriterion("nomClient",
      pfcjee5.common.DAO.Restrictions.contains,"nom");
```

Un objecte *Criterion* permet definir un criteri de recerca indicant el camp la propietat del DTO que conté el valor de la recerca, el tipus de restricció a aplicar i, finalment, l'enllaç (*bind*) amb una propietat d'una entitat.

- A partir del *fieldMap* també definit en el mateix constructor i que finalment indica els valors que són necessaris recuperar; l'objecte DTO amb els valors que l'usuari ha introduït per la recerca; i finalment la informació anterior, la classe HqlBuilder genera una cadena de text amb la sentència HQL necessària.

És la classe *EntityDAOBean* la que utilitza la cadena creada per executar la sentència i obtenir els resultats de la consulta.

Paginació per base de dades

De forma automàtica, per cada llista s'implementa paginació per base de dades. En situacions reals aquesta tècnica dona millors resultats quan el volum d'informació és gran, donat que redueix el nombre de files a retornar pel motor de la base de dades i la memòria necessària per mantenir el resultat és inferior.

Per aquest tipus de paginació és necessari d'una banda, executar les consultes establint la fila inicial i el nombre màxim de resultats i de l'altre, a més de les files resultants, cal conèixer el nombre total real de files de la recerca. Per aquesta darrera raó, la classe *HqlBuilder* també retorna una cadena amb una sentència HQL de tipus escalar (*count*).

