

PRÀCTICA

Disseny i codificació

Uoc

Informació rellevant:

- Data límit de lliurament: 24 de juny.
- Pes a la nota final de Pràctiques: 100%.

Contingut

Informació docent	3
Prerequisits	3
Objectius	3
Resultats d'aprenentatge	3
Introducció	4
Metodologia en cascada o waterfall	4
Patró Model-Vista-Controlador (MVC)	6
Enunciat	7
Aspectes generals a tenir en compte	7
Exercici 1 - Disseny (5 punts)	8
Exercici 2 - Codificació (4.5 punts)	15
Exercici 3 - Vista (0.5 punts)	20
Corol·lari	22
Avaluació	24
Exercici 1 - Disseny (5 punts)	24
Exercici 2 - Codificació (4.5 punts)	24
Exercici 3 - Vista (0.5 punts)	25
Format i data de lliurament	26
Annex: Guia Dia	27
Instal·lació	27
Guia ràpida d'ús	27
Escol·lir elements UML per dibuixar	27
Elements bàsics de dibuix	28
Definir classes i mètodes abstractes	29
Definir atributs i mètodes estàtics	29
Definir classes, atributs i mètodes final	29
Definir una interfície	30
Definir una enumeració	30

Informació docent

Aquesta activitat pretén que posis en pràctica tots els conceptes relacionats amb el paradigma de la programació orientada a objectes vistos a l'assignatura. L'aplicació d'aquests conceptes es durà a terme amb el disseny d'un programa que solucioni un problema donat (és a dir, diagrama de classes UML) i la seva codificació en Java.

Prerequisits

Per fer aquesta Pràctica necessites:

- Tenir assimilats els conceptes dels apunts teòrics (és a dir, els 4 mòduls tractats en les PAC), incloent-hi aquells relacionats amb els diagrames de classes UML.
- Haver adquirit les competències pràctiques de les PAC. Per això et recomanem que miris les solucions que es van publicar a l'aula i les comparis amb les teves.
- Tenir assimilats els coneixements bàsics del llenguatge de programació Java treballats durant el semestre. Per això, et suggerim repassar aquells aspectes que consideris oportuns a la Guia de Java.

Objectius

Amb aquesta Pràctica l'equip docent de l'assignatura busca que:

- Sàpigues analitzar un problema donat i codificar una solució a partir d'un diagrama de classes UML i unes especificacions, seguint el paradigma de la programació orientada a objectes.
- Siguis capaç d'utilitzar un programari lliure per a la realització de diagrames de classes.
- T'enfrontis a un programa de mida mitjana basat en un patró d'arquitectura com és MVC (Model-Vista-Controlador).

Resultats d'aprenentatge

Amb aquesta Pràctica heu de demostrar que sou capaços de:

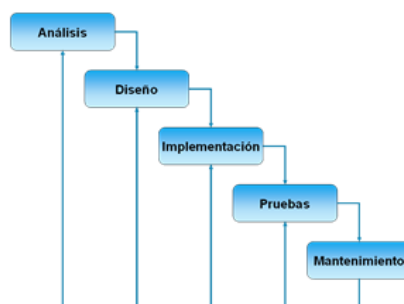
- Crear un diagrama de classes UML que exploti els conceptes i mecanismes de l'orientació a objectes per donar solució a un problema donat.
- Codificar un programa basat en un patró d'arquitectura com és MVC.
- Codificar en Java un diagrama de classes UML seguint el paradigma de programació orientada a objectes i el patró d'arquitectura MVC.
- Usar fitxers de test a JUnit per determinar que un programa és correcte.
- Usar amb certa soltesa un entorn de desenvolupament integrat (IDE) com IntelliJ.

Introducció

L'Equip Docent considera oportú que, arribats a aquest punt, relacionem aquesta assignatura amb conceptes propis de l'enginyeria del programari. Per això, a continuació explicarem la metodologia de desenvolupament anomenada “en cascada” (en anglès, *waterfall*) i el patró d'arquitectura programari “Model-Vista-Controlador (MVC)”. Creiem que aquesta informació, a més de contextualitzar aquesta Pràctica, pot ser interessant per a algú que està estudiant una titulació afí a les TIC.

Metodologia en cascada o *waterfall*

La metodologia clàssica per dissenyar i desenvolupar programari es coneix amb el nom de metodologia en cascada (o en anglès, *waterfall*). Tot i que ha estat reemplaçada per noves variants, és interessant conèixer la metodologia original. En la seva versió clàssica, aquesta metodologia està formada per 5 etapes seqüencials (veure següent figura).



L'etapa d'**anàlisi** de requeriments és reunir les necessitats del producte. El resultat d'aquesta etapa sol ser un document escrit per l'equip desenvolupador (encapçalat per la figura de l'analista) que descriu les necessitats que aquest equip ha entès que necessita el client. No sempre el client se sap expressar o és fàcil entendre què vol. Normalment aquest document el signa el client i és “contractual”.

Per la seva banda, l'etapa de **disseny** descriu com és l'estructura interna del producte (p.ex. quines classes fer servir, com es relacionen, etc.), patrons a utilitzar, la tecnologia a emprar, etc. El resultat d'aquesta etapa sol ser un conjunt de diagrames (p.ex. diagrames de classes UML, casos d'ús, diagrames de seqüència, etc.) acompanyat d'informació textual. Si ens decanem en aquesta etapa per fer un programa basat en programació orientada a objectes, serà també en aquesta fase quan fem servir el paradigma *bottom-up* per a la identificació dels objectes, de les classes i de la relació entre elles.

L'etapa d'**implementació** significa programació. El resultat és la integració de tot el codi generat pels programadors juntament amb la documentació associada.

Un cop acabat el producte, es passa a l'etapa de **proves**. S'hi generen diferents tipus de proves (p.ex. de test d'integració, de rendiment, etc.) per veure que el producte final fa allò

que s'espera que faci. Evidentment, durant l'etapa d'implementació també es fan proves a nivell local –tests unitaris (per exemple, a nivell d'un mètode, una classe, etc.)– per veure que aquesta part, de manera independent, funciona correctament.

L'última etapa, **manteniment**, comença quan el producte es dona per acabat. Encara que un producte estigui finalitzat, i per moltes proves que s'hagin fet, sempre apareixen errors (*bugs*) que cal anar solucionant a posteriori.

Si ens fixem a la figura, sempre es pot tornar enrere des d'una etapa. Per exemple, si ens manca informació a l'etapa de disseny, sempre podem tornar per un instant a l'etapa d'anàlisi per recollir més informació. L'ideal és no haver de tornar enrere.

En aquesta assignatura hem passat, en certa manera, per les quatre primeres fases. L'etapa d'anàlisi l'hem fet, doncs, des de l'Equip Docent. Nosaltres ens hem reunit amb el client i hem analitzat/documentat totes les seves necessitats (Pràctica 1). A partir d'aquestes necessitats, hem pres decisions de disseny. Per exemple, vam decidir que el programari es basaria en el paradigma de la programació orientada a objectes i que faríem servir el llenguatge de programació Java. Un cop decidits aquests aspectes clau, hem anat definint, a partir de la identificació d'objectes, les diferents classes (amb els seus atributs i mètodes, dades i comportaments) i les relacions entre elles a partir de les necessitats del client confirmades a l'etapa d'anàlisi i d'entitats reals que apareixen al problema (ie objectes). Per això hem fet servir un paradigma *bottom-up*. Com podeu imaginar, l'etapa de disseny és una de les més importants i determinants de la qualitat del programari, ja que s'hi realitza una descripció detallada del sistema a implementar. És important que aquesta descripció permeti mostrar l'estructura del programa de manera que la seva comprensió resulti senzilla. Per aquest motiu és freqüent que la documentació de la fase de disseny vagi acompanyada de diagrames UML, entre ells els diagrames de classes (Pràctica 1). Aquests diagrames a més de ser útils per a la implementació, també ho són a la fase de manteniment.

