Pràctica 3: Grups A i B

Objectius:

L'objectiu principal d'aquesta pràctica és acabar d'aplicar els patrons GRASP i SOLID juntament amb el patró per capes i l'arquitectura clàssica de Model-Vista-Controlador. A més a més, es pretén aplicar uns primers patrons de disseny a la pràctica: el patró Singleton, el patró Façana, el patró Strategy i/o el patró de Factory (alguna de les seves variants), si és que són necessaris.

Enunciat de la pràctica 3:

Seguint amb les funcionalitats relatives a marcar les activitats com a preferides i valorar les activitats realitzades per cada Soci que tenies desenvolupades a la pràctica 2, cal que afegeixis alguna una altra història d'usuari:

- Es vol poder valorar l'activitat no només usant estrelles, sinó que també es vol poder valorar l'activitat per likes/disikes. Això comporta diferents funcionalitats: (1) poder valorar l'activitat segons les diferents maneres d'opinar i (2) poder extreure la llista de les 10 més valorades amb estrelles, la llista de les 10 activitats amb més likes, la llista de les 10 activitats pitjors valorades, etc....

Tingues en compte que l'aplicació podrà estar offline (sense connexió externa) i per tant, totes les dades hauran d'inicialitzar-se a memòria. Només caldrà estar online en el moment que es registre o es fa un login d'un Soci.

Aquesta pràctica consta de tres parts:

1. Redissenyar la pràctica 2 per tal d'acabar d'aplicar correctament els patrons GRASP i SOLID.
2. Analtzar el codi de CeXtrem-DAO de la pràctica 2, localitzant els patrons de disseny que s'han utilitzat
3. Aplicar els patrons de disseny explicats a teoria (**el patró Singleton, el patró Façana, el patró Strategy i/o el patró de Factory (alguna de les seves variants)**) allà on ho vegis necessari

Per a realitzar aquesta pràctica es seguirà el Model-Vista-Controlador explicat a classe de teoria, desenvolupant la part del Model i del Controlador. La part de la Vista es fa a partir dels tests de Concordion. Així, l'output d'aquesta pràctica no serà mitjançant la consola o una finestra gràfica, sinó que serà mitjançant els tests d'acceptació que vosaltres heu programat amb Concordion. Les parts de

Controlador i Model s'han d'implementar en dues carpetes separades (controller i model, respectivament) dins de la carpeta src del projecte. La part del projecte que té relació amb la capa de recursos es guardarà en una carpeta interior a src, anomenada resources.

PART 1: Refactoring seguint els criteris GRASP I SOLID (en relació a la pràctica 2)

Segueix els següent passos i contesta els següents punts:

1. **Llegeix els comentaris/observacions de la correcció de la pràctica2 i intenta acabar d'aplicar els patrons GRASP per aconseguir baix acoblament i alta cohesió, repassant els tests d'acceptació. Segueix les indicacions de la correcció i llista a continuació els canvis que has introduït junt amb el nou diagrama de classes que has generat. Només inclous a la llista, els canvis nous que incloguis.**
    a. **Inclou aquí la llista de canvis:**

Test d'acceptació	Criteris GRASP	Breu explicació de les classes canviades
Exemple: Test 2.2 Login erroni per correu erroni	Expert	Delego al controlador el test i després via CeXtrem, delego a la cartera de socis el cercar el correu
Desacoblament controlador i model amb una classe CeXtrem **(tots els tests l'utilitzen)**	Baix Acoblament i Indirecció	Hem mogut les classes gestores (***GestorActivitats, GestorEspecies, GestorExcursions, GestorSoci***), del paquet ***model*** al paquet ***controller*** ja que aquestes ens feien la funció de 'subcontroladors' per tal de desacoblar el controlador (que en el nostre cas és com si fos la classe ***CeXtrem***). Així aconseguim un baix acoblament i indirecció (ja que hem reduït l'acoblament creant una classe que els comunica, el nostre controller principal) del controlador.
Baixa cohesió controlador **(tots els tests l'utilitzen)**	Alta cohesió	Degut al canvi de l'anterior fila (dels anteriors criteris aplicats amunt), el nostre controlador ara té una alta cohesió ja que té els mètodes

		justos i necessaris, té poques línies de codi i delega la seva feina als nostres 'subcontroladors' (les classes gestores indicades anteriorment).

A més d'haver corregit la part dels criteris GRASP seguint les indicacions sobre els comentaris de la pràctica 2, hem aprofitat per també corregir els errors que havíem comès en l'apartat d'implementar els DAO:

1. L'enllaç entre ***DAOExcursions*** i ***DAOEspecies*** no el podem fer directament quan duem a terme la inicialització de les dades en el DAO, ja que com que son classes ***DAOMOCK*** que simulen una base de dades, en aquesta no podem guardar altres objectes que no siguin *POJO* (*Plain Old Java Object*), és a dir, objectes de tipus bàsic de *Java* com podrien ser *String*, *Integer*...

   Per tal de solucionar aquest problema el que hem fet és relacionar les espècies i les excursions amb un ID que identifica l'excursió apart del seu nom (ara les espècies apart del seu nom, contindran el seu ID, i un ID de l'excursió a la que està relacionada). D'aquesta manera podem saber com estan relacionades espècies i excursions i les juntarerm al ***DataService***, posteriorment a haver inicialitzat les dades. Ho fem a la classe ***DataService*** ja que té sentit al ser aquesta la classe que conté tota la informació de les classes a inicialitzar/ajuntar (podríem dir que és l'experta).

2. Hem finalitzat la implementació de totes les classes del projecte dels DAO, fent que només es guardin a la base de dades (MOCK) objectes bàsics de *Java*. Posteriorment inicialitzem totes les carteres al ***DataService***.
3. Totes les dades s'inicialitzen des de les classes ***DAOMOCK*** (que és com si fos la nostra base de dades), però la posterior gestió d'aquestes dades es farà des de les carteres. Tot i així, el *login* i el registre del ***Soci*** es realitzaran fent ús del DAO i no de les carteres. Per tal de fer això continuarem fent servir el nostre ***GestorSoci*** per tal d'accedir al ***DataService*** (és la classe experta en les dades) a comprovar totes les dades necessàries del Soci.