Finalment doncs, dins del mètode findByCriteria de *EntityDAOBean* es farà una primera consulta limitant els resultats segons els paràmetres *startRow* i *pageSize*, i una segona consulta que recuperarà el nombre real de files dins de la base de dades. El resultat s'emmagatzema dins d'un objecte de la classe *QueryResult.*

Diagrama de les classes base

El conjunt de classes base implementat es mostra en el següent diagrama :

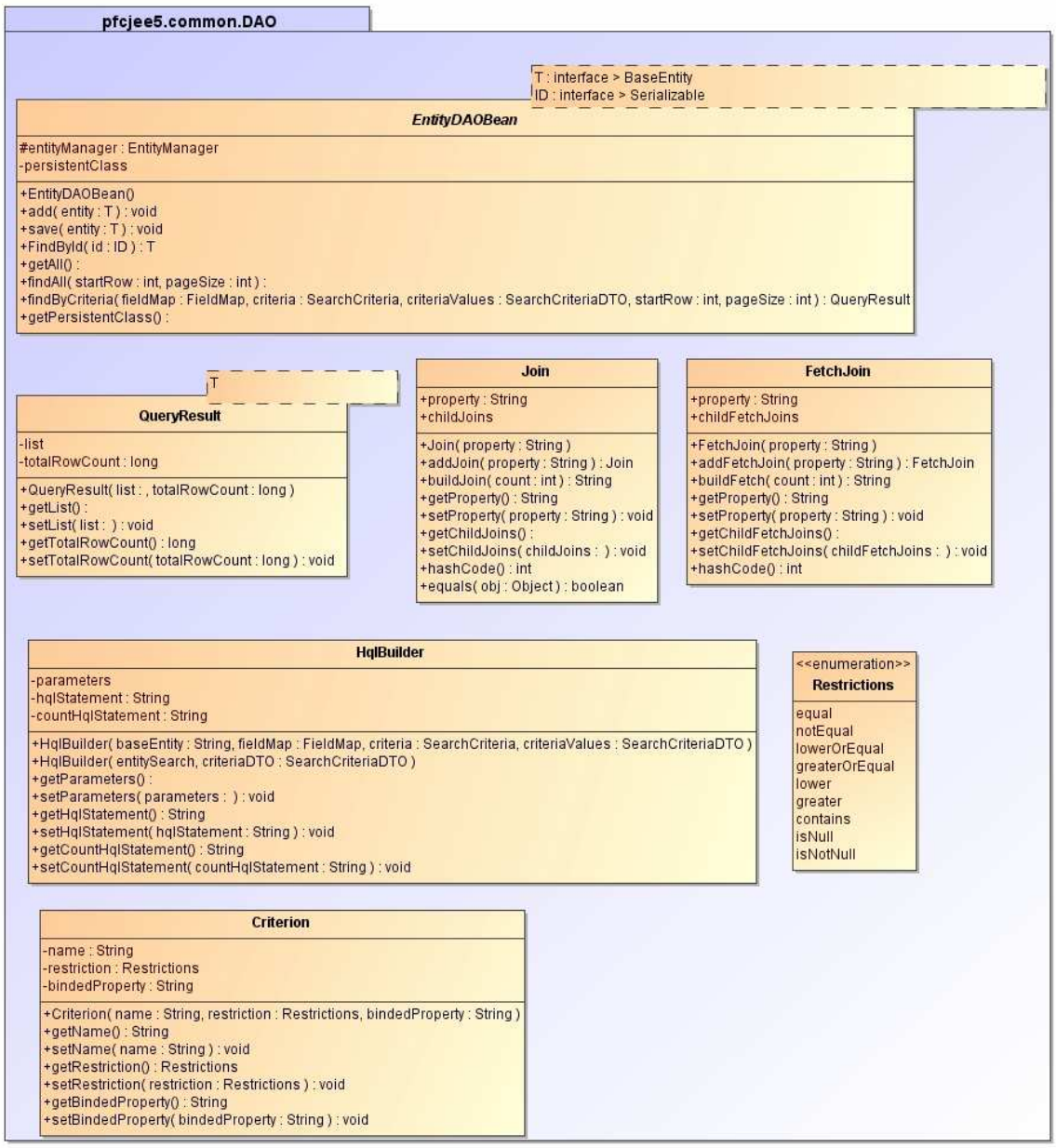


Figura 29 : Classes base per l'accés a dades

Diagrama de les classes generades

D'una banda es generen les classes i interfícies DAO. De l'altre es generen les classes entitats. Aquestes estan anotades segons l'especificació JPA a partir dels diferents estereotips aplicats a les classes i les associacions del model. També es crea la signatura dels mètodes de negoci indicats en el diagrama del model.
En els diagrames següents només s'han representat les classes corresponents a l'entitat *Client*.

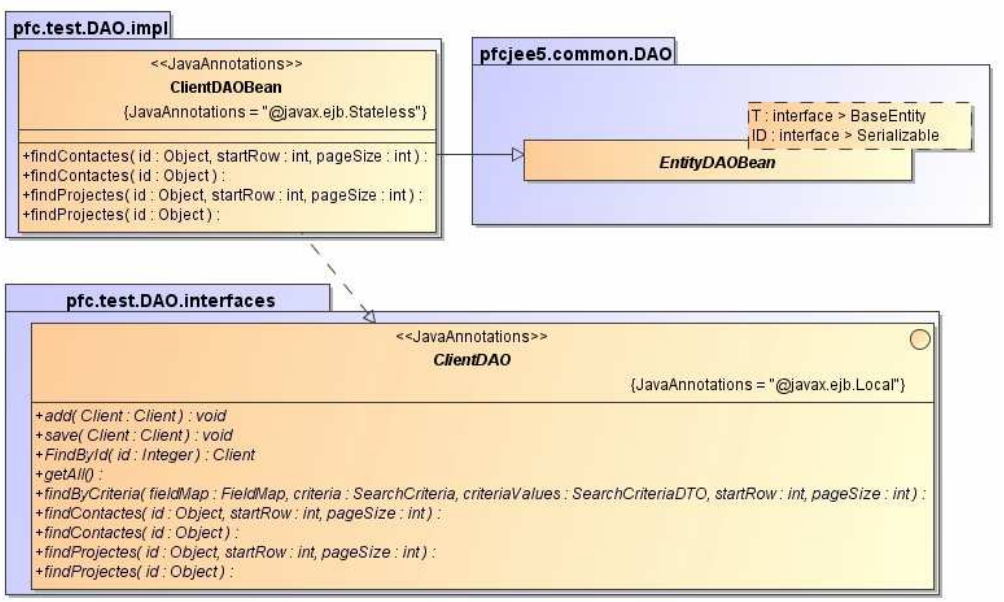


Figura 30 : Classes DAO entitat client

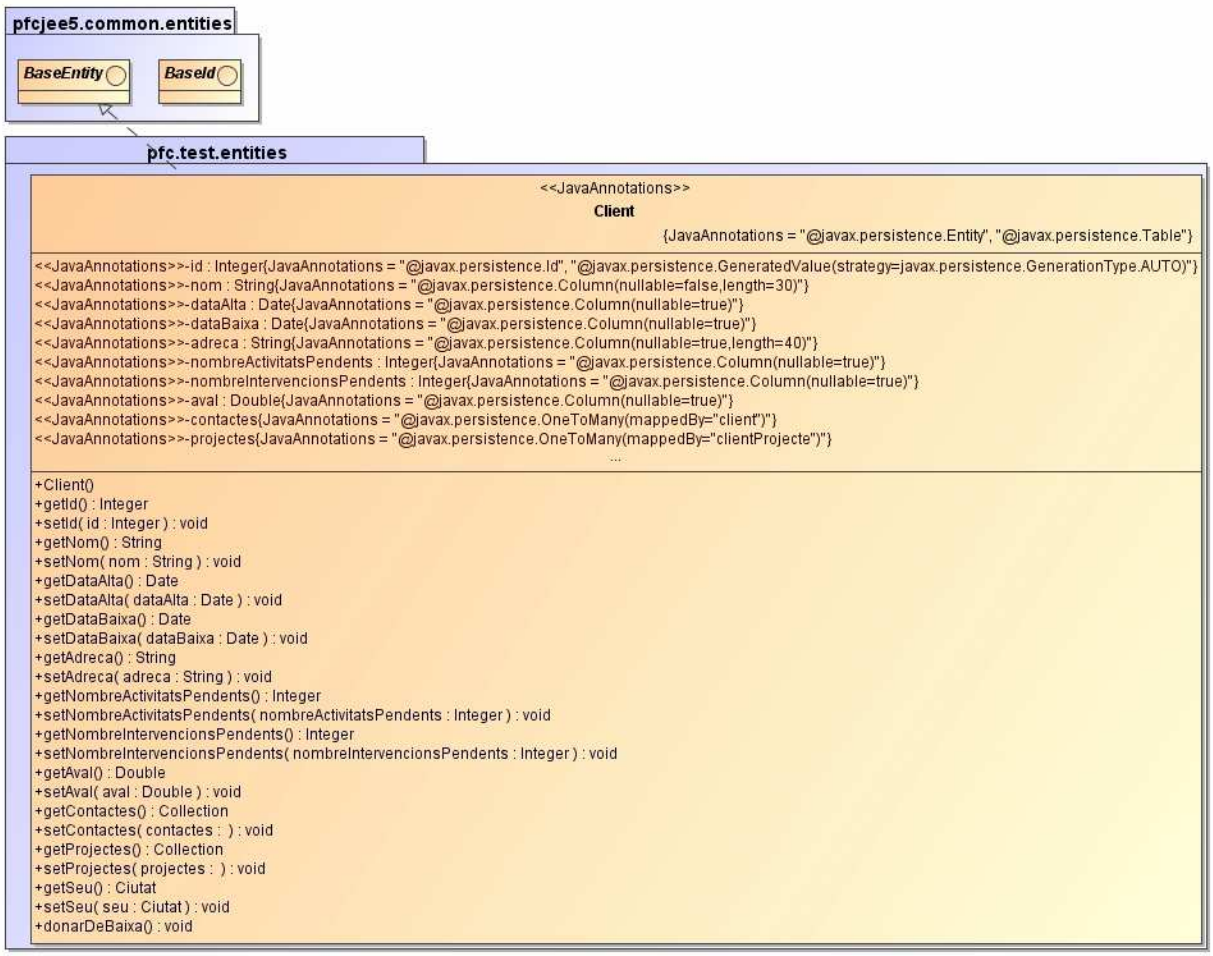


Figura 31 : Entitat client generada

7.2.7 Configuració faces i localització dels serveis remots

A nivell de la capa de presentació el fitxer *faces-config.xml* permet configurar tant la navegació com la creació dels beans. Aquest fitxer és creat automàticament pel generador a partir del diagrama del model.
Com es pot comprovar en el codi de la classe *ClientList* i de la classe heretada *BaseListBean,* les classes de la capa de presentació que representen les llistes i els detalls declaren atributs que fan referència a altres beans. (e.g. *contacteListBean* i *facade*)