L'etapa d'implementació (o també coneguda com a desenvolupament o codificació) l'hem treballada durant tot el semestre i l'abordarem amb un èmfasi especial en aquesta segona pràctica.

Finalment, l'etapa de test/proves l'hem tractada durant el semestre amb els fitxers de test JUnit que es proporcionaven amb els enunciats de les PAC i amb alguna prova extra que hagi fet pel teu compte. Aquests fitxers ens permetien saber si les classes codificades es comportaven com esperàvem. En aquesta segona pràctica, també aprofundirem en aquesta fase de testeig.

Evidentment, la metodologia en cascada no és l'única que existeix, n'hi ha moltes més: per exemple, prototipat i les actualment conegudes com a metodologies àgils: eXtreme Programming, etc. En aquest punt podem dir que a les PAC hem seguit, en part, una metodologia TDD (*Test-Driven Development*), en la fase de disseny de la qual es defineixen els requisits que defineixen els test (te'ls hem proporcionat amb els enunciats) i són aquests

els que dirigeixen la fase d'implementació. Si vols saber més sobre metodologies per desenvolupar programari (on s'inclouen els patrons) i UML, t'animem a cursar assignatures d'Enginyeria del Programari. A aquestes metodologies de desenvolupament de programari se'ls ha d'afegir les estratègies o metodologies de gestió de projectes, com SCRUM, CANVAS, així com tècniques específiques per a la gestió dels projectes, p.ex. diagrames de Gantt i PERT, entre d'altres.

Patró Model-Vista-Controlador (MVC)

En aquesta Pràctica farem servir el patró d'arquitectura de programari anomenat MVC (Model-View-Controller, i.e. model, vista i controlador). El patró MVC és molt utilitzat en l'actualitat, especialment al món web. De fet, amb el temps han sorgit variants com MVP (P de *Presenter*) o MVVM (Model-Vista-VistaModel). En línies generals, MVC intenta separar tres elements clau d'un programa:

- **Model:** s'encarrega d'emmagatzemar i manipular les dades (i l'estat) del programa. En la majoria d'ocasions aquesta part recau sobre una base de dades i les classes que hi accedeixen. Així doncs, el model s'encarrega de realitzar les operacions CRUD (i.e. Create, Read, Update i Delete: crear, consultar, actualitzar i eliminar) sobre la informació del programa, així com de controlar els privilegis d'accés a aquestes dades. Una alternativa a la base de dades és l'ús de fitxers de text i/o binaris. El model també pot ser un conjunt de dades volàtils que desapareixen al tancar el programa.
- **Vista:** és el conjunt de “pantalles” que configura la interfície amb què interactua l'usuari. Cada pantalla o vista pot ser des d'una interfície per línia d'ordres fins a una interfície gràfica, diferenciant entre mòbil, tauleta, ordinador, etc. Cada vista sol tenir una part visual i una altra d'interactiva. Aquesta última s'encarrega de rebre els inputs/esdeveniments de l'usuari (p.ex. clic en un botó) i de comunicar-se amb el/s controlador/s del programa per demanar informació o per informar d'algun canvi realitzat per l'usuari. A més, segons la informació rebuda pel/s controlador/s, modifica la part visual en consonància.
- **Controlador:** és la part que controla la lògica del negoci. Fa d'intermediari entre la vista i el model. Per exemple, mitjançant una petició de l'usuari (p.ex. fer clic en un botó), la vista –a través de la seva part interactiva– demana al controlador que li doni el llistat de persones que hi ha emmagatzemades a la base de dades; el controlador sol·licita aquesta informació al model, el qual se la proporciona; el controlador envia la informació a la vista que s'encarrega de processar (i.e. part interactiva) i mostrar (i.e. part visual) la informació rebuda del controlador.

Gràcies al patró MVC es desacoblen les tres parts. Això permet que tenint el mateix model i el mateix controlador, la vista es pugui modificar sense veure's alterades les altres dues parts. El mateix, si canviem el model (per exemple, canviem de gestor de base de dades de MySQL a Oracle), el controlador i les vistes no haurien de veure's afectades. El mateix si

modifiquéssim el controlador. Així, amb l'ús del patró MVC es minimitza l'impacte de futurs canvis i es millora el manteniment del programa. Si vols saber-ne més, et recomanem veure el següent vídeo: <https://youtu.be/UU8AKk8Slqg>.

Enunciat

Aquesta Pràctica conté 3 exercicis. Aquesta es basa en el patró MVC. Concretament es demana:

- **Exercici 1 - Disseny del model (5 punts):** elaboració del diagrama de classes del model que doni resposta al problema plantejat.
- **Exercici 2 - Codificació del model i controlador (4.5 punts):** seguint el patró MVC, implementació en Java del model proposat en el diagrama de classes de l'Exercici 1 i dels `TODO` del controlador proporcionat seguint el paradigma de programació orientada a objectes.
- **Exercici 3 - Vista (0.5 punts):** millora d'una de les vistes proporcionades.

Per a cada exercici has de lliurar la teva solució en el format que s'especifica en el propi exercici i/o en l'apartat "Format i data de lliurament" d'aquest enunciat.

Aspectes generals a tenir en compte

En aquest apartat volem ressaltar algunes qüestions que considerem importants.



1. Per fer aquesta pràctica **hauràs de tenir en compte les especificacions que s'indiquin en aquest enunciat i en els tests. Recorda que els tests tenen prioritat en cas de contradicció.**

2. Abans de seguir llegint, **llegeix els criteris d'avaluació** d'aquesta Pràctica.

3. Quan tinguis problemes, rellegeix l'enunciat, busca en els apunts i a Internet. Si encara així no aconsegueixes resoldre'ls, **usa el fòrum de l'aula abans que el correu electrònic.** Això sí, **en el fòrum no pots compartir codi.**

4. **La realització de la Pràctica és individual.** Si es detecten indicis de plagi o l'ús d'eines d'intel·ligència artificial, l'Equip Docent es reserva el dret de contactar amb l'estudiant per aclarir la situació. Si finalment es ratifica la falta d'originalitat i autoria, l'assignatura quedarà suspesa amb un 0 i s'iniciaran els tràmits per obrir un expedient sancionador tant a l'estudiant que ha copiat com al que ha facilitat la informació.