També hem aprofitat per afegir una classe ***CeXtrem*** que contindrà totes les carteres dels objectes de la pràctica, per tal d'alliberar de responsabilitat als gestors.

b. Inclou aquí el nou Diagrama de classes amb els nous canvis:

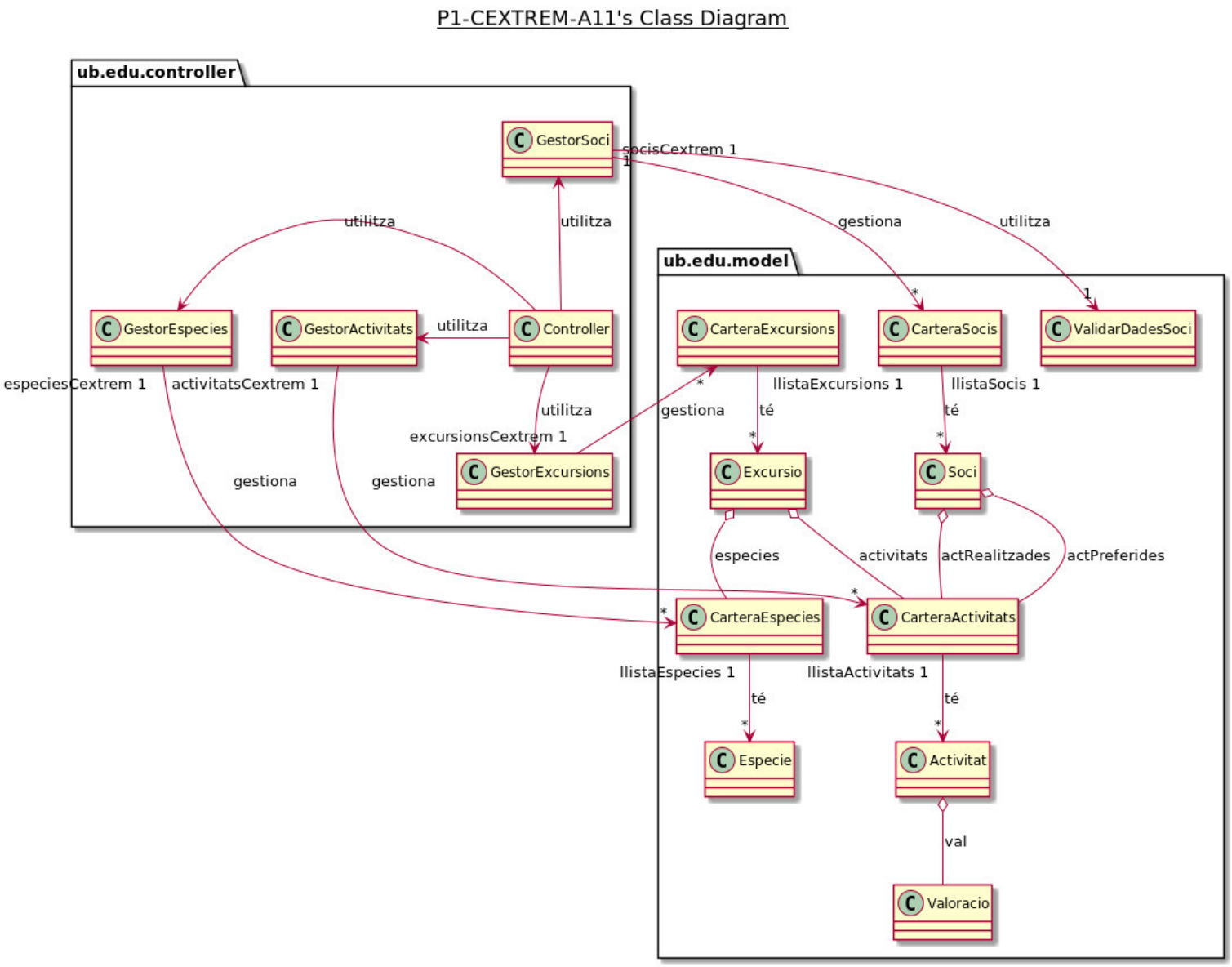


Figura 1: Diagrama de classes (general pràctica 3)

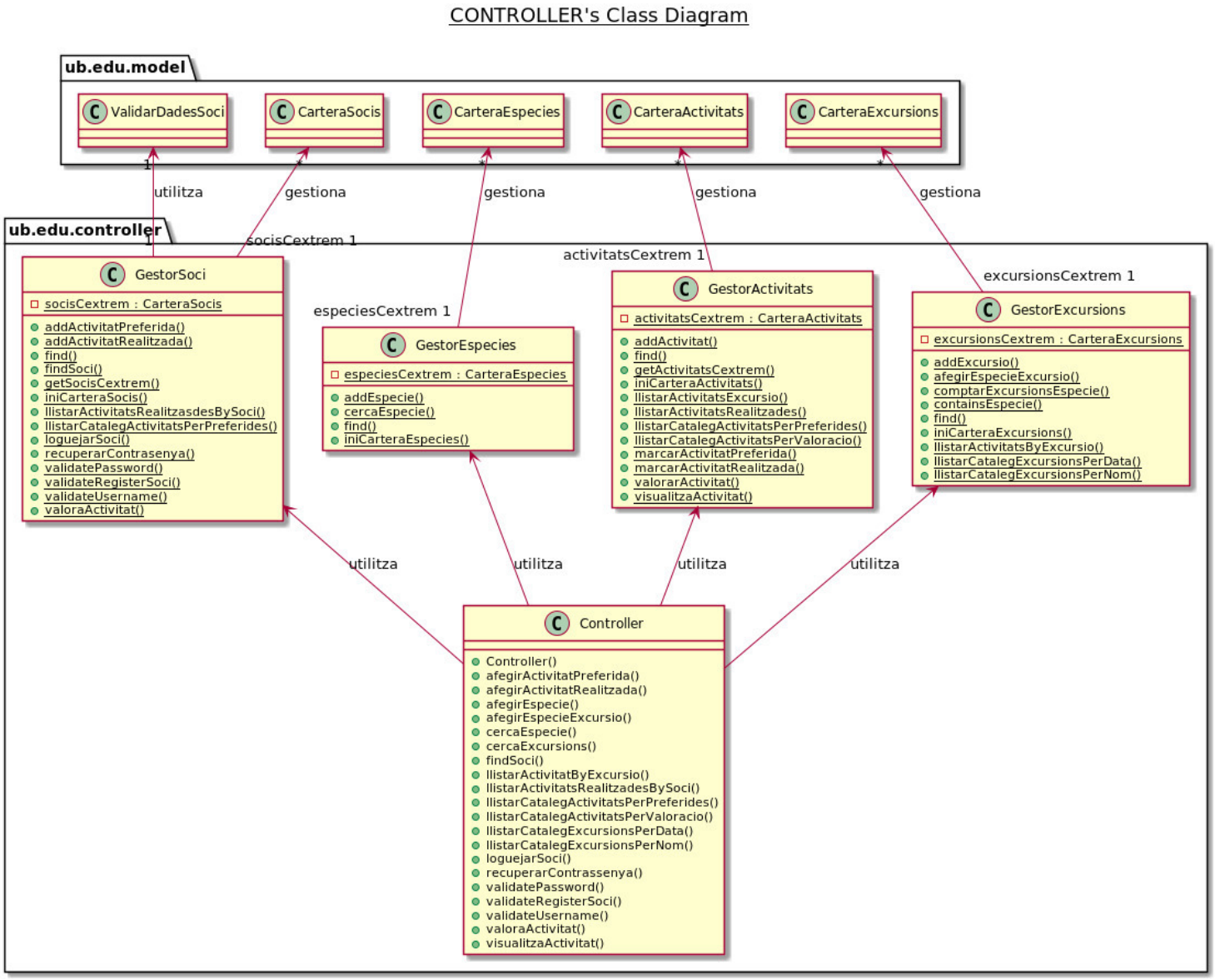


Figura 2: Diagrama de classes (controlador)