```
public class ClientList extends pfcjee5.web.common.BaseListBean{
	private ClientViewFacade facade;
	private ContacteListBean contacteListBean;
…
```

```
public abstract class BaseListBean extends PagedListDataModel{
…
	private DataDetail dataDetailBean;
…
```

La instanciació d'aquestes propietats es realitza mitjançant el mateix fitxer *faces-config.xml* utilitzant la tècnica de la injecció de dependències. En el fragment següent del fitxer de configuració de faces es mostra la part corresponent al bean que controla la llista de clients:

```
<managed-bean>
	<managed-bean-name>clientListBean</managed-bean-name>
	<managed-bean-class>pfc.test.views.clients.beans.impl.ClientListBean</managed-
bean-class>
	<managed-bean-scope>session</managed-bean-scope>
	<managed-property>
		<property-name>facade</property-name>
		<value>#{serviceLocatorBean['ClientViewFacade']}</value>
	</managed-property>
	<managed-property>
		<property-name>dataDetailBean</property-name>
		<value>#{clientDetailBean}</value>
	</managed-property>
	<managed-property>
		<property-name>contacteListBean</property-name>
		<value>#{contacteListBean}</value>
	</managed-property>
</managed-bean>
```

Les propietats *dataDetailBean* i *contacteDetailBean* s'inicialitzen amb la referències creades per contenidor. Aquest mecanisme clarifica el codi en no ser necessari accedir directament al mapa de sessió.
