Pràctica 2: Grups A i B

Objectius:

L'objectiu principal d'aquesta pràctica és avaluar el codi desenvolupat a la pràctica 1 usant els principis disseny GRASP, SOLID i el patró arquitectònic per capes integrant l'arquitectura clàssica de Model-Vista-Controlador.
Com a objectiu secundari, s'aprendrà a utilitzar el patró DAO per a codificar la capar de recursos.

Enunciat de la pràctica 2:

Seguint amb les funcionalitats relatives a marcar les activitats com a preferides i valorar les activitats realitzades per cada Soci, es suposa que teniu la següent llista de Històries d'Usuari implementada:

- Històries UC1. Enregistrar-se
- Històries UC2. Login
- Històries UC3. Llistar excursions
- Històries UC4. Cercar Excursio
- **Històries UC9. Llistar d'Activitats i derivats com per exemple:**
  - **UC9.1 Llistar Activitats per Excursió**
  - **UC9.2 Llistar Activitats Favorites**
  - **UC9.3 Llistar Activitats Realitzades**
  - **UC9.4 Llistar Activitats per Valoració (no implementada)**
- **Històries UC10. Cercar Activitats i derivats**
- **Històries UC11. Afegir Activitat com a Favorita**
- **Històries UC12. Marcar Activitat coma a Realitzada**
- **Històries UC13. Valorar Activitat**
- **Històries UC14. Visualitzar Activitat**

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 dues parts:

1. Analitzar el codi sota la perspectiva de l'acoblament i la cohesió de les classes que has desenvolupat i implementat a la pràctica 1

2. Permetre la inicialització de les dades a partir de diferents recursos: d'unes classes que "simulen" una base de dades (DAO-MOCK) o a partir d'unes classes que guarden a disc (DAO-JSON)

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.

NOTA: Material de suport pel desenvolupament de la pràctica 2:

Si ets sents confortable amb la teva pràctica 1, segueix amb el teu codi i el teu projecte d'IntelliJ. En el cas que no l'hagis pogut lliurar a temps o saps que el teu lliurament està a mitges, replica el projecte solució a partir del següent enllaç:

https://classroom.github.com/a/l5Km3LCL

PART 1: Refactoring seguint els criteris GRASP

Segueix els següent passos i contesta els següents punts:

1. **Extreu el diagrama de classes del teu codi de la pràctica 1 utilitzant el plugin Sketch it! De IntelliJ. Completa el fitxer plantUML per a detallar totes les dependències entre classes. Inserta aquí la imatge del diagrama de classes del teu projecte:**