5. **La data límit indicada en aquest enunciat és inajornable.** Pots fer diversos lliuraments durant el període de realització de la Pràctica. El/La PDC corregirà l'últim lliurament. **Passada la data límit no s'acceptaran lliuraments.**

Exercici 1 - Disseny (5 punts)

Abans de començar amb aquest exercici has de tenir en compte els següents aspectes:



1. En aquest exercici es demana el diagrama de classes del model. No existeix una solució única, així que no et preocupis.
2. Per realitzar aquest exercici has d'utilitzar l'editor de diagrames [Dia](#) o [Drawio](#). Per a Dia et donem una breu guia amb aquest enunciat (veure annex).
3. Com que a l'Exercici 2 codificaràs en Java, el diagrama de classes UML que facis ha de ser per a aquest llenguatge. Així doncs, si algun atribut/mètode utilitza alguna classe/interfície de l'API de Java, el seu nom ha d'aparèixer com si fos un tipus primitiu (no cal fer una relació binària, agregació, etc.). En canvi, si la classe/interfície de Java forma part d'una relació d'herència, només és necessari que la caixa contingui el seu nom.
4. L'enunciat d'aquest exercici pot contenir informació que no sigui necessària per al disseny del programa a nivell de diagrama de classes. Quan parlem amb un client, aquest ens dona informació que pot abordar-se en diferents fases del producte, per exemple, disseny, codificació, etc. Cal ser capaç de discernir quan una informació és útil i quan no per a cada fase.
5. Has de lliurar un fitxer editable de tipus `.dia` o `.drawio` amb el diagrama de classes UML que conté la solució que proposes per donar resposta a les necessitats i especificacions del problema plantejat.
6. Un fitxer `.png` que representi en format imatge el diagrama de classes UML que conté el fitxer editable anterior.

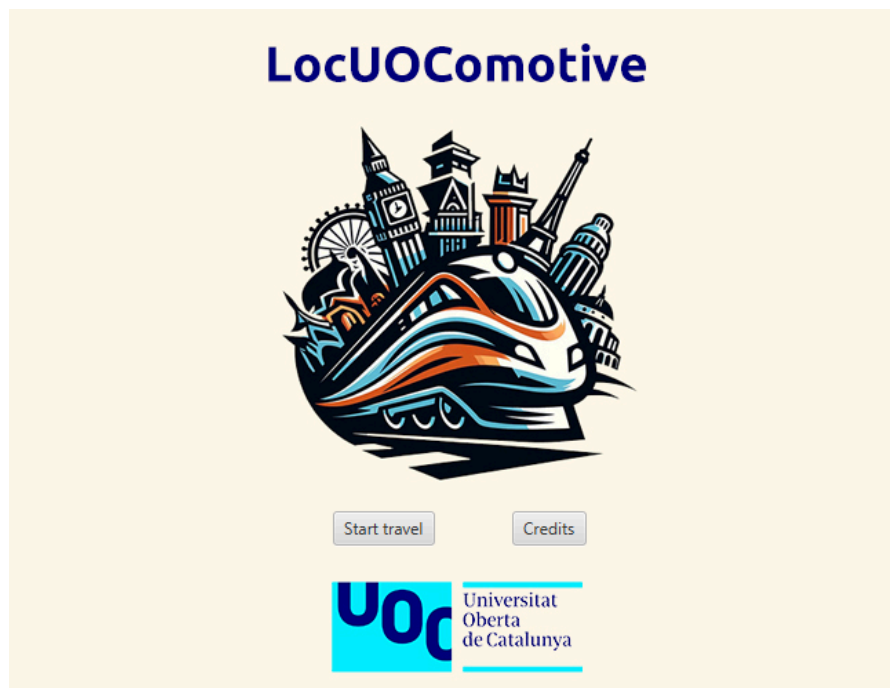
A continuació, descrivim el problema per al qual has de donar una solució en forma de diagrama de classes UML:

Aquest semestre volem realitzar una aplicació de gestió de viatges de tren. Aquesta aplicació ens permetrà viatjar per diferents ciutats d'Europa i tenir un registre de tots els viatges realitzats.

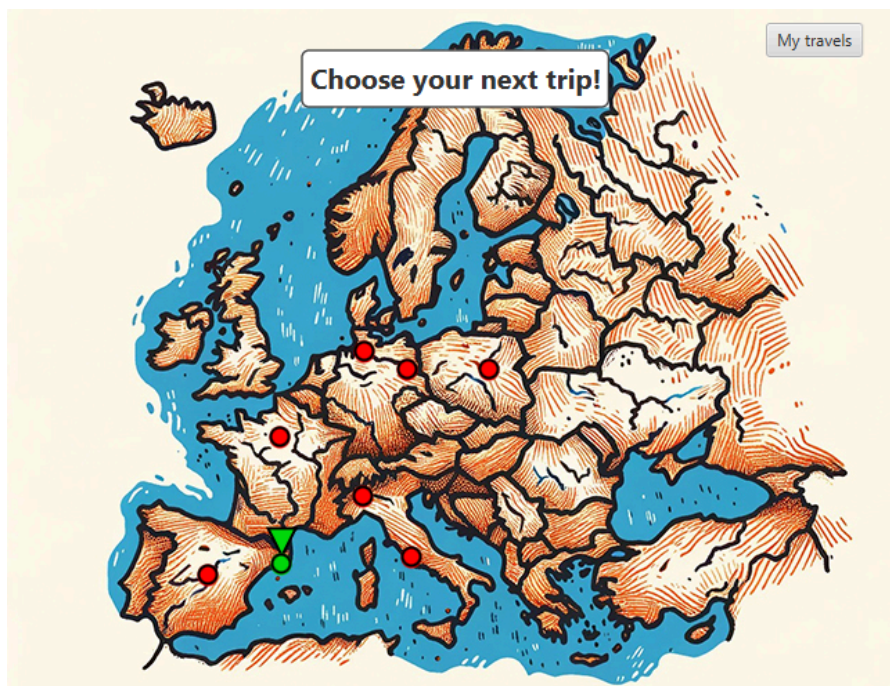
El prototip que es desenvoluparà només conté 8 estacions de tren, un total de 8 rutes i 8 models de tren diferents. No obstant això, podeu afegir nous elements a l'aplicació si així ho desitgeu per fer més proves o simplement per gust al final del desenvolupament.

A continuació, us mostrem un parell de captures de pantalla per poder tenir una idea inicial de l'aplicació que dissenyarem i codificarem.

Pantalla d'inicio



Pantalla de selecció del proper destí del viatge en tren



Les imatges utilitzades per desenvolupar aquesta aplicació han estat generades amb intel·ligència artificial.

Com es pot observar a la segona imatge, els punts corresponen a les estacions de tren que hi ha repartides per Europa. Aquests punts són botons pulsables que permeten seleccionar la següent destinació a la qual viatjarem. Així mateix, el punt verd (exclamació) correspon a la ubicació actual de l'usuari que farà la compra del viatge.

Per poder representar aquesta situació, per a cada estació és necessari emmagatzemar el seu identificador, que ha de ser únic, el nom de l'estació, la ciutat on es troba, l'any d'obertura, el tipus d'estació que és segons la ubicació de les seves andanes (per sobre del nivell del terra, sota el terra o elevades) i les seves coordenades x i y al mapa de l'aplicació. A més, com que l'aplicació pot mostrar la imatge de l'estació, és necessari emmagatzemar el nom del fitxer JPG de la imatge de la mateixa. Tota la informació que carrega l'aplicació sobre les estacions la podeu trobar al fitxer `stations.txt` al directori `/resources/data/`.



