Documentació Projecte de Programació B9

Index

1.FLUX DE FUNCIONAMENT DE L'APLICACIÓ

Executar JAR: java -jar programa.jar

Comencem pel punt d'entrada que és la primera pantalla, hi hauran 5 inputs per pujar els fitxers CSV amb les dades de la simulació (llocs, camins, vehicles, conductors i peticions). 2 inputs més per penjar un fitxer amb format JSON, un per posar tots els objectes d'una simulació guardada i així poder-la carregar i un altre on podràs posar un JSON amb estadístiques d'una simulació. Llavors tindrem dos inputs que seran per posar la hora inicial i final de la simulació en format HH:mm.

Finalment tindrem varies opcions:

Iniciar simulació: Cal que els 5 fitxers CSV estiguin carregats i cal posar una hora d'inici i final, et portarà a la pantalla de la simulació.

Carregar simulació: Cal pujar el fitxer de simulació.json i carregarà la simulació.

Optimitzar simulació vehicles redundants: A partir d'un fitxer json d'una simulació, donarà unes estadístiques de quins vehicles son redundants.

Optimitzar simulació punts de càrrega redundants: A partir d'un fitxer json d'una simulació, donarà unes estadístiques de quins punts de càrrega son redundants.

Visualització d'estadístiques: Cal pujar el fitxer d'estadistiques.json i et mostrarà varis gràfics segons les estadístiques.

Com llegim els CSV?

El Main crida el mètode inicialitzar, el qual rep els fitxers com a paràmetres i carrega les dades de cada fitxer amb l'ajuda de la classe LectorCSV, on hi ha un mètode per cada fitxer que s'hagi de carregar. Un cop els hem llegit tots, creem el simulador i mostrem la pantalla del mapa.

A la pantalla del mapa hi veiem els llocs i els camins entre ells, un botó a sota per iniciar la simulació, una llegenda de cada símbol del mapa, adalt a l'esquerra un botó per afegir peticions aleatòries, adalt a la dreta la hora actual de la simulació, i al mig a la dreta un requadre on anirà sortint la informació dinàmica de la simulació.

·Quan iniciem la simulació, s'intenten assignar totes les peticions als conductors voraços i planificador:

- **Assignar a voraços**: En aquesta funció és on crec que tenim un problema del primer principi SOLID(Single responsability), ja que fa coses que hauria de fer la classe del conductor voraç. Però el que fa bàsicament és recorre les peticions i si aquesta està pendent, recorre per cada conductor, calcula per cada un el temps i la distància fins a l'origen de la petició i guarda el que estigui més a prop (tot això tinguent en compte que el conductor estigui lliure i el seu vehicle tingui bateria). Un cop ha trobat el conductor més proper, crida el mètode de planificar ruta del

conductor, i amb aquest ruta es crea un esdeveniment de IniciRuta.

- **Assignar a planificadors:**

Després de veure l'error que vam cometre de Single responsability, aquest mètode si que fa el que li toca, recórre els conductors un per un i crida el mètode de planificar la ruta. Per tant és el conductor que decideix si pot o no fer les peticions. Si la ruta que em retorna no és null, s'afegeix un esdeveniment de IniciRuta.

Un cop fet això, al mètode iniciar es crea un Timer el qual cada segon va llançant i executant events. Això s'anira fent mentre hi hagi events i la hora actual no superi a la hora final. Una vegada un conductor acaba una ruta, s'afegeix un esdeveniment de FiRuta, i allà es quan tornem a intentar assignar una petició.

Quan s'acaben de servir totes les peticions, s'acaba el temps o ningun conductor pot servir cap més petició, la simulació acaba. Es para el Timer i es mostra un pop-up amb les estadístiques de la simulació com: peticions servides, peticions no servides, temps total d'espera, temps màxim d'espera, ocupació total dels vehicles, temps de viatges…

2.Analisis del codi i compliment dels principis SOLID

A l'hora de desenvolupar aquest projecte, hem buscat construir una arquitectura clara, modular i fàcilment mantenible, aplicant principis fonamentals de la programació orientada a objectes. Ens hem basat especialment en els **principis SOLID**, que ens han servit com a guia per estructurar les classes i responsabilitats del sistema de manera neta i coherent.

1. Separació clara de responsabilitats

Hem estructurat el projecte en diverses classes amb funcions molt específiques. Les quatre principals són: Simulador, Conductor, Planificador i Mapa.

- Simulador és la classe central que coordina l'execució de la simulació, orquestrant la resta de components.

- Conductor encapsula el comportament dels conductors dins del sistema, incloent la seva lògica de moviment i decisió.

- Planificador s'encarrega de decidir les rutes i assignacions dins del sistema, separant la planificació estratègica de l'execució directa.