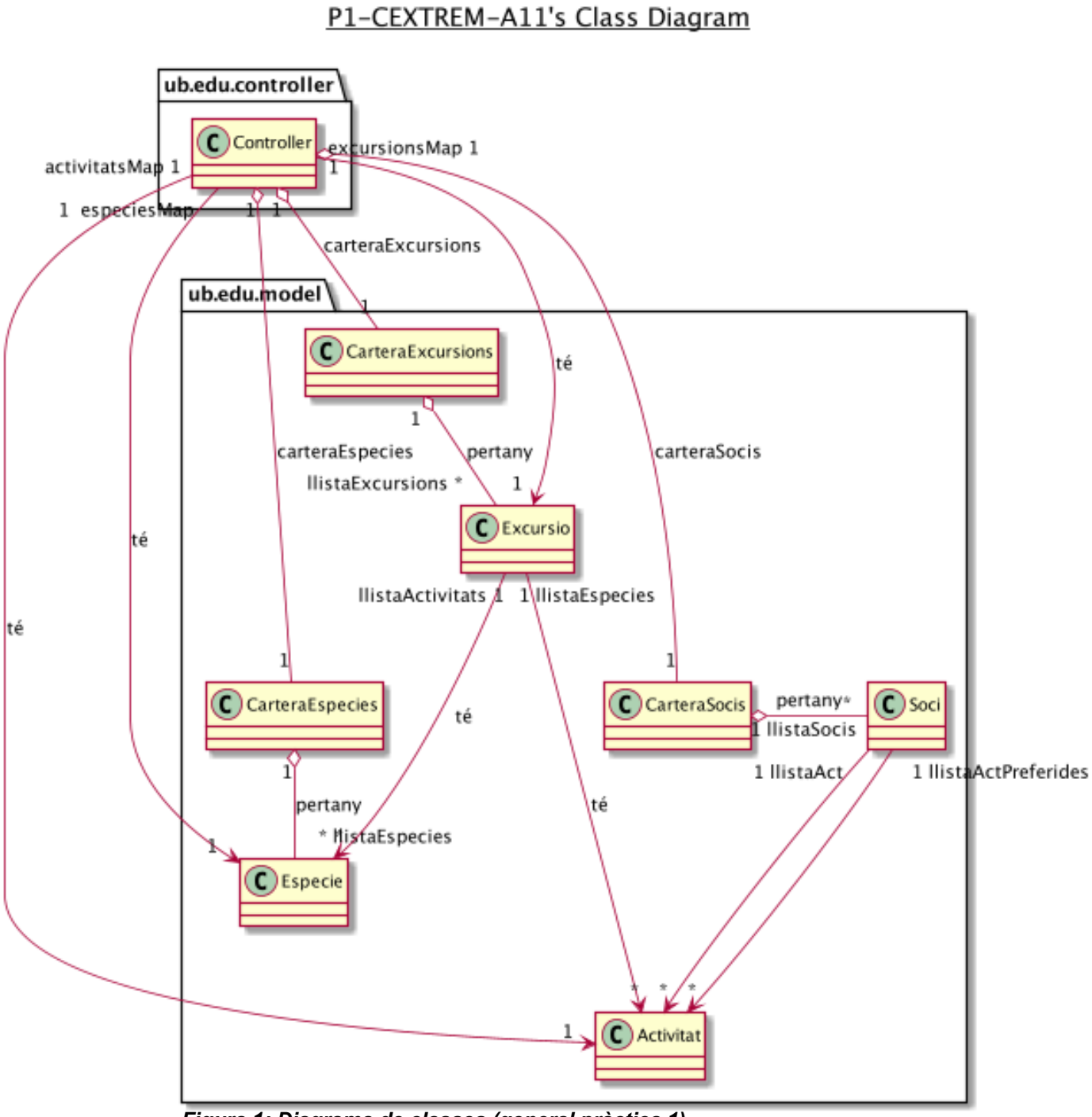

Figura 1: Diagrama de classes (general pràctica 1)

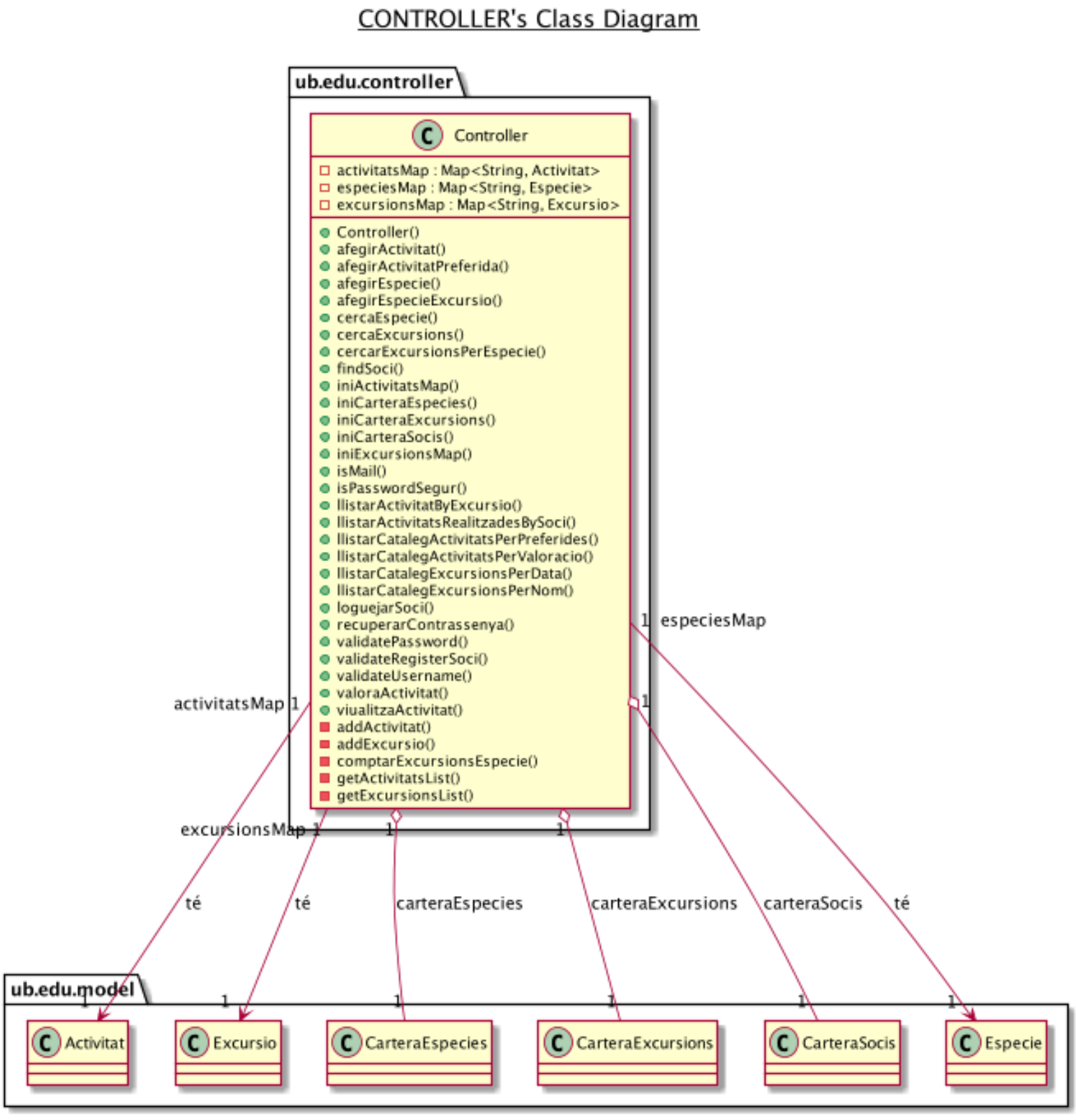

Figura 2: Diagrama de classes (controlador)