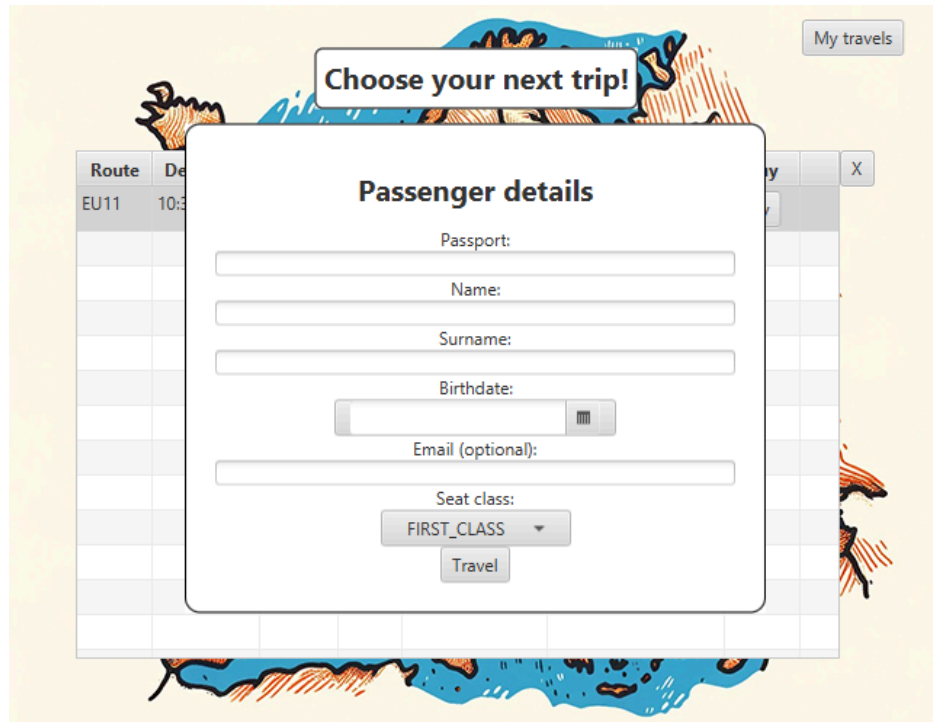
Pista: Considereu la sobrescriptura d'alguns mètodes com `toString` durant el disseny i la codificació de les classes si considereu que us poden ser útils per simplificar la implementació.

Com és lògic, l'aplicació també guarda informació sobre els trens que hi ha disponibles. En aquest cas, la informació rellevant és l'identificador únic del tren, el seu model i el conjunt de vagons que té. Igual que passa amb els trens actuals de llarga distància, un tren pot tenir un o més vagons dedicats al servei de restauració. Per tant, els vagons de restauració (també anomenats cafeteria) no poden disposar de seients a la venda. En canvi, per als vagons de passatgers és important emmagatzemar tots els seients i tenir coneixement de quin passatger va a cada seient.

A més, no tots els vagons de passatgers són de la mateixa categoria, ja que en total hi ha 3 classes de vagons, oferint d'aquesta manera un viatge en primera, segona i tercera classe (`FIRST_CLASS`, `SECOND_CLASS` i `THIRD_CLASS`). Els de primera classe són els que menys seients poden tenir per vagó, menys de 20; els de segona, a partir de 20 i menys de 50; i els de tercera classe, que poden tenir 50 o més seients per vagó. Tots els vagons, incloent-hi els de restauració han d'estar identificats amb la lletra C seguida d'un número que va incrementant a mesura que s'afegeixen vagons al tren. Per exemple, el primer vagó tindria com a número "C1", el segon "C2" i així successivament. Tota la informació que carrega l'aplicació sobre els trens i els seus vagons la podeu trobar al fitxer `trains.txt` al directori `/resources/data/`.

Per acabar amb els elements principals de l'aplicació, també és necessari emmagatzemar la informació referent als passatgers. Per comprendre millor la informació que s'ha d'emmagatzemar, es proporcionarà una captura de pantalla del formulari que conté l'aplicació.

Pantalla per introduir les dades del passatger i el tipus de seient.



Choose your next trip!

Passenger details

Passport:

Name:

Surname:

Birthdate:

Email (optional):

Seat class:

Com es pot observar, per a cada passatger s'ha d'emmagatzemar el seu passaport, el seu nom, els seus cognoms, la seva data de naixement i, opcionalment, un correu electrònic de contacte. Els camps obligatoris (tots menys l'email) han de ser validats i no poden ser mai null, i en el cas de les cadenes de caràcters aquestes no poden contenir un text buit. En el cas de l'email (únic camp opcional), en cas de ser informat, ha de tenir un format d'email correcte. És a dir, pot ser una cadena de caràcters buida si no s'informa, però si conté algun valor, aquest sempre ha de ser un email vàlid. En aquest cas, es defineix un email vàlid sota les següents condicions:

- Abans de l'@ ha d'haver-hi com a mínim un caràcter. Els caràcters permesos són lletres de l'alfabet anglès (majúscules i minúscules, sense caràcters com la ç o la ñ), números, punts (.), guions baixos (_) o guions (-).
- Ha de contenir sempre un símbol @.
- Després de l'@ ha d'haver-hi un text format com a mínim per un caràcter. Els caràcters permesos són lletres de l'alfabet anglès (només en minúscules, sense caràcters com la ç o la ñ), números, punts (.) i guions (-). El text que va després de l'@ representa el nom del domini (o subdomini) del correu electrònic.
- Després d'aquest últim conjunt ha de contenir una extensió de domini, que sempre ha de començar amb un punt (.) seguit de 2 o 3 caràcters en minúscula.

Seguint amb el cas d'ús de compra d'un bitllet, una vegada l'usuari ha seleccionat l'horari per viatjar a una estació (és a dir, fa clic al botó `Buy`), apareix el formulari que s'ha mostrat anteriorment, en el qual l'usuari introdueix les seves dades i prem el botó `Travel`. Si totes les dades són correctes, es considera que l'usuari compra correctament el bitllet i, a més, es simula el viatge a l'estació desitjada.

És important tenir en compte que els bitllets de tren han de ser emmagatzemats per poder ser consultats a l'històric i, a més, cada passatger ha de poder consultar els seus bitllets adquirits. De cada bitllet, ens interessa saber el propietari d'aquest bitllet, la referència del seient seleccionat, el seu preu i la informació del viatge (l'hora i l'estació de sortida, l'hora i l'estació d'arribada, i el seient en què es va viatjar –el qual s'identifica pel número del vagó seguit d'un guió i el número del seient dins del vagó).

A més, l'assignació de seient es produeix durant el procés de compra, triant sempre el primer seient disponible començant pel primer vagó i recorrent tots els seients per número de seient. Això sí, és important que el seient assignat sigui del mateix tipus que el passatger va especificar durant el formulari. És a dir, si l'usuari va especificar que vol un seient en primera classe, s'haurà d'assignar el primer seient que compleixi amb aquesta condició.

Una vegada el passatger adquireix el bitllet, automàticament es simula el viatge i es trobarà a l'estació de destinació. Per tant, el mapa ha d'actualitzar l'estació en què es troba, deixant de color vermell i amb un botó pulsable l'estació d'origen i marcant amb l'exclamació verda l'estació de destinació, ja que és l'estació en què es troba en aquell moment el passatger. Així mateix, si el passatger prem el botó de la part superior dreta amb el text `My travels`, ha d'aparèixer la informació del bitllet com s'observa a la següent captura de pantalla.