Mereix especial atenció l'assignació corresponent a la propietat *facade* que correspon al EJB3 remot *clientViewFacadeRemoteBean.*
Inicialment, en l'anàlisi s'esmentava que no seria necessari implementar cap classe de tipus *ServiceLocator* per la localització dels diferents EJB. En efecte, el framework EJB3 la facilita utilitzant la tècnica de la injecció de dependencies associada a les anotacions Java. Tot i així, donat que s'ha decidit que el resultat de la generació fossin dos aplicacions separades (una responsable de la presentació i l'altre responsable de la lògica de negoci, model i persistència) , ha estat necessari incloure dins de la capa de presentació un mecanisme per localitzar els serveis remots implementats per la capa de negoci (en l'exemple tractat *ClientViewFacadeRemoteBean*). Per implementar aquesta funció s'ha utilitzat una classe ServiceLocator que segueix una implementació clàssica.

Per permetre utilitzar la injecció de dependencies també amb els serveis remots, s'ha creat el bean *ServiceLocatorBean* que actua a nivell d'aplicació com mostra la declaració dins del *faces-config.xml.*

```
<managed-bean>
      <managed-bean-name>serviceLocatorBean</managed-bean-name>
            <managed-bean-class>
                  pfcjee5.web.common.locator.ServiceLocatorBean
            </managed-bean-class>
      <managed-bean-scope>application</managed-bean-scope>
</managed-bean>
```

Bàsicament, el bean fa de pont entre *ServceLocator* i el motor JSF. En heretar de la classe *HashMap,* es pot fer la localització de serveis remots directament dins del *faces-config.xml* simplement fent #{serviceLocatorBean['ClientViewFacade']}. (cal recordar que amb l'expression language de JSF no es poden passar paràmetres i amb aquest mecanisme es pot indicar fàcilment el nom del EJB remot desitjat)

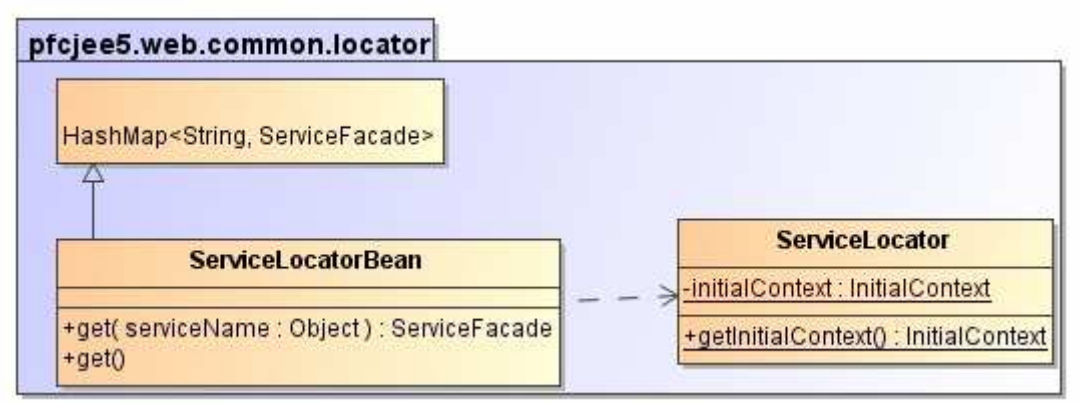


Figura 32 : Classes base per la localització dels serveis remots

El diagrama de seqüència següent resumeix el funcionament explicat :

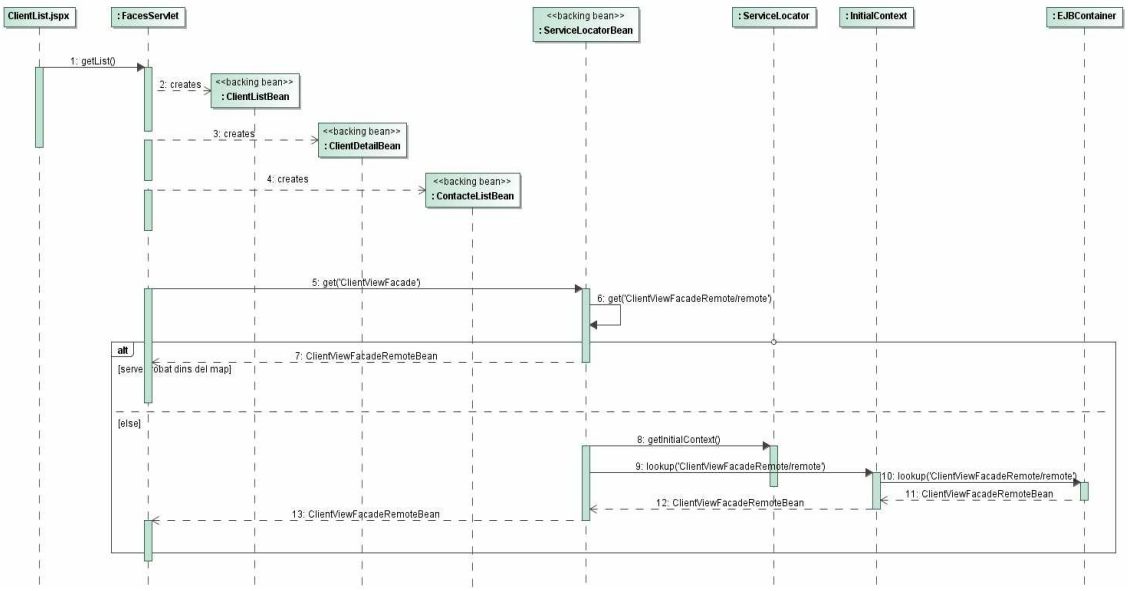


Figura 33 : Diagrama de seqüència inicialització beans

8 CONCLUSIONS

En relació als objectius marcats inicialment, es podria considerar que aquests han estat assolits. Primer, s'ha definit una arquitectura bàsica utilitzant els frameworks escollits (JEE, JSF, EJB i JPA). Aquesta arquitectura pren en compte els elements més usuals de les aplicacions empresarials reals.

Després, s'han especificat els elements i regles de modelatge que serveixen de base al diagrama del model i aplicació. Aquestes especificacions porten a un model relativament simple i no depenent de l'arquitectura final d'implementació.