MODEL's Class Diagram

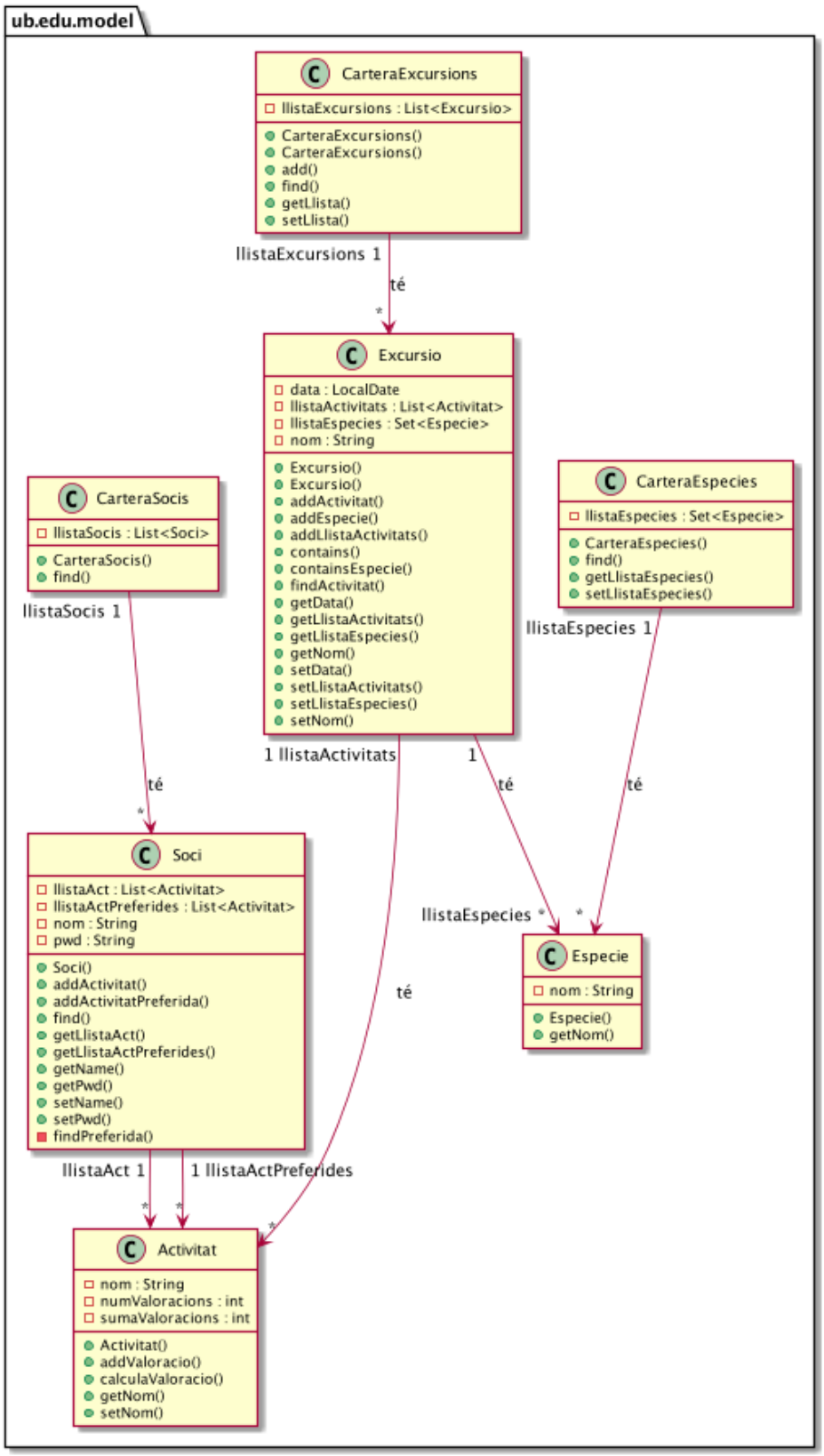

Figura 3: Diagrama de classes (model)

2. **Detecta males olors del teu codi:**
   a. **Identifica quines classes estan més acoblades amb d'altres**

Controller acoblada amb **CarteraSocis**, **CarteraExcursions**, **CarteraEspecies**. Ja que segurament no podríem entendre que fa la classe **Controlador** aïlladament si no fos per aquestes tres classes i això implica que també seria més difícil reutilitzar-la perquè requereix de la presència d'aquestes classes.

Soci acoblada amb **Activitat** ja que és difícil de reutilitzar per haver de requerir de la presència d'aquesta última classe per entendre plenament el seu sentit.

Excursio acoblada amb **Activitat** i **Especie** pel mateix motiu que la classe **Soci**, ja que si volguéssim reutilitzar la classe **Excursio** hauríem de requerir de les classes a les que està acoblada per entendre-la i per fer-la servir plenament.

Les classes **CarteraSocis**, **CarteraExcursions** i **CarteraEspecies** estan acoblades amb les classes **Soci**, **Excursio** i **Especie** respectivament. Totes elles es poden arribar a entendre aïlladament però en el cas de voler reutilitzar-les s'hauria de fer en presència de les classes a les que estan acoblades.