[illegible]

Per simplificar el funcionament de l'aplicació, es considerarà que tots els passatgers baixen del tren quan ho fa l'usuari de l'aplicació. Per tant, una vegada s'arribi a l'estació de destinació, tots els seients del tren han de quedar disponibles de nou per a un altre viatge.

Finalment, com haureu observat a la pantalla del formulari, l'aplicació sol·licita les dades del passatger en cada compra. No obstant això, internament no s'han d'emmagatzemar passatgers duplicant el seu passaport. En cas que s'introdueixi un passaport que ja estigui registrat a l'aplicació, ha de prendre la seva referència i actualitzar les dades guardades per aquelles noves que hagi rebut durant aquest procés de compra.



A l'hora de crear el vostre diagrama de classes, podeu afegir tantes classes i associacions com vulgueu sempre que considereu que permeten representar de millor manera l'escenari descrit. A més, també podeu crear les enumeracions i interfícies que necessiteu.

Exercici 2 - Codificació (4.5 punts)

Aquest exercici es divideix en 2 parts:

- Codificar el diagrama de classes per al model que has proposat a l'Exercici 1.
- Codificar els `TODO` del controlador que et proporcionem en el fitxer `.zip`.

Abans de començar amb aquest exercici has de tenir en compte els següents aspectes:



1. Per fer aquest exercici **hauràs de tenir en compte les especificacions que s'indiquin en aquest enunciat i en els tests. Els tests tenen prioritat en cas de contradicció.**

2. Pots utilitzar qualsevol classe, interfície i enumeració que et proporcionï l'API de Java. No obstant això, **no pots afegir dependències** (és a dir, llibreries de tercers) que no s'indiquin en aquest enunciat.

3. En el controlador no podeu modificar les signatures dels mètodes proporcionats amb l'enunciat. No obstant això, podeu codificar mètodes auxiliars, però **de cap manera es poden modificar els que venen amb l'enunciat.**

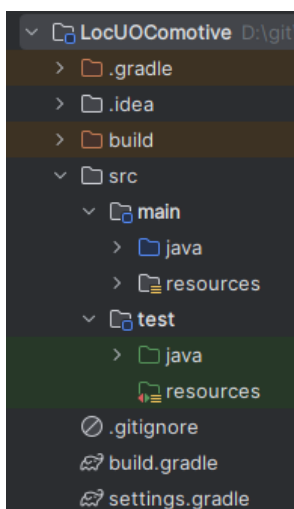
4. **Els tests proporcionats no han de modificar-se.** Així mateix, si es detecten trampes, per exemple, *hardcodejar* codi per superar els tests, la nota final serà 0.

Entorn

Per a aquesta pràctica utilitza el següent entorn:

- JDK ≥ 21 .
- IntelliJ Community.
- Gradle, qui descarregarà les dependències necessàries per al projecte.

Estructura general del projecte



Si obres el `.zip` que se't proporciona amb aquest enunciat, trobaràs el projecte `LocUOComotive`. Si l'obres a IntelliJ, veuràs l'estructura que es mostra en la següent imatge. D'aquesta estructura cal destacar:

src: és el projecte en si, que segueix l'estructura de directoris pròpia de Gradle (i Maven).

A `src/main/java` veuràs tres paquets anomenats `model` (que pots dividir en els subpaquets que consideris), `view` i `controller`. Ho hem organitzat així perquè, com hem comentat, utilitzarem el patró MVC.

A `src/main/resources` trobaràs els estils i pantalles que s'utilitzen en la vista gràfica del joc, així com els fitxers de configuració de l'aplicació.

Per la seva part, `src/test/main` conté els fitxers de test JUnit. Així mateix, a `src/test/resources` trobaràs fitxers de configuració de l'aplicació que són utilitzats per testejar el programa.

build.gradle: conté tota la configuració necessària de Gradle. En aquest fitxer hem definit tasques específiques per a aquesta Pràctica amb la finalitat d'ajudar-te durant la seva realització.

Model

Dins del *package* `edu.uoc.locuocomotive.model` codifica el diagrama de classes que has proposat a l'Exercici 1.

Controlador

El controlador és qui maneja la lògica del negoci. En aquest cas, la lògica de l'aplicació. És a dir, el controlador és el responsable de decidir què fer amb la petició que ha realitzat l'usuari des de la vista. El més habitual és que el controlador faci una petició al model.

Al *package* `controller` del projecte veuràs una classe anomenada `LocUOComotiveController`. Aquesta és la classe controladora de l'aplicació (un programa pot tenir diverses classes controladores).



Des dels mètodes del controlador amb el comentari `//TODO` hauràs d'utilitzar els elements definits en el model. Les signatures dels mètodes proporcionats amb l'enunciat no poden ser modificades. Podeu codificar funcions auxiliars addicionals, però **de cap manera es poden modificar els que venen amb l'enunciat**.

Igualment, hi ha mètodes que ja us donem completament codificats i que no heu de modificar, per exemple, `loadStations`.

Per a aquesta classe volem donar unes indicacions addicionals per a alguns mètodes que has de codificar. Abans, haureu notat que el controlador no té cap atribut i molts mètodes només reben i retornen tipus de dades del llenguatge Java com `String`, és a dir, no manegen instàncies del model com estacions o altres. Per tant, no només heu d'implementar els mètodes, sinó que heu de declarar tots els atributs necessaris per al correcte funcionament de l'aplicació.

LocUOComotiveController (constructor)

El constructor del controlador ha d'inicialitzar tots els atributs necessaris per al correcte funcionament de l'aplicació. Concretament, l'aplicació necessita emmagatzemar les estacions, les rutes, els trens i tots els bitllets adquirits. Per a totes aquestes dades, es

recomana utilitzar el tipus de dada `ArrayList`. A més, també ha de poder emmagatzemar tots els passatgers que s'hagin registrat a l'aplicació. Per a aquest cas, es recomana utilitzar una estructura de tipus `HashMap`.

Finalment, el constructor també ha de carregar les dades dels tres arxius mencionats anteriorment (`trains.txt`, `stations.txt` i `routes.txt`, en aquest ordre). A més, ha d'assignar com a estació actual (en la qual es troba el passatger usuari de l'aplicació) la primera estació que hagi carregat de l'arxiu `stations.txt`, que en el fitxer que us donem correspon a la de Barcelona.



Pista: reviseu els mètodes ja implementats amb l'enunciat per simplificar la codificació d'aquesta funcionalitat.

addStation

Aquest mètode ha de registrar l'estació a l'aplicació amb les dades que rep per paràmetre.

addRoute

Aquest mètode ha d'afegir una ruta a partir de la informació rebuda per paràmetres. Per a cada ruta, cal afegir tots els horaris per a les estacions per les quals passa. És a dir, les rutes registrades han de tenir accés a la referència de l'horari per on passen per cada estació. Quan s'invoca aquest mètode, es pressuposa que el tren que fa la ruta així com les estacions per les quals passa ja han estat afegits a l'aplicació prèviament.