MODEL's Class Diagram

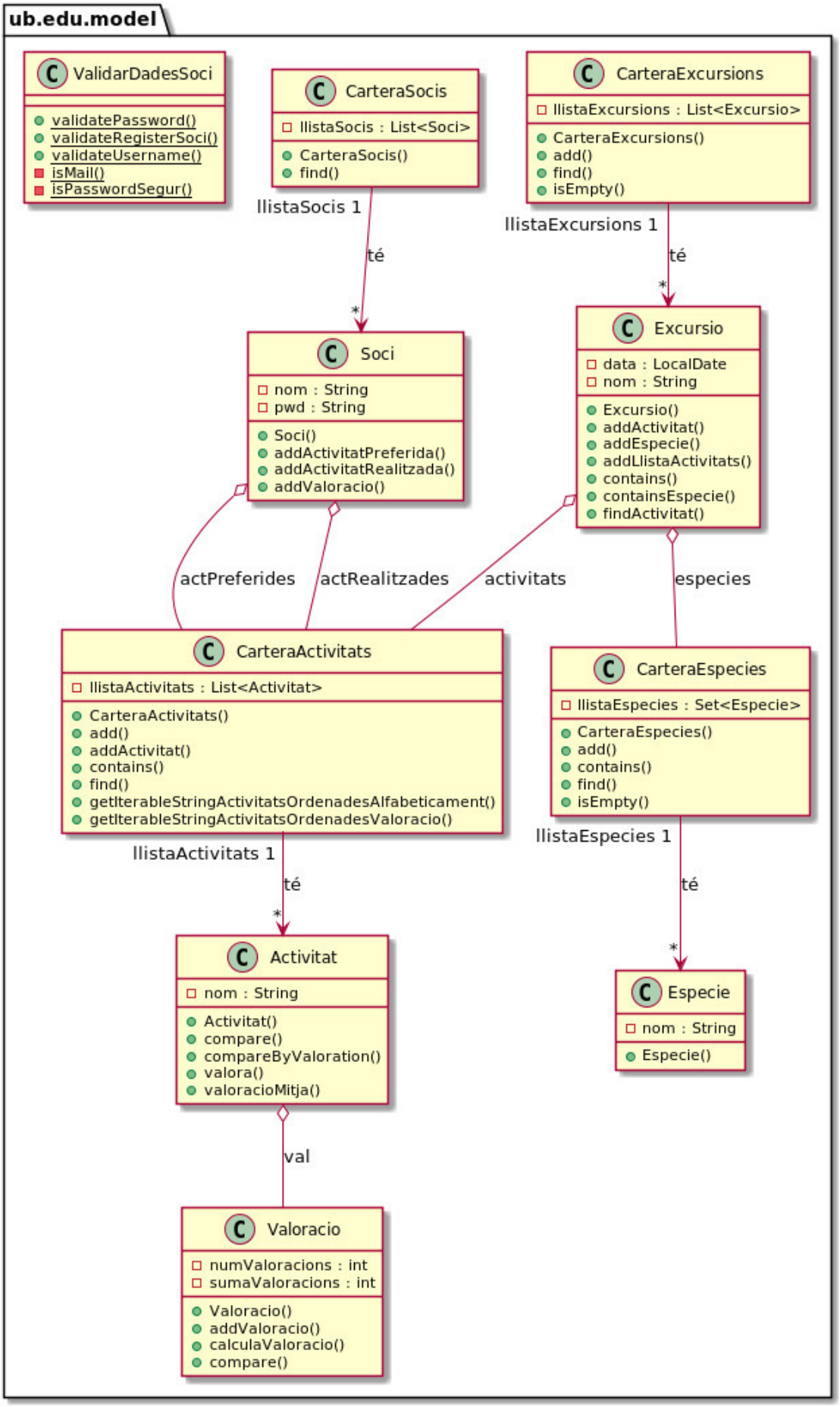


Figura 3: Diagrama de classes (model)

2. **Detecta vulneracions de principis SOLID:**
    a. **Identifica les vulneracions que has detectat en el codi, indica el per què de la vulneració del principi i digues com les has arreglades:**

Classe	Principi SOLID que vulnera	Justificació de la vulneració	Breu explicació de les classes canviades
Exemple: Controlador	Single Responsability Principle	El Controlador té més d'una responsabilitat ja que serveix de comunicació de la vista al model però també comprova la correctesa de les contrasenyes, inicialitza les dades, etc.	Creació d'una nova classe especialitzada en contrasenyes, creació d'una nova classe per inicialitzar les dades (un nou minicontrolador).....
Valoracio	***Open-Closed Principle***	Si volguéssim afegir nous tipus de valoracions, vulneraríem el principi ***Open-Closed*** ja que hauríem de distingir els diferents tipus de valoracions amb estructures *if, else if...*	Solucionat implementant el patró ***STRATEGY*** a la part 3 de la pràctica.
Valoracio	***Dependency Inversion Principle***	Si volguéssim afegir un nou tipus de valoració, això ens faria canviar la classe ***Valoracio***. És a dir, estaríem fent que l'abstracció valoració depengués dels detalls de les implementacions dels diferents tipus de valoració.	Solucionat implementant el patró ***STRATEGY*** a la part 3 de la pràctica.
Controller	***Single Responsability***	La nostra classe ***Controller***, a més d'inicialitzar les dades feia de pont de l'aplicació fent les crides als diferents mètodes dels diferents gestors. Vulnerava així el ***Single Responsability Principle***, ja que una classe ha de tenir una i només una responsabilitat.	Ara la inicialització de les dades es fa des d'un altre *subcontroller* (***GestorBaseDades***) que inicialitza les dades generals de la pràctica (socis, excursions, activitats, espècies) a les carteres d'aquests objectes guardades en la classe ***CeXtrem*** (conté les carteres amb les dades generals de la pràctica).
Gestors	***Single Responsability***	Els nostres gestors guardaven les carteres amb les dades generals de la pràctica, vulnerant així el ***Single Responsability Principle*** (una classe ha de tenir una i només una responsabilitat).	Ho hem solucionat amb la creació d'una classe ***CeXtrem*** que guardarà les dades generals de la nostra pràctica.

b. Inclou aquí el nou diagrama de classes obtingut.

Aquí inclourem el diagrama de classes final de la pràctica, un cop solucionades i detectades les vulnerabilitats ***SOLID*** i després d'haver aplicat els patrons de la ***Part 3*** de la pràctica. Per una millor distribució de les imatges, o hem dividit en tres:

1. Diagrama de classes general de la pràctica 3.
2. Diagrama de classes del paquet ***Controller***.
3. Diagrama de classes del paquet ***Model***.