Tot i que la classe **Activitat** no està acoblada amb cap altre classe, en una possible implementació en què no féssim servir les valoracions hauríem de canviar la implementació de la classe ja que no es podria reutilitzar directament.

   b. **Identifica quines estan menys cohesionades.**

Podem observar com la classe **Excursio** té una cohesió molt **baixa** ja que disposa de molts mètodes que no estan molt relacionats funcionalment. No podríem afirmar que col·labora gaire amb altres classes, ja que només disposa de mètodes independents que tenen en compte atributs de la pròpia classe i que fan servir el mateix objecte. A més realitza molt treball ja que ha de controlar ella mateixa tant una llista d'activitats com una llista d'espècies, a l'hora que els atributs primaris de la classe (nom i data).

Igualment observem que la classe **Soci** té també una **baixa** cohesió ja que, pel mateix motiu que la classe anterior, disposa d'un nombre bastant alt de mètodes que no estan gaire relacionats funcionalment a més de realitzar massa feina (controla les activitats realitzades i les activitats preferides).

Per contra, la classe **Especie** té una **alta** cohesió ja que només té un mètode (*getNom*) i no realitza 'res' de treball, ja que només té un constructor i el mètode esmentat.

La classe **Activitat** disposa d'una **certa** cohesió ja que en la nostra implementació té dos mètodes (**addValoracio** i **calculaValoracio**) que poden arribar a estar relacionats

funcionalment, però estem donant la responsabilitat a la classe **Activitat** de calcular la seva mitja. Una millor implementació podria ser derivar el càlcul de la valoració mitja a un altre classe per tal de que **Activitat** col·laborés amb ella, alliberant-se de càrrega de treball.

Podríem dir que la classe **CarteraEspecies, CarteraSocis** i **CarteraExcursions** tenen una alta cohesió ja que tenen pocs mètodes (*getters & setters* + find), relacionats funcionalment i la càrrega de treball de les classes no és gens excessiva.

Segurament la classe menys cohesionada de totes és el **Controller** ja que té masses mètodes, no gaire relacionats funcionalment i realitza massa treball (realitza totes les funcionalitats de la nostra aplicació, per ella passa tot el flux del programa). A més, no col·labora amb altres classes per alliberar-se de feina a fer i de mètodes.

3. **Segueix la metodologia TDD per anar refactoritzant el codi pas a pas:**
    a. Dedica't a UN test d'acceptació per començar a veure com es comporta el codi internament en el teu diagrama de classes anterior.
    b. Intenta seguir les recomanacions GRASP per poder millorar el codi d'aquest test. Recorda que els patrons GRASP els pots trobar a les transparències de classe de teoria: Baix Acoblament i Alta Cohesió, Expert en Informació, Creador, Controlador, Fabricació Pura, Indirecció, Polimorfisme i Variacions Protegides. Els tens explicats també en el vídeo 18 – Patrons GRASP. No vol dir que hagis d'usar tots els patrons a cada test, sinó potser et serveixen de guia per poder modificar el codi.
    c. Refactoritza segons els criteris que pensis i prova el test via Concordion i tots els tests anteriors per a validar que segueixen funcionant. Potser que ara no vagi algun dels tests posteriors però fixa't només en els que vagis refactoritzant
    d. Es puja el github el test fet i validat amb el missatge que identifica el test correctament i el patró o patrons GRASP que has usat.
    e. Es procedeix a refactoritzar el següent test d'acceptació (anar al pas 1).

Així, per cadascuna de les funcionalitats demanades, haureu de fer **un add/commit/push a github per cada test d'acceptació** implementat amb el comentari adient del test i quin patró GRASP heu usat. Això vol dir, que en finalitzar l'entrega haureu de tenir, com a mínim tants commits/push a github com tests d'acceptació. Normalment, es procedeix a fer un commit/push a cada test d'acceptació per a no acumular molts canvis del projecte local en relació al remot. Al final del punt 3, llista aquí per cada tests d'acceptació el criteri GRASP que has fet servir i explica breument què has canviat en el teu codi inicial, Afegeix files a la taula si així ho necessites:

Test d'acceptació	Criteris GRASP	Breu explicació de les classes canviades
marcarPreferida/marcarRealitzada	Creador	**GestorActivitat** té ara la responsabilitat de la creació de les instàncies de la classe **Activitat** (contindrà un atribut estàtic amb la cartera d'activitats generals de l'enunciat).
	Creador	**GestorSoci** té ara la responsabilitat de la creació de les instàncies de la classe **Soci** (contindrà un atribut estàtic amb la cartera de socis generals de l'enunciat).
	Expert	Repartiment de **afegirActivitatPreferida** o **AfegirActivitatRealitzada** a **GestorActivitat** i també **CarteraActivitats** (perquè és l'experta en la informació de les activitats).
	Baix Acoblament	El **Controller** només coneix **GestorSoci** i no la classe **Soci**, **Activitat**, **CarteraActivitat**, **CarteraSocis**.
	Controlador	Ofereix la utilitat de **afegirActivitatPreferida** / **afegirActivitatRealitzada** a la Vista (o test)
valorarActivitat	Expert	La classe **GestorActivitat** (juntament amb **CarteraActivitats**) tenen la