Els horaris rebuts tenen el següent format:

```
stationId|hh:mm, hh:mm, ...]
```

Per tant, haureu d'utilitzar el mètode `split` per poder separar els elements com s'ha fet en altres mètodes que podeu prendre de referència.

addTrain

Ha d'afegir un tren a l'aplicació amb els valors rebuts per paràmetre. És a dir, no només ha de crear un tren, sinó que també ha de preparar tots els vagons i els seients de cada un d'ells.

getStationsInfo

Ha de retornar una `List` de Java de tipus `String` que contingui tota la informació de totes les estacions, en el següent ordre:

```
id|name|city|image|x|y
```

getSeatTypes

Ha de retornar un `array` de Java de tipus `String` que contingui els noms dels tipus de seients que s'hagin definit prèviament en el model. Es recorda que els tipus de seients poden ser de primera, segona o tercera classe.

getRoutesByStation

Rep l'identificador d'una estació i ha de retornar una `List` amb totes les rutes amb tots els horaris que té aquesta estació. Aquesta `List` ha d'estar ordenada segons l'hora de sortida del tren. Cada element de la `List` serà un `String` amb les dades següents respectant el format que es mostra a continuació:

```
departureTime|arrivalTime|routeId|ticketCost|departureStationId|
arrivalStationId|departureStationName|arrivalStationName
```

Així mateix, una de les informacions que ha de retornar és el cost del bitllet (element marcat en vermell). Per calcular-lo, s'aplica una fórmula on es multipliquen el nombre d'hores completes de viatge per una constant (30€/hora completa de viatge). Per exemple, si un viatge de Barcelona a París té una durada de 3 hores i mitja, el cost serà de 90€.

addPassenger

Registra un passatger a l'aplicació amb les dades rebudes per paràmetre. A més, en cas de trobar un passaport ja registrat (és a dir, mateix passaport), no ha de crear cap nou registre, sinó que ha d'actualitzar les dades del passatger amb les que ha rebut en aquest últim procés de compra. En cas que sorgeixi qualsevol error, s'ha de llançar una excepció.

createTicket

És el mètode que s'encarrega de crear un nou bitllet a partir de les dades rebudes. Per a això, haurà de trobar el primer seient disponible en el tren d'aquesta ruta i assignar el passatger. A més, és el mètode encarregat d'actualitzar l'estació en la qual es troba el passatger, viatjant a l'estació destinació i buidant el tren d'altres passatgers. En cas que sorgeixi qualsevol error, s'ha de llançar una excepció.

buyTicket

És el mètode que s'encarrega de gestionar tot el cas d'ús complet de la compra d'un bitllet. Per tant, ha de registrar el passatger a l'aplicació en cas que no hi estigui (o actualitzar les seves dades si ja existeix el seu passaport) i crear el bitllet amb les mateixes especificacions que s'han especificat per al mètode anterior. En cas que sorgeixi qualsevol error, s'ha de llançar una excepció. El missatge de l'excepció es mostrarà per la pantalla com un error.

getAllTickets

Ha de retornar una `List` de Java de tipus `String` amb la informació de cadascun dels bitllets registrats a l'aplicació. Cada element de la `List` (és a dir, un bitllet), serà un `String` amb el següent format i informació:

```
routeId|departureTime|departureStationName|  
arrivalTime|arrivalStationName|carNumber-seatNumber|ticketCost
```

getPassengerInfo

Aquest mètode rep un passaport i retorna la informació del passatger que coincideix amb aquest passaport com una `String` amb el següent format i informació:

```
passport|name|surname|birthday|email
```

En cas de no trobar el passatger, ha de retornar una `String` buida.

getTrainInfo

Aquest mètode rep un identificador de tren i retorna la informació del tren que coincideix amb aquest identificador com una `String` amb el següent format i informació:

```
trainId|model|numberOfCars
```

En cas de no trobar cap tren, ha de retornar una `String` buida.

getPassengerTickets

Aquest mètode rep un passaport i retorna una `List` de Java de tipus `String` amb la informació de cadascun dels bitllets que té el passatger amb el passaport rebut. Cada element de la `List` (és a dir, un bitllet), serà un `String` amb el següent format i informació:

```
routeId|departureTime|departureStationName|  
arrivalTime|arrivalStationName|carNumber-seatNumber|ticketCost
```

getRouteDepaturesInfo

Aquest mètode rep l'identificador d'una ruta i retorna una `List` de Java de tipus `String` amb la informació de cada estació incloent els seus horaris. Per a cada estació, és a dir, per a cada element de la llista, serà un `String` amb el següent format i informació:

```
stationId|[hh:mm, hh:mm, ...]
```

On la primera hora correspon al primer horari i, la segona, al segon horari.

getCurrentStationId

Aquest mètode retorna l'identificador de l'estació en la qual es troba l'usuari de l'aplicació. És a dir, retorna l'identificador de l'estació que es troba de color verd. La funció assumeix que l'usuari sempre està en una estació.

Exercici 3 - Vista (0.5 punts)

Les vistes són les "pantalles" amb les quals interactua l'usuari. En aquest cas, tenim una manera d'interactuar, és a dir, l'aplicació en mode gràfic. Amb el projecte ja et donem les vistes/pantalles del programa "fetes". Diem "fetes" perquè **et demanem que afegeixis noves columnes a la taula que conté els viatges que ja s'han fet. A més, a la pantalla credits.fxml, has d'afegir el teu nom i correu de la UOC.**

Pots modificar el contingut d'aquesta taula des del fitxer `PlayViewController` del paquet `edu.uoc.locuocomotive.view`. Veuràs que en la teva versió no hi són totes les columnes. **Hauràs d'afegir les que falten per deixar la taula tal com es veu en la següent captura de pantalla:**

[illegible]



Per fer aquesta part de l'enunciat et recomanem llegir l'apartat 5.2 de la Guia de Java obviat les referències a Eclipse. Com veuràs, es suggereix utilitzar el programa Scene Builder, el qual permet crear i modificar interfícies gràfiques de manera WYSIWYG. Trobaràs Scene Builder en el següent enllaç: <https://gluonhq.com/products/scene-builder/#download>. Si vols vincular Scene Builder amb IntelliJ (no és obligatori, però sí pràctic), ves a `File` → `Settings...` Després a `Languages & Frameworks`. Dins d'aquesta opció, escull la opció `JavaFX` i a la dreta indica on es troba l'arxiu executable de Scene Builder. A partir d'aquí, depenent de la versió d'IntelliJ, podràs fer clic dret a IntelliJ sobre un fitxer `.fxml` i dir-li que l'obri amb Scene Builder. Igualment, quan s'obre un fitxer `.fxml` a IntelliJ, aquest mostra dues pestanyes, una amb el codi FXML i una altra pestanya "Scene Builder" que integra Scene Builder dins de l'IDE.

Corol·lari

Si estàs llegint això, és que ja has acabat la Pràctica. Felicitats!! Arribats a aquest punt, segurament t'estiguis preguntant: *com m'ho faig per a passar-li el programa a algú que no tingui ni IntelliJ ni JDK instal·lats?* Bona pregunta. La resposta és que has de crear un fitxer executable, concretament, un JAR (Java ARchive). Un `.jar` és un tipus de fitxer –en veritat, un `.zip` amb l'extensió canviada– que permet, entre altres coses, executar aplicacions escrites en Java. Gràcies als `.jar`, qualsevol persona que tingui instal·lat JRE (*Java Runtime Environment*) el podrà executar como si d'un fitxer executable es tractés. Normalment, els ordinadors tenen JRE instal·lat.