Com a producte final, s'ha implementat un cartridge AndroMDA que, a partir d'un model dibuixat segons les regles esmentades en el punt anterior, genera gran part del codi necessari per implementar la solució.

Més concretament, s'han pogut implementar de forma automàtica els elements principals següents :

- Navegació i declaració de beans.
- Estructura bàsica de les pàgines Web.
- Validacions i conversions.
- Llistes desplegables.
- Paginació per base de dades.
- Recerca bàsica i avançada : formularis i construcció de les sentències HQL corresponents.
- Funcionalitats CRUD.
- Accés controlat als serveis (confirmació, notificació èxit o error).
- Construcció classes d'entitat segons especificació JPA.
- Arquitectura empresarial.

Altres funcionalitats no implementades que millorarien el producte serien:

- Gestió complerta de la seguretat (com indicat en el disseny)
- Tractar associacions d'herència i agregació/composició.
- Millora de la gestió de les supressions, permetent indicar des del model si s'ha d'efectuar una supressió lògica o real. En el projecte proposat la supressió es gestiona com un servei més, la implementació del qual s'ha de fer posteriorment.
- Gestionar l'ordenació dels resultats de les llistes.

Les funcionalitats incloses en el producte final demostren, al meu entendre, la viabilitat i utilitat d'una arquitectura MDA en l'àmbit empresarial. Evidentment, si bé s'automatitzen aspectes recurrents en qualsevol projecte, el resultat de la generació ha de ser completat de forma tradicional per l'equip de desenvolupadors. Principalment, s'hauran d'implementar les regles de negoci que no es poden expressar directament dins del model i millorar els aspectes lligats a la presentació.

A diferència dels cartridges que es poden utilitzar amb AndroMDA, queda clar que l'objectiu d'aquest projecte no és implementar un altre cartridge d'àmbit general. L'idea principal és, partint d'un diagrama del model relativament simple i deslligat de qualsevol tecnologia de destí donada, generar una implementació concreta i pròpia. Idealment, els beneficis que es podrien obtenir en l'àmbit empresarial no són negligibles. Efectivament, fomenta l'estandardització dels processos de

desenvolupament; evita tasques repetitives que representen un cost considerable i que solen ser font d'errors; millora la reutilització d'arquitectures i no només de classes; millora la qualitat; es pot efectuar una millor avaluació del cost econòmic i temporal d'un projecte.