		informació necessària per valorar una activitat. (en el cas que el soci no existeixi, el flux acaba a **GestorSoci**)
	Baix Acoblament	El **Controller** només coneix **GestorSoci** i no a la classe **Soci**, **Activitat**, **CarteraActivitat** o **CarteraSocis**.
	Alta Cohesió	La nova classe Valoració ens ha permés augmentar la cohesió **d'Activitat**, ja que ara no s'encarrega d'afegir valoracions ni de calcular la seva mitja.
	Controlador	Ofereix la utilitat de **valoraActivitat** a la Vista (o test)
	Creador	**Activitat** té la responsabilitat de crear una instància de la classe **Valoracio**, on emmagatzemarà totes les valoracions rebudes i podrà fer el càlcul de la mitja.
visualitzarActivitat	Expert	La classe **GestorActivitat** (juntament amb **CarteraActivitats**) tenen els recursos necessàris per mostrar la informació d'una activitat.
	Controlador	Ofereix la utilitat de **visualitzarActivitat** a la Vista (o test)
llistaActivitats	Expert	La classe **CarteraActivitat** és l'experta per tots els mètodes de llistar activitats (realitzades, valorades, preferides i per les

		activitats que hi ha en una excursió) ja que és la classe que disposa de la informació per realitzar els mètodes.
	Alta Cohesió	El fet de només haver de disposar de dos mètodes (a la classe **CarteraActivitats**) per a llistar totes les activitats, segons dos criteris (alfabèticament o per valoració), ens ha permès augmentar la cohesió respecte haver hagut de disposar d'un mètode per llistar per preferides, per realitzades i per les activitats que té una excursió.
	Controlador	Ofereix la utilitat de **llistaActivitats** a la Vista (o test)

4. **Detalla el diagrama final de classes que has obtingut al final de la refactorització de tots els tests d'acceptació. Si detectes encara alguna "mala olor" en el codi, explica-la aquí breument:**

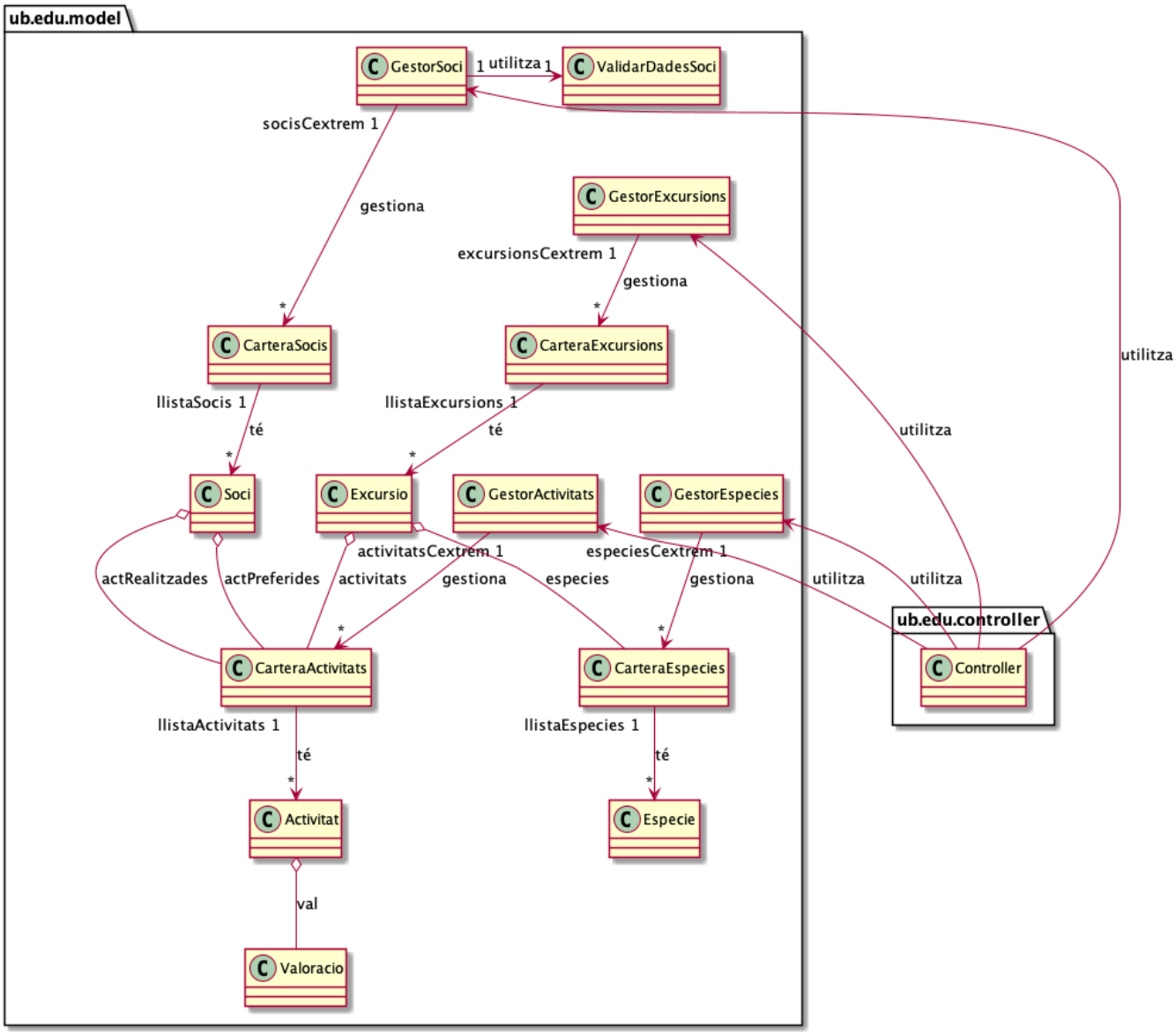

Figura 4: Diagrama de classes (general pràctica 2)