Per crear un fitxer `.jar` per a una aplicació JavaFX cal tenir present que la classe principal (i.e. aquella que té el `main`) no pot heretar d'`Application`. Si ho fa, el `.jar` no s'executarà correctament. És per això que la solució més senzilla és crear una nova classe que cridi al `main` de la classe que hereta d'`Application`. Si mires el fitxer `build.gradle`, veuràs que dins de la configuració del plugin `application` utilitza com a `main`, el que té `LocUOComotive`, mentre que la tasca `jar` invoca el `main` de la classe `Main`. Així mateix, pel fet que JavaFX no pertany al core de JDK des de la versió 11, hem d'afegir els mòduls que el programa necessita, altrament, l'execució del `.jar` fallarà. Per indicar els mòduls hem de fer el projecte modular, que no és res més que afegir el fitxer `module-info.java` al projecte. Si t'hi fixes, t'hem facilitat aquest fitxer a `src/main/java`. A l'apartat 4.3 de la Guia de Java donem una pinzellada molt breu al tema dels mòduls introduïts per JDK 9.

Per crear un fitxer `.jar` que s'executi en una màquina que tingui instal·lada JRE, has de descomentar la tasca `jar` que trobaràs dins de `build.gradle`. Aquesta tasca està configurada per crear un *fat jar*, és a dir, un fitxer `.jar` que, a més de les classes del nostre programa, conté també totes les classes de totes les llibreries de què depèn. Així doncs, és un fitxer més gran (d'aquí l'ús de l'adjectiu *fat*) del que seria un `.jar` generat de manera normal. Un cop descomentada la tasca i actualitzades les tasques Gradle (recorda donar-li al botó refrescar que apareix al fitxer `build.gradle`), només has de fer doble clic a la tasca `jar` i es crearà el fitxer `.jar` dins d'una carpeta anomenada `build`. Més concretament, és dins de `build/libs`. Simplement copia el fitxer `LocUOComotive-1.0-SNAPSHOT.jar` (conté tot: `.class` i recursos) i executa'l on vulguis (assegura't que a l'ordinador que utilitzis estigui, com a mínim, la versió 21 de JRE). Pots executar-ho fent doble clic o usant l'ordre `java -jar LocUOComotive-1.0-SNAPSHOT.jar` en un terminal.

Potser estàs pensant: *què succeeix si a l'ordinador on es vol executar el .jar no hi ha ni JDK ni JRE?* Doncs, o bé l'instal·les, o bé usas `jlink`. El que fa `jlink` és empaquetar el `.jar` juntament amb una versió *ad hoc* de JRE. Per fer-ho necessita que el projecte Java estigui modularitzat, doncs, segons els mòduls que s'indiquin en el fitxer `module-info.java`, el JRE *ad hoc* que creï serà més gran o més petit. Per a usar `jlink`

has de comentar, a `build.gradle`, la tasca `jar` que genera el *fat jar*. A continuació, descomentar la tasca `jlink` que trobaràs a `build.gradle`. Després, només has de fer doble clic a la tasca de Gradle anomenada `jlink` que trobaràs dins del grup `build`. El resultat es crearà a la carpeta `build/image`. Per executar l'aplicació has d'anar a `image/bin` i executar el fitxer `LocUOComotive`, no sense abans copiar el directori `levels` que hi ha a `resources` dins del directori `bin` (sincerament, no sabem per què no funciona deixant-ho a `resources`). De vegades hi ha problemes perquè l'aplicació generada amb `jlink` llegeixi correctament els fitxers afegits a `resources`, així que si no us funciona, no us frustreu.

Cal destacar que `jlink` és una ordre pròpia de JDK i, per tant, es pot executar de sde línia d'ordres sense necessitat d'utilitzar Gradle (i el plugin corresponent): <https://www.devdungeon.com/content/how-create-java-runtime-images-jlink>.

/ si volem un instal·lador? Doncs a partir de JDK 16 està disponible `jpackage`. Llegeix més sobre `jar`, `jlink` i `jpackage` a: <https://dev.to/cherrychain/javafx-jlink-and-jpackage-h9>.

De totes maneres, avui en dia s'usen aplicacions com Docker per a distribuir programes.

Avaluació

Aquesta Pràctica s'avalua de la següent manera:

Exercici 1 - Disseny (5 punts)

S'avaluarà la qualitat de la proposta així com l'ús correcte de l'estàndard UML per a la creació de diagrames de classes. Així mateix, la no presentació del fitxer `.png` exigirà suposarà la pèrdua de 0.5 punts.

Exercici 2 - Codificació (4.5 punts)

Aquest exercici s'avaluarà mitjançant la superació dels tests proporcionats.

Tipus de test	Pes	Comentaris
4 basic	2 pts.	<p>Aquests tests comproven que els mètodes bàsics són funcionalment correctes. Per provar-los fes:</p> <pre>Gradle → verification → testBasic</pre> <p>La nota es calcularà a partir de la següent fórmula:</p> $(\#test_basic_passats / \#test_basic) * 2$
3 advanced	1.5 pts.	<p>Aquests tests comproven que els mètodes avançats són funcionalment correctes. Per provar-los fes:</p> <pre>Gradle → verification → testAdvanced</pre> <p>La nota es calcularà a partir de la següent fórmula:</p> $(\#test_advanced_passats / \#test_advanced) * 1.5$
1 special	1 pts.	<p>Aquest test comprova que el mètode més difícil de tots funciona correctament. Per provar-lo fes:</p> <pre>Gradle → verification → testSpecial</pre>



`Gradle → verification → testAll` executa tots els tests.

S'avaluarà la qualitat del codi lliurat observant qüestions com ara:

- Ús de les convencions i bones pràctiques del llenguatge Java.
- Qualitat dels algorismes.
- Llegibilitat/Claredat.
- Comentaris Javadoc per a classes, interfícies, enumeracions, atributs i mètodes que formen part del model. **No és necessari generar la documentació.**
- Comentaris en punts crítics o de difícil comprensió per a un tercer.

L'estudiant pot rebre una penalització de fins a 1 punt a causa de la mala qualitat del seu codi.

Exercici 3 - Vista (0.5 punts)

En aquest exercici s'avaluaran els següents ítems:

Número	Puntuació
Afegir nom i login als crèdits	0.10 punts
Afegir columnes a la taula de viatges	0.15 punts
Visualització correcta de la informació a les columnes afegides	0.25 punts

Format i data de lliurament

Has de lliurar un fitxer *.zip, el nom del qual ha de seguir aquest patró: `loginUOC_PRAC.zip`. Per exemple: `dgarciaso_PRAC.zip`. Aquest fitxer comprimit ha d'incloure els següents elements:

- Un directori anomenat `ex1` a l'arrel del fitxer *.zip que contingui el fitxer editable `Dia/Drawio loginUOC_PRAC.dia` (o `loginUOC_PRAC.drawio`) així com el fitxer `loginUOC_PRAC.png` generat, sent `loginUOC` el teu login UOC.
- El projecte `LocUOComotive` completat seguint les peticions i especificacions dels enunciats dels Exercicis 2 i 3.

L'últim dia per lliurar aquesta Pràctica és el **24 de junio de 2024** abans de les 23:59. Qualsevol Pràctica lliurada més tard serà considerada com no presentada.