- Mapa representa l'entorn de la simulació i gestiona la distribució espacial dels conductors i les rutes.

Aquesta divisió ens ha permès garantir el **principi de responsabilitat única (SRP)**, fent que cada classe tingui una sola raó de canvi i que sigui més fàcil de provar, entendre i mantenir.

2. Codi extensible i preparat per a futurs canvis

Hem intentat que les classes estiguin **obertes a l'extensió però tancades a la modificació (OCP)**. Això vol dir que, si volem afegir noves funcionalitats (com nous tipus de conductors, estratègies de planificació o escenaris de simulació), ho podrem fer mitjançant noves classes o mètodes sense haver de modificar la lògica ja implementada. Aquesta flexibilitat ens permet fer créixer el projecte de manera ordenada i segura.

3. Comportament coherent i reutilitzable

Ens hem assegurat que el comportament de cada component sigui previsible i coherent, seguint el **principi de substitució de Liskov (LSP)**. Per exemple, qualsevol instància de Conductor pot ser tractada pel Simulador o el Mapa de manera uniforme, sense requerir canvis en el comportament d'aquests components. Això facilita la reutilització i integració de nous elements sense haver de modificar les estructures existents.

4. Classes amb interfícies públiques netes

Hem dissenyat les classes amb interfícies públiques clares i concises, evitant exposar atributs interns innecessaris. Encara que no hem implementat interfícies formals en Java, hem aplicat el **principi de segregació d'interfícies (ISP)** mantenint mètodes públics ben definits i acotats a la seva responsabilitat concreta. Això assegura una millor encapsulació i redueix l'acoblament entre components.

5. Dependències controlades i baixa dependència entre classes

Hem procurat que les classes tinguin **poca dependència directa entre elles**, utilitzant paràmetres simples (com col·leccions o objectes bàsics) per comunicar-se, seguint el **principi d'inversió de dependències (DIP)**. El Simulador pot coordinar les accions sense dependre de detalls interns del Planificador o dels Conductors, i el Mapa proporciona la infraestructura sense conèixer la lògica de simulació. Tot i això, hi ha marge per millorar aquest aspecte si es desitgés un desacoblament més profund mitjançant l'ús d'abstraccions o interfícies.

6. Bones pràctiques generals que hem seguit

- **Encapsulació correcta**: hem mantingut els atributs com a privats i els hem exposat només quan ha estat estrictament necessari, utilitzant getters i setters adequats.

- **Codi llegible i estructurat**: la nomenclatura triada per a mètodes i atributs és clara i significativa, i la lògica interna s'ha estructurat en mètodes curts, amb una única responsabilitat.

- **Ús adequat de col·leccions**: les estructures de dades s'han seleccionat d'acord amb els requisits de rendiment i organització del sistema, prioritzant la claredat i eficiència.

- **Separació entre dades i lògica**: hem evitat barrejar les dades amb la lògica operativa. Per exemple, el comportament dels conductors no depèn del format intern del Mapa, i la lògica de planificació està separada de l'execució directa per part dels conductors.

Conclusió

Amb aquest projecte hem volgut demostrar un enfocament rigorós i professional a la programació orientada a objectes. Aplicant els principis SOLID, hem aconseguit un disseny modular, flexible i fàcil de mantenir. La claredat en l'estructura del codi i la separació de responsabilitats ens permeten adaptar-lo a nous requeriments i escenaris, fent-lo escalable i preparat per a la seva evolució futura.

.

·Resum d'Anàlisi de Disseny - Simulador

El nostre disseny de la classe `Simulador` segueix bones pràctiques orientades a objectes però té alguns punts millorables:

El que hem fet bé:
- Hem separat clarament les responsabilitats principals (gestió d'esdeveniments, assignació de peticions, etc.)
- Aprofitem el polimorfisme pels diferents tipus de conductors i esdeveniments
- Encapsulem bé les dades amb getters/setters quan cal
- Tenim una bona jerarquia d'objectes (conductors, vehicles, peticions...)

Punts a millorar (SOLID):
1. **SRP**: Alguns mètodes com `finalitzarSimulacio` fan massa coses (mostrar stats + guardar fitxers). Millor separar-ho.
2. **OCP**: Si afegim nous tipus de conductors, cal tocar `assignarPeticions()`. Podríem fer-ho més extensible.
3. **DIP**: Depenem massa de classes concretes. Seria millor treballar amb interfícies.

Getters/Setters:
- En general bé, però alguns com `getMapa()` exposen massa. Millor oferir mètodes més específics.

Problemes destacats:
1. El constructor que llegeix JSON té massa lògica. Millor separar la lectura.
2. Gestió del temps una mica embolicada en alguns mètodes.
3. Podríem encapsular millor les col·leccions (llistes de vehicles, etc.)