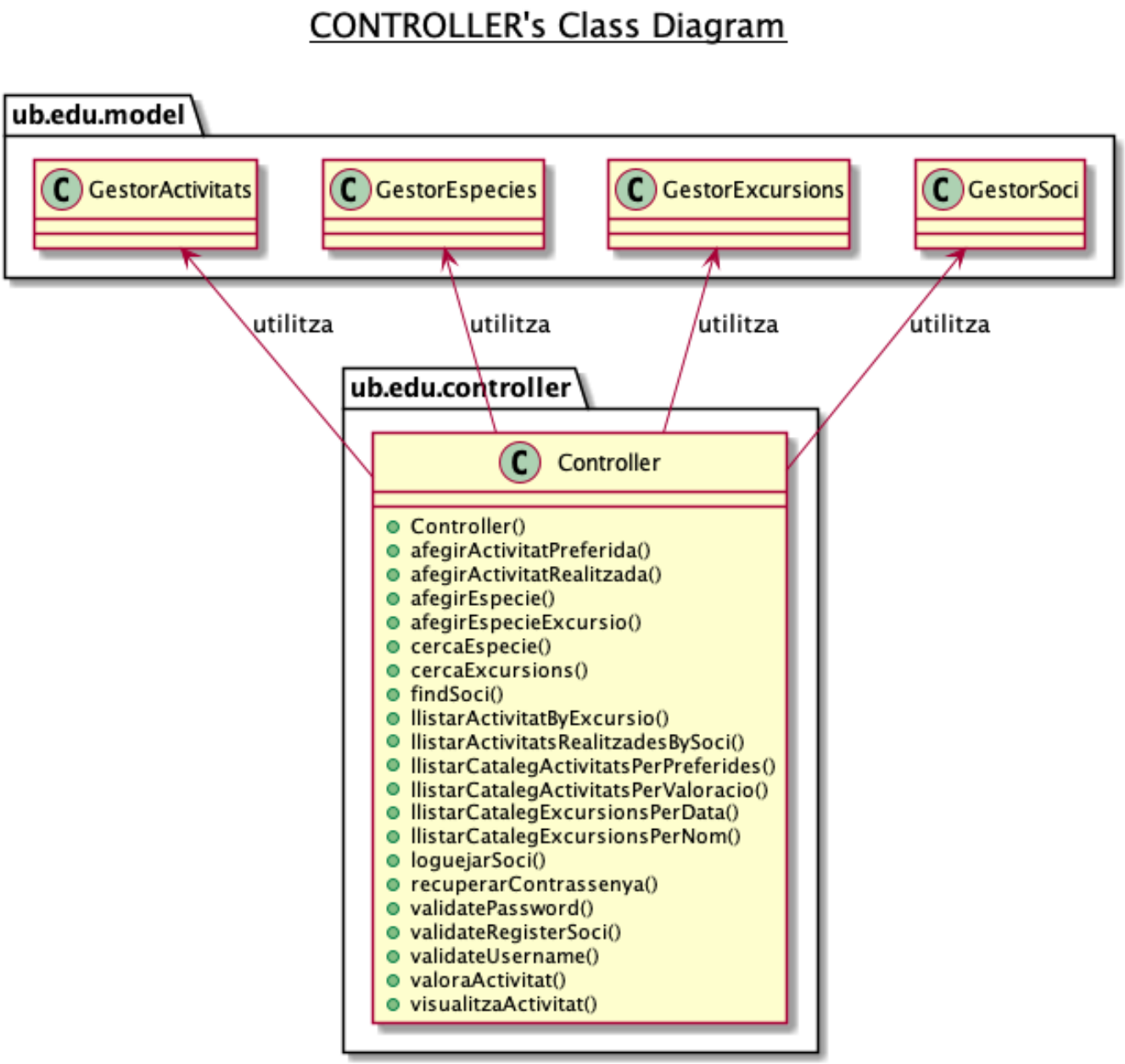

Figura 5: Diagrama de classes (controlador)