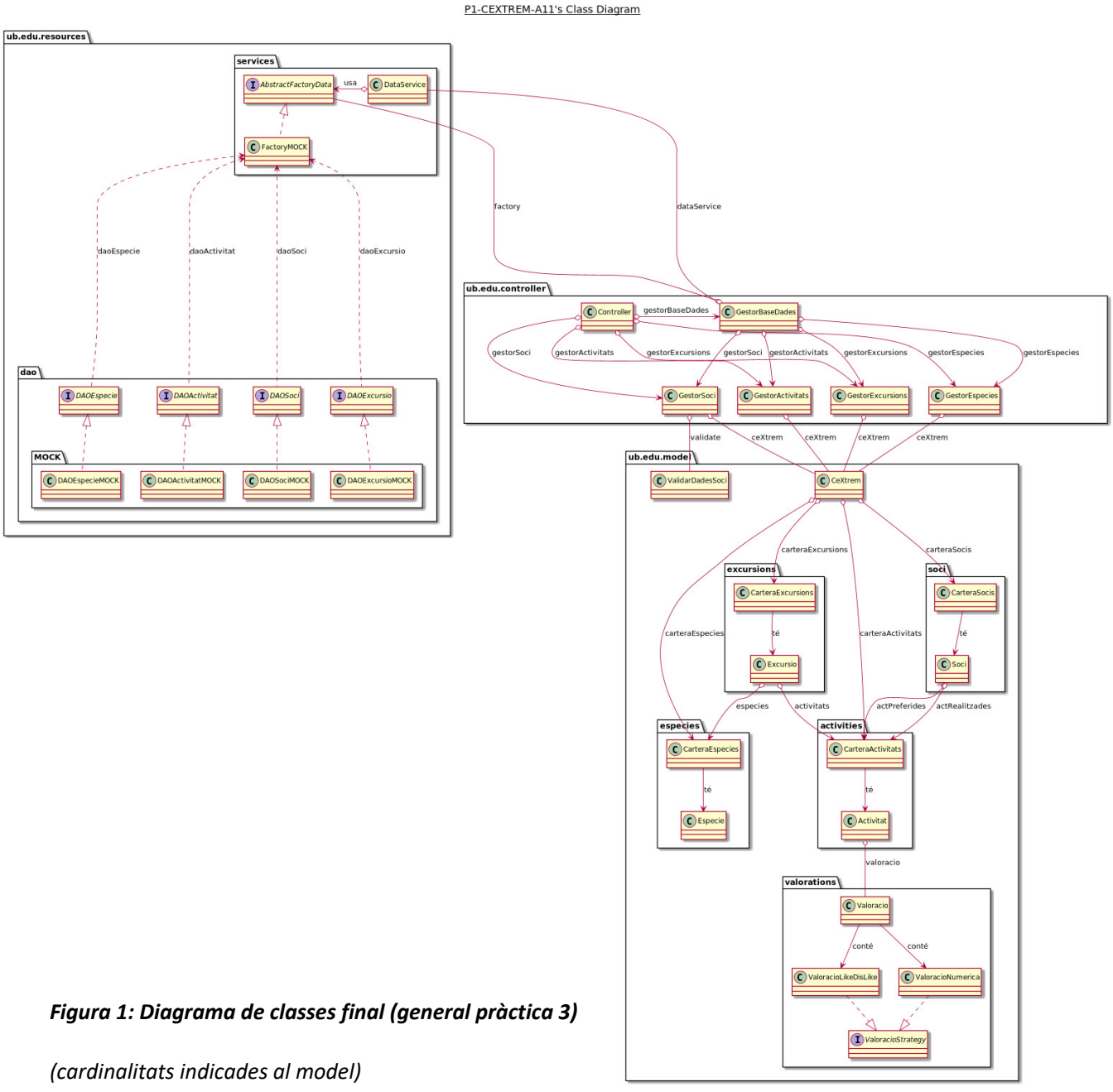


Figura 1: Diagrama de classes final (general pràctica 3)

(cardinalitats indicades al model)

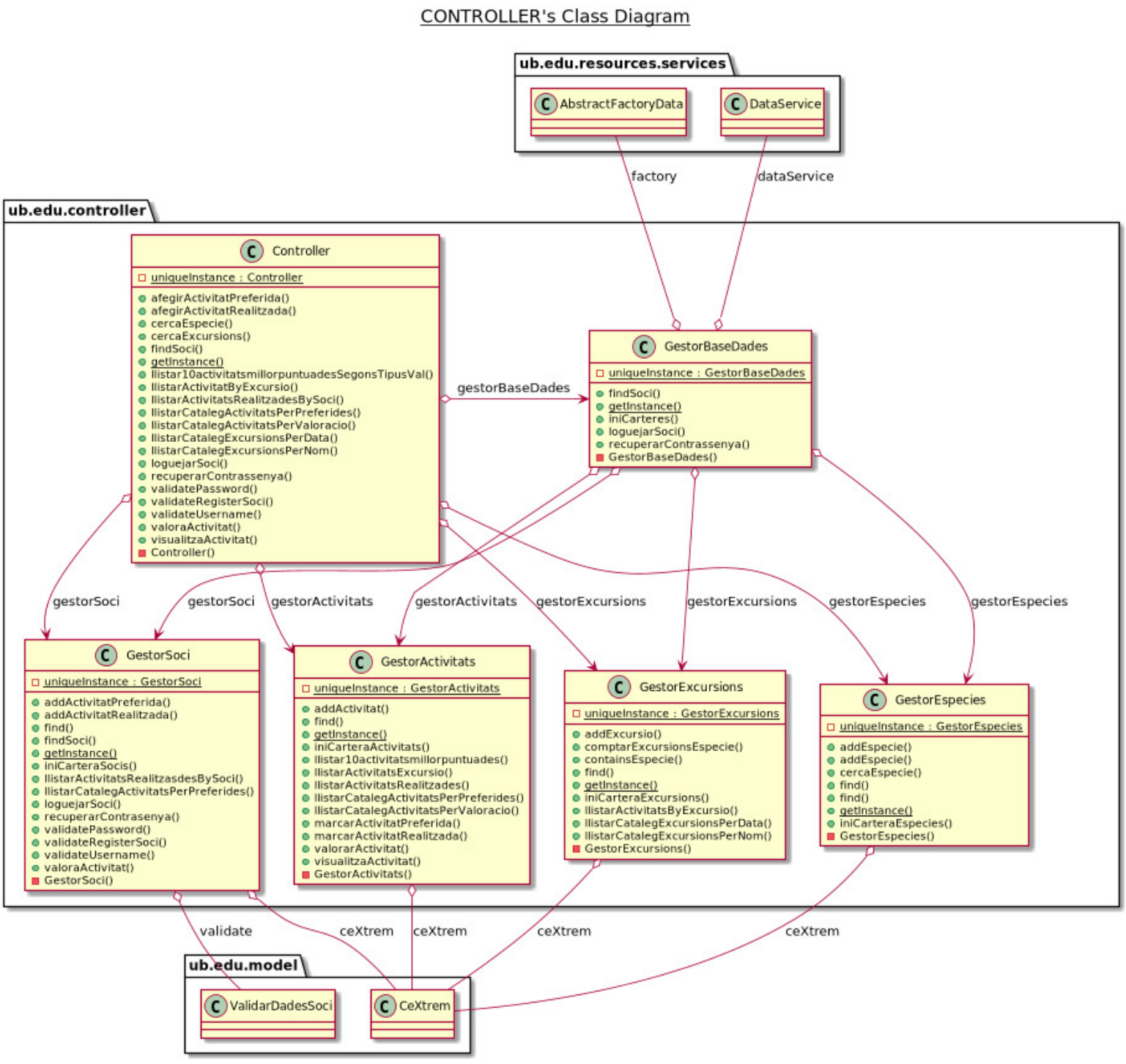


Figura 2: Diagrama de classes final (controlador)