Com ho faria ara:
1. Aplicaria el patró Strategy per als algorismes d'assignació
2. Crearia interfícies per a les dependències
3. Separaria la lògica de persistència (JSON)
4. Faria classes més petites i enfocades

Conclusió:
El disseny actual és sòlid i funciona, però amb el que sé ara aplicaria més estrictament SOLID per fer-lo:
- Més fàcil d'estendre
- Menys acoblat
- Més fàcil de testejar

El que tens és un bon punt de partida que amb petits ajustos pot quedar encara més net i mantenible.

Aquí tens l'anàlisi de disseny de la classe ConductorVorac seguint el format que has demanat:

· Resum d'Anàlisi de Disseny - ConductorVorac

La classe ConductorVorac implementa el comportament d'un conductor que planifica rutes de manera "vorac", gestionant tant la planificació com l'execució dels esdeveniments d'una ruta. Estén la classe Conductor i aplica lògica específica per a rutes de passatgers i càrrega.

El que hem fet bé

- **Herència ben utilitzada:** Estén Conductor de manera clara, implementant els mètodes abstractes amb lògica específica.
- **Ús correcte de composició:** La classe treballa amb col·laboradors com Simulador, Mapa, Ruta, i Vehicle.
- **Responsabilitat clara del rol:** Se centra en la planificació i execució de rutes com a conductor.
- **Documentació molt completa:** Cada mètode té especificació de pre/postcondicions i explicacions clares.

Punts a millorar (segons SOLID)

1. **SRP (Single Responsibility Principle):**
    - executarRuta() fa múltiples coses: gestiona horaris, calcula distàncies, afegeix esdeveniments, decideix si carregar bateria, etc. Podria dividir-se en mètodes auxiliars més petits (afegirMoviment(), gestionarPassatgers(), gestionarCarrega(), etc.)
    - També planificarRuta() té molta lògica de càlcul; es podria delegar a un servei de "planificació de ruta".
2. **OCP (Open/Closed Principle):**
    - Per suportar nous tipus de rutes o noves regles (com tarifes, condicions meteorològiques), caldria modificar mètodes existents. Aplicar patrons com Strategy o Template Method ajudaria.
3. **LSP (Liskov Substitution Principle):**
    - No s'aprecien violacions clares, però caldria assegurar que altres subclasses de Conductor també poden ser substituïdes sense trencar funcionalitats (ex: ús coherent de executarRuta() i teBateria()).

4. **ISP (Interface Segregation Principle):**
    - No és rellevant directament aquí, però si Conductor implementa moltes interfícies, podria ser un problema.
5. **DIP (Dependency Inversion Principle):**
    - ConductorVorac depèn directament de classes concretes com Mapa o Simulador. Es podria invertir la dependència via interfícies (IMapa, ISimulador), per facilitar proves i inversió de control.

Getters/Setters i Encapsulació

- Accés al vehicle via vehicle.getMaxPassatgers() i vehicle.getUbicacioActual() pot exposar massa detalls interns.
- Es podria encapsular millor algunes condicions com vehicle.teBateria(...) && !vehicle.esCarregant() en un mètode com vehicle.potFerRuta(...).

Problemes destacats

1. **Mètode executarRuta() molt llarg i amb massa responsabilitats.**
2. **Ús directe de System.out.println(...) per missatges d'error** → millor llançar excepcions o usar un logger.
3. **Decisions de lògica de negoci (com si cal carregar o no) barrejades amb generació d'esdeveniments.**
4. **Repetició de càlcul de distància/temps en múltiples llocs.**

Com ho milloraria

1. **Aplicaria el patró Strategy** per a la planificació de rutes (camivorac, camimesCurt, etc.).
2. **Refactoritzaria executarRuta()** en mètodes privats per millorar la llegibilitat i testabilitat.
3. **Injectaria dependències com Mapa o Simulador mitjançant interfícies.**
4. **Evitaria missatges de consola dins la lògica del model.**
5. **Crearia una classe PlanificadorRuta** per encapsular la planificació de rutes.

3.Analisis i explicació de mètodes principals

·Mètodes principals de la classe Conductor Planificador:

FUNCIONAMENT DEL MÈTODE planificarRuta(...) Planner

Aquesta funció és el cervell del **conductor planificador**. La seva feina és buscar, entre totes les peticions pendents, quines pot atendre en una sola ruta, i crear pas a pas aquest trajecte, tot respectant:

- La bateria del vehicle,
- Els horaris límit d'arribada dels passatgers,
- La capacitat màxima del vehicle,
- I si cal, afegint un desviament a un punt de càrrega.

Com funciona?

1. **Punt de partida**
   El vehicle arrenca des de la seva ubicació actual i comença a mirar quines peticions pot anar a buscar. També es guarda el nivell actual de bateria, l'hora i quants passatgers porta.