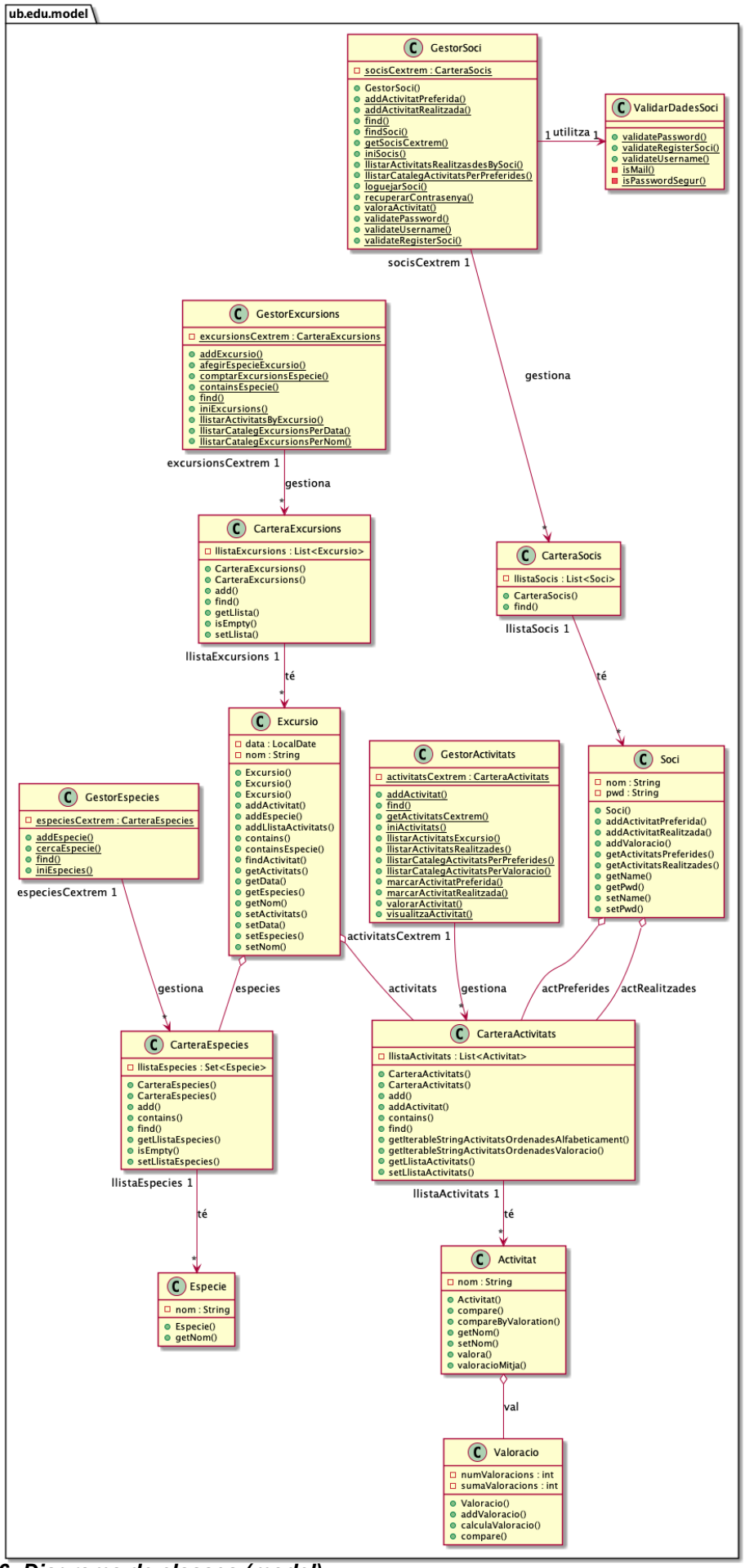

Figura 6: Diagrama de classes (model)

Baixa Cohesió: El **Controller** té baixa cohesió en servir moltes responsabilitats a la Vista. Abans el controlador era l'expert en realitzar totes les funcionalitats, en canvi ara no ho és en cap, tot i que totes les funcionalitats que es criden des de la vista (tests) passen pel controlador. Per tant aquí podríem pensar en realitzar una Façana (amb una Indirecció o Fabricació Pura).

PART 2: Codificant la capa de recursos

En aquesta part, aprendrem a utilitzar i codificar el patró DAO per inicialitzar les dades de l'aplicació usant de forma flexible recursos externs com poden ser fitxers o bases de dades. Per això, segueix els següent passos i contesta els punts que ho requereixin:

1. 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. La capa de recursos o persistència es podrà substituir per l'accés a una bases de dades compartides amb tota la classe amb dades sobre les excursions, activitats, socis, etc.. Quan s'inicií l'aplicació, s'inicialitzaran totes les dades de la capa de lògica de negoci necessàries per a l'aplicació.
   - 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.
   - En aquest projecte base s'inicialitzen les dades des d'un **MOCK** de la Base de Dades (un *mock* és un objecte que simula un altre objecte a efectes de test). En el projecte base que us proporcionem hi han dos Mocks, un per inicialitzar Socis i un altre per inicialitzar Excursions.
   - En el projecte base també es proporciona els mateixos tests de la pràctica 1 base. Trobareu una descripció més detallada en el Manual del Campus Virtual.

3. **En relació al codi de la capa de persistència (carpeta recursos i data service), quins patrons GRASP s'estan identificant?. Llista'ls a continuació, justificant breument la teva resposta (afegeix files a la taula si és necessari):**

Patró GRASP usat a la capa de persistència	Breu Justificació
Per exemple, Cohesió alta a les classes DAO-MOCK	Les classes DAO-MOCK internament només tenen la responsabilitat de gestionar les dades com si fossin una base de dades i cap altra

Creador de l'origen de les dades	La classe **FactoryMOCK** és la responsable de crear les que seran les classes que gestionaran les dades com una base de dades (**DAOEspecieMOCK**, **DAOSociMOCK**, **DAOExcursioMOCK**)
Expert de connexió amb les dades	La classe **DataService** és l'encarregada de fer de connexió entre la base de dades i la nostra aplicació, ja que té la informació necessària per dur a terme aquesta responsabilitat.
Baix acoblament a les classes DAO-MOCK	Les classes **DAO-MOCK** disposen d'un baix grau de connexió, coneixement i dependència respecte d'altres classes, ja que la seva única funció és actuar com una base de dades.
Polimorfisme entre la interfície DAO i les classes DAO-MOCK	Primerament passant per les diferents interfícies DAO (**DAOEspecie**, **DAOExcursio**, **DAOSoci**) per tenir un baix acomplament, fem polimorfisme per manegar diferents alternatives de comportament basades en el tipus de dades.