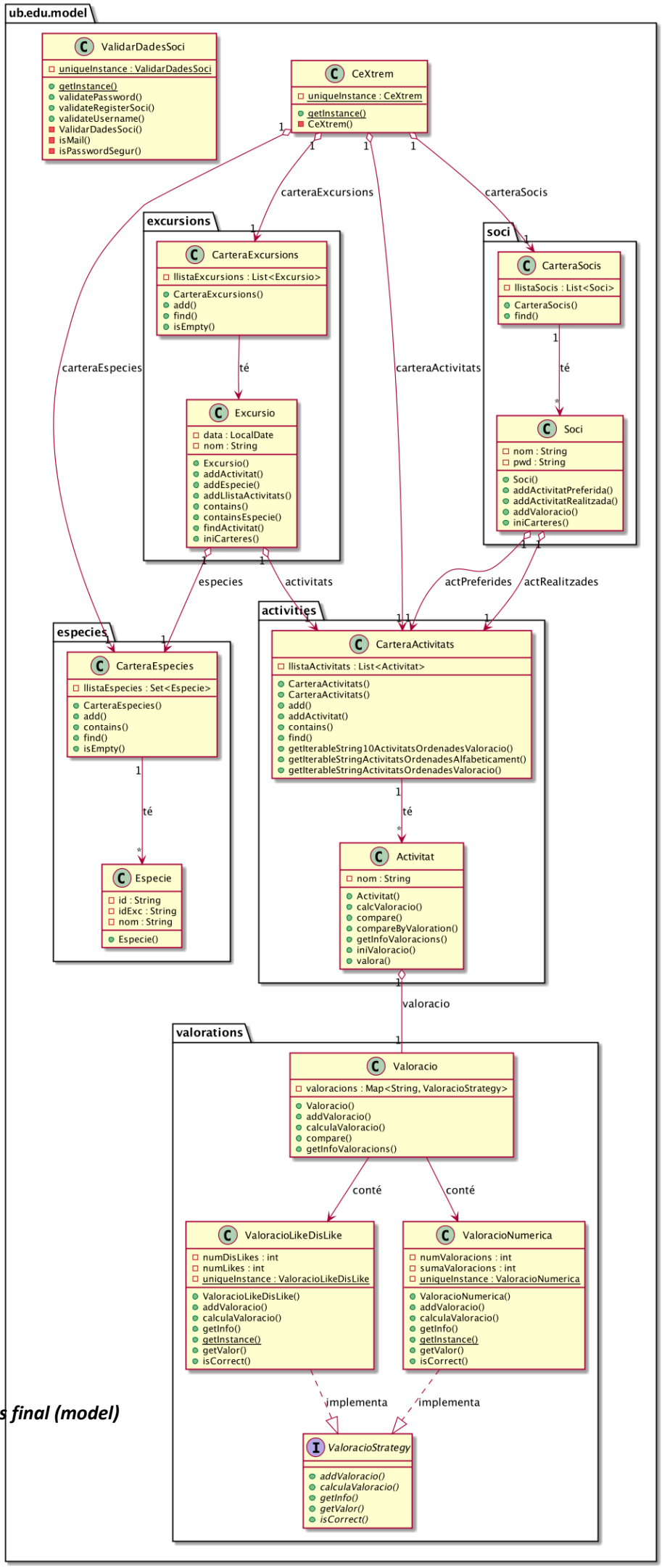


Figura 3: Diagrama de classes final (model)

PART 2: Analitzant els patrons de disseny a la capa de recursos

En aquesta part, exploraràs els patrons utilitzats a la capa de recursos:

1. Si no tens els DAO's integrats a la teva pràctica 2, clona el projecte base que trobaràs al classroom:

   https://classroom.github.com/a/Ys-xd_Cj

2. Explora com el projecte s'estructura en una arquitectura per capes:

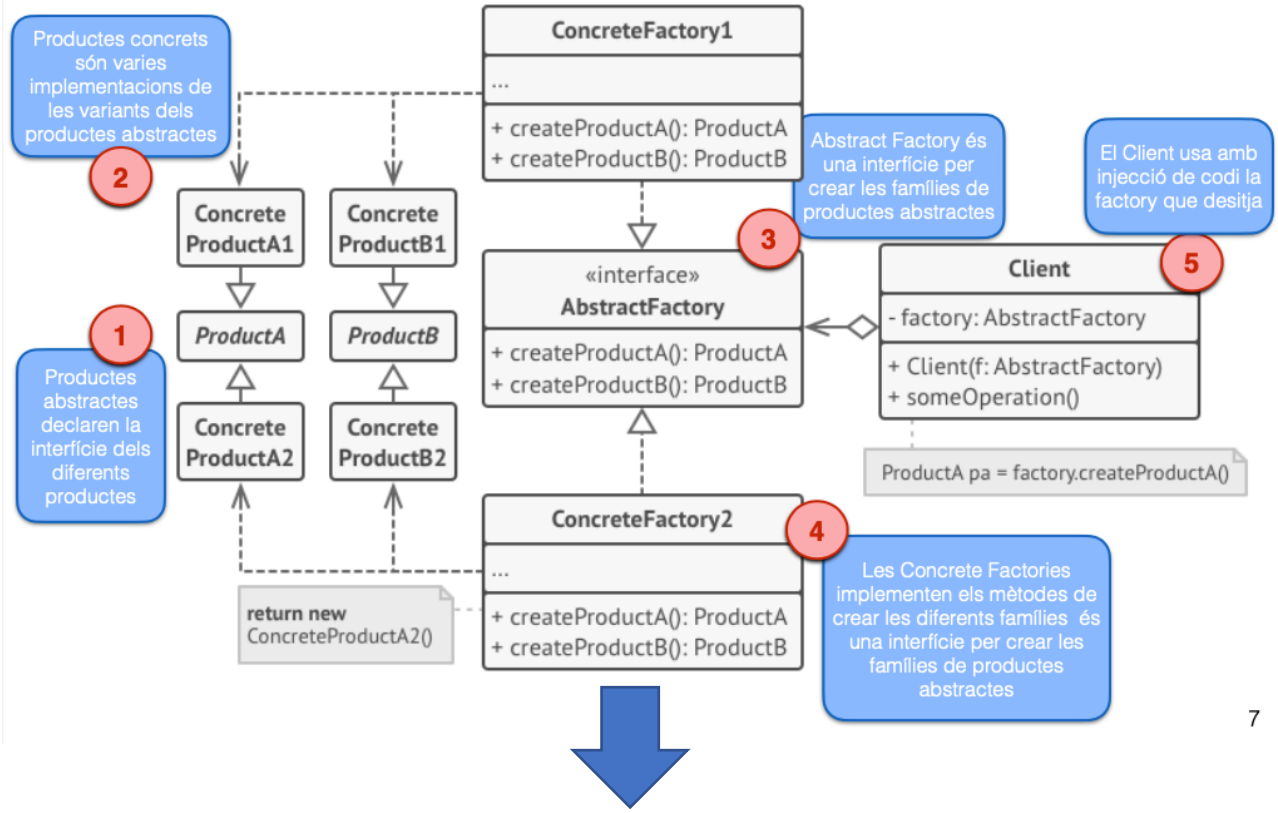
   - El software es basa en una arquitectura en tres capes: (1) la capa de vista que és la que correspon als tests, (2) la capa de lògica de negoci (on està el model i el controlador) i (3) la capa de recursos o persistència, que és l'encarregada de guardar les dades.
   - En el repositori de github es proporciona un codi de la part de persistència (o recursos) que cal utilitzar i ampliar. En aquesta part s'utilitzen els patrons de disseny d'AbstractFactory i DAO per a poder canviar fàcilment la capa de persistència.