Annex: Guia Dia

Dia és un programari lliure per fer diferents tipus de diagrames, entre ells, diagrames de classes. Sabem que hi ha altres programes (ArgoUML, MagicDraw, Rational Rose, etc.), però Dia és molt senzill i a més és gratuït. És important que us acostumeu a utilitzar programari lliure ja que a les empreses moltes vegades no es poden permetre pagar les llicències que suposen alguns programes.

Instal·lació

El programa Dia podràs descarregar-lo a la següent pàgina web:
<http://dia-installer.de/download/index.html>.



Si utilitzes MacOS i, més concretament la versió Yosemite, has de seguir les indicacions que s'expliquen en un dels següents enllaços perquè el programa s'executi correctament:

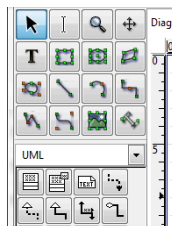
- <https://paletosdelaelectronica.wordpress.com/2015/02/23/dia-solucion-error-yosemite/>
- <http://navkirats.blogspot.com/2014/10/dia-diagram-mac-osx-yosemite-fix-i-use.html>

Guia ràpida d'ús

Encara que Dia és un programa molt senzill (i creiem que intuïtiu) d'utilitzar, comentarem com fer algunes coses:

Escollir elements UML per dibuixar

Dia permet dibuixar diferents tipus de diagrames. Per això, hem d'indicar-li que volem utilitzar elements UML. Per fer-ho, anem al panell esquerre del programa i al desplegable que apareix escollim UML. En aquell precís moment, just a sota del desplegable, apareixeran els elements que podem dibuixar a l'hora de crear el nostre diagrama de classes.



Elements bàsics de dibuix



- Clica en aquest botó i després clica al llenç per crear una caixa, per exemple, per crear una classe. Fes doble clic a la caixa perquè et mostri una finestra d'edició. A la pestanya "Atributs" pots crear i modificar atributs, i a la pestanya "Operacions" pots crear i modificar mètodes.



- Fes clic en aquest botó i després uneix dues caixes (d'origen a destinació; destinació = punta de fletxa) per crear una relació de dependència. Si fas doble clic a la línia, només t'has de preocupar d'escriure a l'apartat "Nom" per posar, per exemple, "throws".



- Fes clic en aquest botó i després uneix dues caixes (d'origen a destinació; origen = rombe) per crear una associació d'agregació o composició. Si fas doble clic a la línia, veuràs que apareix una finestra amb dues columnes: "Side A" (rombe) i "Side B" (sense rombe). Aquí pots escriure el rol (nom de l'atribut), la multiplicitat (p.ex. *), la visibilitat del rol (p.ex. si indiques "Privat" s'afegirà "-" davant del rol) i si vols mostrar la punta de la fletxa (important per a relacions unidireccionals).



- Clica aquest botó i després uneix dues caixes per crear una associació binària. Si fas doble clic a la línia, veuràs que apareix la mateixa finestra que amb el botó anterior. De fet, pots fer servir qualsevol dels 2 botons per crear una associació i escollir el tipus mitjançant l'opció "Tipus" de la finestra. L'opció "Res" correspon a una associació binària.



- Clica aquest botó i després uneix dues caixes (de superclasse a subclasse) per crear una relació d'herència.



- Clica aquest botó i després uneix dues caixes (d'interfície a classe) per crear una relació d'implementació.

Definir classes i mètodes abstractes

- Per definir una **classe abstracta** (i.e. nom en cursiva) has de fer doble clic a la classe, a la pestanya "Classe" marcar l'opció "Abstracta" i després a la pestanya "Estil" posar el valor `Bold-Oblique` al desplegable en què interseccionen la fila "Classe abstracta" i la segona columna de "Font".
- Per definir un **mètode abstracte** (i.e. nom en cursiva) has de fer doble clic a la classe, triar la pestanya "Operacions", escollir el mètode que vols que sigui abstracte i en el desplegable anomenat "Tipus d'herència", escollir l'opció "Abstracta". A continuació, perquè els mètodes abstractes de la classe es vegin correctament, has d'anar a "Estil" i posar el valor `Oblique` en el desplegable en què interseccionen la fila "Abstracta" i la segona columna de "Font".

Definir atributs i mètodes estàtics

- Per definir un **atribut static** (i.e. nom subratllat) has de fer doble clic a la classe, anar a la pestanya "Atributs", escollir l'atribut que vols fer estàtic i abaix a l'esquerra marcar l'opció "Vista de classe". Si és el darrer atribut del llistat, hauràs d'afegir un atribut buit amb visibilitat "Implementació" perquè es vegi clarament la línia que subratlla l'atribut `static`.
- Per definir un **mètode static** (i.e. nom subratllat) has de fer doble clic a la classe, anar a la pestanya "Operacions", escollir el mètode que vols fer estàtic i just a sota de "Tipus d'herència" marcar l'opció "Vista de classe". Si és l'últim mètode de la llista, hauràs d'afegir un mètode buit amb visibilitat "Implementació" perquè es vegi clarament la línia que subratlla el mètode `static`.

Definir classes, atributs i mètodes `final`

- Per definir una **classe final** pots escriure com a part de el nom de la classe el text `{leaf}` (p.ex. `Person {leaf}`). També pots escriure `final` a l'apartat "Estereotip" de la pestanya "Classe".
- Per definir un **atribut final** has d'escriure el nom d'aquest atribut en UPPERCASE amb els espais substituïts per `_`.
- Per definir un **mètode final** cal, a la pestanya "Operacions", escollir el mètode que vols fer `final` i just a sota de "Tipus d'herència" marcar l'opció "Consulta". Això farà que s'afegeixi el text `const` al costat del mètode.

Definir una interfície

- Una interfície es defineix igual que una classe, però escrivint `"interface"` a l'apartat Estereotip de la pestanya Classe.
- Si la interfície no tindrà atributs, pots amagar aquesta part de la caixa desmarcant l'opció "Atributs visibles" de la pestanya "Classe".
- Recorda que els mètodes que no són `default` a la interfície són abstractes.

Definir una enumeració

- Una enumeració es defineix igual que una classe però escrivint `"enumeration"` a l'apartat Estereotip de la pestanya Classe.
- Si l'enumeració no tindrà mètodes, pots amagar aquesta part de la caixa desmarcant l'opció "Operacions visibles" de la pestanya "Classe".
- Els valors de l'enumeració són atributs públics, `final` i `static` sense tipus ni valor per defecte. Això és així implícitament, per la qual cosa en aquest cas només hem d'escriure el nom dels valors en majúscula (són com constants, i.e. atributs `final`), i obviar el subratllat (per indicar que és `static`) i el modificador d'accés públic (es pot posar el valor "Públic" o "Implementació", que significa "per defecte o res").
- Sabem que les enumeracions són molt potents a Java, però a UML són molt limitades. Així doncs, podem afegir atributs al mateix apartat dels valors i, a la part d'operacions, podem posar constructors i mètodes. Tanmateix, no podem indicar quins valors agafa un valor de l'enumeració per als atributs declarats. Com que al final UML no és quelcom sagrat, podem, si volem, posar els diferents valors com a "valor per defecte" de l'atribut, però això ha de ser consensuat amb l'equip de treball ja que no és estàndard.