4. **Modifica aquest projecte base per inicialitzar especies en el seu DAO corresponent i no des dels tests com es fa fins ara (mira CercaExcursions a la carpeta de tests). Quines classes caldria que actualitzessis? Per això intenta resseguir el codi des del Controlador per veure com es fa la inicialització de les excursions, per exemple. Intenta modificar el codi per donar d'alta les Espècies. Explica aquí breument la teva solució**

   Les classes que caldria actualitzar serien les següents:

   - **FactoryMOCK**: afegir un mètode *public* de tipus ***DAOEspecie***, anomenat ***createDAOEspecie*** que faci un retorn d'un ***new DAOEspecieMOCK***.
   - **Controller**: afegir un atribut de tipus ***private Map<String, Especie> especiesMap***, afegir un mètode al constructor de **Controller** per tal d'inicialitzar la llista d'espècies (***iniLlistaEspecies***), i per últim implementar el darrer mètode esmentat de la mateixa manera en què s'inicialitzen les excursions.

- **DataService**: afegir un atribut de tipus ***private DAOEspecie*** anomenat ***daoEspecie*** per posteriorment inicialitzar-lo en el constructor de la classe mitjançant la interfície ***AbstractFactoryData*** que té un mètode anomenat ***createDAOEspecie***. Afegim també un mètode de tipus ***public List<Especie>*** anomenat ***getAllEspecies*** que cridarà al mètode ***getAll*** de l'atribut ***daoEspecie*** (retornant la llista amb totes les espècies).
- **DAOEspecieMOCK**: al constructor de la classe hem afegit tres espècies per comprovar el correcte funcionament, fent ús del mètode ***addEspecie***, que afegeix el nom de l'espècie i l'objecte espècie al ***HashMap*** de ***llistaEspecies***.
- **AbstractFactoryData**: hem afegit un mètode de tipus ***DAOEspecie*** a la interfície, anomenat ***createDAOEspecie***.

5. [OPT] Integra el codi que gestiona la capa de persistència a la teva pràctica resultat del pas 1. Vigila d'integrar poc a poc, mètodes del controlador des del projecte DAO al teu codi. **FET**
6. [OPT] Afegeix els DAOs corresponents a noves dades que necessites. **Com gestionaràs la càrrega d'activitats referents a una excursió? Raona la teva solució a continuació, justificant-la segons els patrons GRASP**.

   La forma en què gestionem la càrrega d'activitats referents a una excursió és tenint un atribut de tipus ***CarteraActivitats*** per a cada objecte de tipus ***Excursio***, de tal forma que en el moment que per exemple vulguem llistar les activitats d'una excursió, nomes haurem d'accedir a aquest atribut. Així obtenim una cohesió alta a la classe ***CarteraActivitats*** ja que només té la responsabilitat de gestionar una llista d'activitats i un baix acoblament ja que la classe ***CarteraActivitats*** es pot entendre aïlladament i li és igual les altres classes, ja que només té una única responsabilitat. Podríem dir que la classe ***CarteraActivitats*** és l'experta en gestionar la càrrega d'activitats referent a una excursió ja que és la qui disposa de la informació i mitjans per fer-ho.

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

Estudiant 1 (Noah Márquez Vara):

PART 1:

- Diagrama de classes
- Males olors codi
- Test marcarPreferida
- Test login
- Test cercaExcursions
- Test llistaExcursions
- Informe (parlat conjuntament)

PART 2:

- Exercici 3
- Exercici 4
- Exercici 5

Estudiant 2 (**Lluc Aresté Saló**):

PART 1:

- Diagrama de classes
- Males olors codi
- Test marcarRealitzada
- Test valorar Activitat
- Test visualitzar Activitat
- Test llista Activitats
- DCU actualitzat i Històries d'usuari Excel (a la carpeta doc)

PART :

- Exercici 3
- Exercici 5
- Exercici 6

Observacions finals: (si vols explicar algun altre comentari)

Finalment hem realitzat tota la pràctica 2 amb el codi de la nostra entrega corresponent a la pràctica 1.

Hem realitzat també els exercicis optatius de la part 2 d'aquesta pràctica (hem editat el codi sobre el projecte p2-cextrem-a11. Per tant, al ZIP de l'entrega només inclourem el projecte IntelliJ del nostre codi.

Instruccions per al lliurament

Tots els lliuraments es presenten junt amb una còpia d'aquest document amb les respostes incloses a cada pregunta.

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-BartSimpsonLisaSimpsonL2.zip, on L2 indica que es el "lliurament 2" i que pertanys a l'equip A01. 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.