   a. Inclou aquí el patró genèric d'Abstract Factory explicat a teoria (pag. 7 de les transparències Tema3.5 – Patrons de Disseny: Abstract Factory) i identifica cadascuna de les classes del patró amb les classes de la capa de persistència: quina classe és l'Abstract Factory? Quina son les ConcreteFactories? Qui son els ConcreteProducts? Qui és la classe Client?

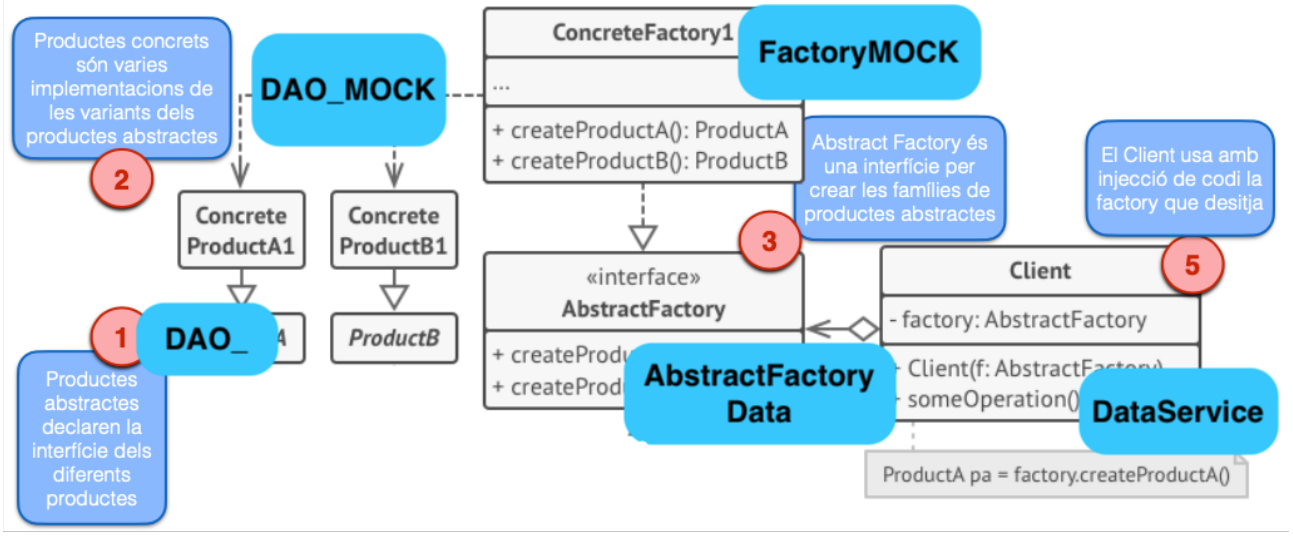
   Pots incloure aquí el diagrama de classes del projecte identificant o ressaltant cadascuna de les classes del patró.

   El patró genèric i el patró modificat indicant les classes de la nostra pràctica s'indiquen a continuació:

Patró Abstract Factory



7



Identificació dels diferents components del patró:

- ***AbstractFactory***: A la nostra pràctica és la interfície ***AbstractFactoryData***. És la interfície encarregada de crear els productes abstractes de la nostra pràctica.
- ***ConcreteFactories***: En el nostre cas és la classe ***FactoryMOCK***. És la que implementa els mètodes de crear les diferents famílies de productes abstractes.
- ***ConcreteProducts***: A la pràctica són els diferents ***DAOMOCK***: ***DAOActivitatMOCK***, ***DAOEspecieMOCK***, ***DAOExcursioMOCK***, ***DAOSociMOCK***. Són vàries implementacions dels productes abstractes.
- ***Client***: En el nostre cas és la classe ***DataService***. Aquesta utilitza amb injecció de codi la *factory* que desitja.
- ***Productes Abstractes***: Són les nostres interfícies ***DAO***: ***DAOActivitat***, ***DAOEspecie***, ***DAOExcursio***, ***DAOSoci***. Els productes abstractes declaren la interfície dels diferents productes.