El problema més important al qual s'enfronta la implantació d'arquitectures orientades al model és trobar un framework adient a les necessitats reals de l'empresa. En el cas estudiat, AndroMDA proporciona *out-of-the-box* un nombre considerable de cartridges que permeten generar codi pels frameworks actuals més usuals. Al meu entendre, sense que sempre sigui un aspecte negatiu, els generadors oferts estan molt lligats a tecnologies concretes. Així, resulta difícil aprofitar un model donat per generar codi en un framework diferent a l'original. També, l'esforç en formació és superior donat que és necessari aprendre els modismes propis a cada cartridge.

Aleshores, una possible solució és la implementació d'un cartridge propi. Aquest ha de recollir tota l'experiència i saber fer de l'empresa per tal d'assolir els beneficis esmentats anteriorment. En el cas concret d'AndroMDA i en el marc d'aquest projecte, he de reconèixer que la implementació d'un cartridge propi no ha estat una tasca fàcil. Efectivament, la disponibilitat de documentació, tutorials i recursos Web no és molt abundant i sovint resulta complicat avançar.

Tot i així, és cert que AndroMDA resolt problemes que altrament s'haurien d'implementar manualment des de zero i que poden ser complexes com ara la interpretació dels models UML en format XMI per tal de generar el PIM i el PSM. Per tant, tot i la seva complexitat, i una vegada es controla el procés, AndroMDA resulta una bona elecció per la construcció d'un generador propi.

9 GLOSSARI

AndroMDA: Framework MDA.

Cartridge: Plantilles d'AndroMDA que permeten generar codi automàticament per una tecnologia de destí concreta.

EJB3: (Enterprise Java Beans versió 3) Components de programari gestionats pel servidor d'aplicacions. Generalment, implementen la lògica de negoci i la persistència. S'hi pot accedir de forma remota o local.

Hibernate: Framework de persistència open-source.

JPA: (Java Persistence API) . API de persistència per JEE. Es tracta només d'una especificació. Algunes implementacions possibles són Hibernate o JDO.

JSF: (Java Server Faces). API de presentació per JEE. Es tracta només d'una especificació. Algunes implementacions possibles són myFaces o IceFaces.

MDA: (Model-driven-architectures). Arquitectura on el disseny i la implementació del programari es fa partint del model. Generalment, el codi es genera automàticament.

PIM: (Platform independent model). Dins d'una estructura MDA, modela les funcionalitats del sistema. És independent de la tecnologia de destí.

PSM: (Platform especific model). Dins d'una estructura MDA, modela la implementació definida en el PIM. Al contrari del PIM, depen de la tecnologia de destí.

XMI: Format XML que permet descriure diagrames UML.

10 BIBLIOGRAFIA I REFERENCIES

BIBLIOGRAFIA

Arlow, Jim; Neustadt, Ila (2005). *UML2.* Madrid: ANAYA.

Bauer, Christian; King, Gavin (2007). *JAVA PERSISTENCE WITH HIBERNATE.* USA: Manning Publications.

Fowler, Martin (2002*). PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE.* USA: Addison Wesley.

Herrington, Jack (2003). *Code Generation in Action.* USA: Manning

Keith, Mike; Schincariol, Merrick (2006). *PRO EJB3. Java Persistente API.* USA: Apress.

Larman, Craig (2003). *UML Y PATRONES. Una introducción al análisis y diseño orientado a objetos y al proceso unificado. Segunda edición*. Madrid: PEARSON EDUCACIÓN.

Marinescu, Floyd (2002). *EJB Design Patterns. Advanced Patterns, Processes, and Idioms*. USA: Wiley

Sun Microsystems (2007). *The Java EE5 Tutorial*. Sun Microsystems

REFERENCIES

Bhatia, Naresh (2006, 21 de juny). *AndroMDA Getting Started (Java).* http://galaxy.AndroMDA.org/index.php?option=com_content&task=category&sectionid=11&id=42&Itemid=89

Bohlen, Matthias (2006, 21 de març). *10 steps to write a cartridge.* http://galaxy.AndroMDA.org/index.php?option=com_content&task=blogcategory&id=35&Itemid=77