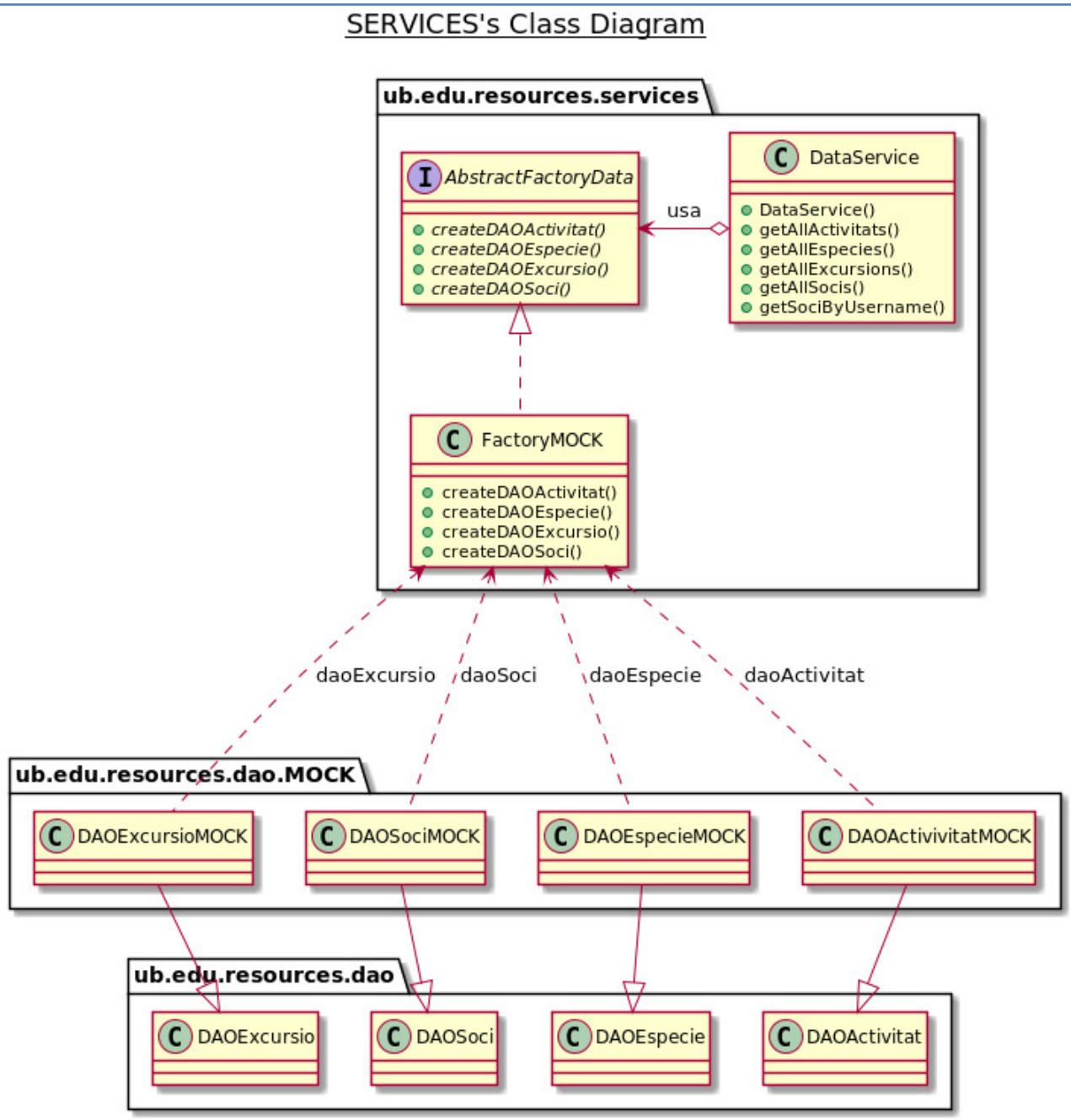


Figura 4: Diagrama de clases (Patró Abstract Factory)

PART 3: Aplica patrons de disseny a la teva implementació

1. **Explora el codi que has desenvolupat fins la pràctica 2 i explora si hi ha algun dels patrons explicats a teoria que puguis aplicar. Omple la taula següent per explicar-ho:**

Patró	Quin problema soluciona?	Classes implicades	Breu explicació de les classes canviades
Singleton	Només cal una instanciació del controlador	***Controller***	Codificació del ***Controller*** com a singleton. Canvi en els tests per a accedir a aquesta instància única del controlador.
Singleton	Només cal una instanciació dels gestors	***GestorActivitats***, ***GestorEspecies***, ***GestorExcursions***, ***GestorSoci*** i ***GestorBaseDades***	Codificació dels diferents gestors seguint el patró de ***Singleton*** per tal d'accedir als seus mètodes sempre que ens sigui necessàri (permet accés global a les dades des de diferents llocs de l'aplicació) i a més perquè té sentit que el nostres gestors ('subcontrollers') tinguin només una instància a cada execució del programa.
Singleton	Només cal una instanciació de la classe ***ValidarDadesSoci***	***ValidarDadesSoci***	Ens interessa només tenir una única instància de la classe ***ValidarDadesSoci*** ja que només el fem servir pel que el propi nom indica, per validar les dades del ***Soci***. Això ens permet tenir accés global als mètodes d'aquesta classe per tal de comprovar que el *login* i el *registre* d'un soci és correcte.
Strategy	Teniem classes molt similars que només variaven en la forma de comportar-se.	***Valoracio***, ***ValoracioLikeDisLike***, ***ValoracioNumerica***, ***ValoracioStrategy***	(a l'apartat on s'implementa la nova funcionalitat es podrà visualitzar més gràficament els canvis realitzats) S'ha afegit una interfície ***ValoracioStrategy*** que és qui declara la interfície comuna a tots els algorismes. Les classes ***ValoracioLikeDisLike*** i ***ValoracioNumerica*** són les implementacions concretes de les estratègies. Per últim la classe ***Valoracio*** és el nostre *Client*, és qui crea l'estratègia concreta que vol fer servir i a més guarda els tipus de ConcreteStrategies que hi ha al sistema per tal d'evitar encara més vulnerar el principi ***Open-Closed***.
Facade	Conjunt d'utilitats d'un	***Controller***	La nostra classe Controller realitza el paper de façana on poden haver-hi 1 o més clients

	subsistema complex.		que accedeixin a ella (en el nostre cas accedeixen els testos, la vista). Els clients que accedeixen a la façana per fer ús de les seves funcionalitats no necessiten saber com funciona el subsistema (una mica més complex) format pels diferents gestors.

2. **Afegeix la nova funcionalitat de valorar per Likes/Diskes i les funcionalitats de llistar les 10 activitats del sistema amb més likes i les 10 activitats millor puntuades. Quin patró podries aplicar aquí? Per què?**

   a. **Identifica el/s patró/ns de disseny que has usat i escriu aquí la justificació.**

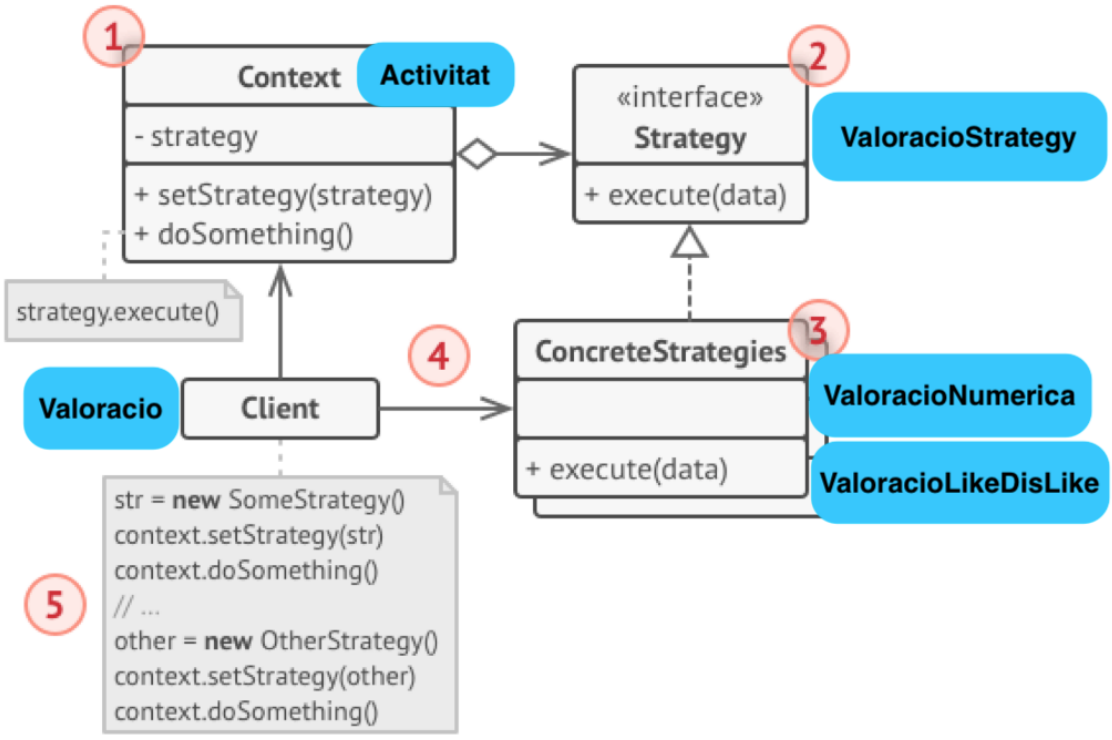
   El patró que hem fet servir ha estat l'**STRATEGY**. Bàsicament l'hem fet servir ja que com hem implementat un nou mètode de valoració (per *Likes*/*Dislikes*) ens hem adonat de que si no implementéssim el patró *d'Strategy*, tindríem classes molt similars que només varien en la forma de comportar-se (a més, per distingir el comportament entre els diferents tipus de valoració estaríem vulnerant el principi ***Open-Closed***). És per això que aplicar el mètode *Strategy* ens soluciona varies qüestions:

   - Ens permet aïllar els detalls de la implementació de la solució.
   - Ens permet utilitzar diferents estratègies i canviar-les en temps d'execució.
   - Ens trenca la vulnerabilitat del principi ***Open-Closed***.

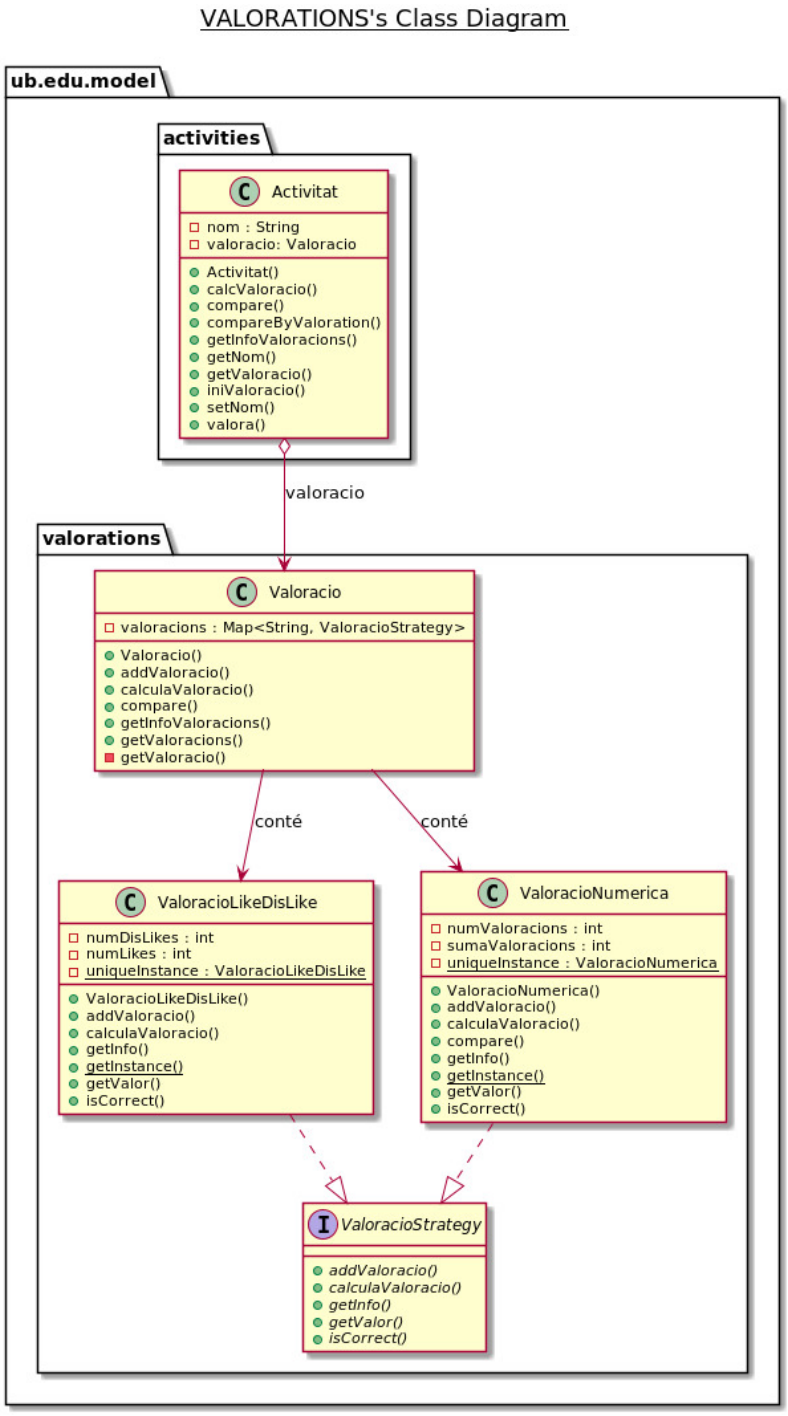
   Hem comprovat que un mateix problema (valorar una activitat) es pot solucionar amb diferents estratègies o algorismes. L'elecció d'una estratègia o un altra (***ValoracioNumerica*** / ***ValoracioLikeDisLike***) ve donada pel tipus de valoració que vulgui realitzar un soci o per com es vulguin llistar les activitats. Per tal de fer això utilitzem una interfície comuna per a poder encapsular tots els tipus d'algorismes. Utilitzem també herències per modelar els clients que utilitzen les diferents estratègies.

b. Posa els diagrames genèrics del patró utilitzat i identifica cadascuna de les classes del patró amb les classes que inclouràs en el teu codi

2. Patró a aplicar: Strategy



c. Dibuixa el diagrama de classes de la part on has aplicat el patró en el teu codi.



Finalment inclou aquí el repartiment de la feina que heu decidit per realitzar aquesta pràctica:

Estudiant 1 (Noah Márquez Vara):

- Part 1: *GRASP* i *SOLID*.
- Part 2: Comentari *Abstract Factory Method*
- Part 3: Patrons *Singleton, Strategy, Facade* i implementació de la nova funcionalitat.
- Realització de la memòria.

Estudiant 2 (Lluc Aresté Saló):

- Part 1: GRASP i SOLID
- Part 2: Comentari *Abstract Factory Method*
- Part 3: Patrons *Singleton, Strategy, Facade* i implementació de la nova funcionalitat.

Observacions finals: (si vols explicar algun altre comentari)

En la part d'implementar la nova funcionalitat hem considerat que seria correcte entendre com a *Like* una entrada de '1' i com a *Dislike* una entrada de '0'.

Els ***commits*** (aportacions) al *GitHub* no indiquen la feina desenvolupada per cadascun de nosaltres ja que en aquesta pràctica hem tingut certs problemes al fer els pull del git per tal de seguir el TDD i que cadascú implementés un apartat. Tot i així, el 90% de la pràctica l'hem fet conjuntament de manera presencial, participant tots dos en la implementació i resolució de les diferents parts de la pràctica.

Instruccions per al lliurament

Inclou una còpia d'aquest document amb les respostes de cada pregunta a la carpeta doc del teu projecte final.

El dia del lliurament es penjarà en el campus virtual un fitxer comprimit en **format ZIP** amb el nom dels dos membres del grup i el numero de lliurament com a nom de fitxer. Per exemple, A01-BartSimpsonLisaSimpsonL3.zip, on L3 indica que es el "lliurament 3". El fitxer ZIP inclourà: aquest document amb les respostes en format pdf i el projecte de IntelliJ final corresponent al projecte de la tasca del classroom de github del teu grup.