2. **Selecció de peticions (loop principal)**
   Aquest és el cor de la funció. A cada volta:
   - Mira totes les peticions pendents.
   - Les descarta si:
     - No hi ha camí fins a l'origen o fins a un punt de càrrega,
     - No hi arriba a temps,
     - No té prou bateria,
     - Ja ha passat l'hora límit,
     - No hi caben més passatgers.
   - I si alguna compleix tots els requisits, es guarda **la millor** (la que permet fer tot això en menys temps).

3. **Primera petició?**
   Quan troba la primera petició vàlida, la marca com a “primera”, ja que això determinarà quan hauria de sortir realment de la base per arribar-hi a temps.

4. **Afegeix la petició a la ruta**
   Si la petició trobada és bona:

    - L’afegeix a la ruta.
    - Actualitza la ubicació, el temps, la bateria i el nombre de passatgers.
    - Comprova si al final del trajecte haurà de carregar el cotxe. Si no té bateria suficient per arribar a un carregador, afegeix aquest desviament a la ruta.

5. **Finalització**
   El bucle para si:

    - No es poden afegir més peticions,
    - Ja s’ha afegit una petició **no compartida** (perquè llavors no es pot afegir ningú més al cotxe).

6. **Retorn de la ruta**

    - Si no ha trobat cap petició, torna null.
    - Si ha trobat una ruta vàlida, crea un objecte Ruta amb:
        - El recorregut,
        - L’hora real de sortida (ajustada segons el temps fins al primer client),
        - El conductor i distància total.
    - També s’hi afegeixen les dades de quants passatgers es recullen o deixen a cada punt.

Exemple en la pràctica

Imagina que un conductor comença amb bateria al 100% i cap passatger. Troba una petició compartida a les 9:30. Hi arriba a temps, així que la guarda. Com que és compartida, segueix buscant si pot recollir algú més de camí al mateix destí. Si pot, ho fa; si no, acaba la ruta i la retorna. Si al final veu que es quedarà sense bateria, també afegeix una parada al carregador abans d’acabar.

FUNCIONAMENT DEL MÈTODE executarRuta(...) Planner

Aquest mètode és el que **posa en marxa** la ruta que un conductor ja ha planificat prèviament. No decideix res: simplement converteix la ruta en **esdeveniments** que s'executaran dins la simulació (recollir, moure, deixar, carregar…).

Què fa pas a pas?

1. **Itera pel camí planificat**
   Per cada parell consecutiu de llocs (origen i destí):
    - Calcula la distància i el temps entre ells.
    - Actualitza l'hora actual sumant aquest temps.
    - Mira si cal recollir passatgers en el lloc d'origen: si sí, crea un RecollirPassatgersEvent.
    - Afegeix un marge d'1 minut abans de moure's.
    - Crea un MoureVehicleEvent per simular el desplaçament del vehicle.
    - Mira si cal deixar passatgers al lloc de destí: si sí, crea un DeixarPassatgersEvent.
2. **Finalització de ruta**
   Quan ja s'han recorregut tots els trams:
    - Si la ruta acaba en un **parquing per carregar**, i hi ha punts de càrrega disponibles:
        - Crea un CarregarBateriaEvent amb la duració corresponent (lenta o ràpida).
        - Mostra per consola que el vehicle ha arribat al punt de càrrega.
    - Si **no és una ruta de càrrega**, llavors es considera una ruta de transport normal i es genera un FiRutaEvent, que indica que el conductor ha acabat i pot tornar a planificar una nova ruta si hi ha peticions pendents.

Exemple pràctic

Suposem que una ruta planificada fa: A → B → C, on:

- A: s'hi recull un passatger.
- B: es deixa aquest passatger i se'n recull un altre.
- C: es deixa el segon passatger i es carrega el vehicle.

Aquest mètode s'encarregarà de crear i afegir al simulador els esdeveniments següents, en aquest ordre:

1. Recollir a A.
2. Moure d'A a B.
3. Deixar a B.
4. Recollir a B.
5. Moure de B a C.
6. Deixar a C.
7. Carregar bateria a C (si hi ha punt lliure).

·Mètodes principals de la classe Conductor Voraç:

FUNCIONAMENT DEL MÈTODE planificarRuta(...) Voraç

Aquest mètode s'utilitza quan un **conductor voraç** ha de planificar una ruta per una única petició concreta. És una planificació molt senzilla: només es busca el camí més directe entre l'origen i el destí de la petició, i no es tenen en compte altres peticions ni punts de càrrega. És per això que l'anomenem **voraç**: no optimitza res, simplement fa la feina ràpid.

Què fa exactament?

1. **Calcula el camí directe**
   A través del mapa, s'obté el camí més curt (camí voraç) entre l'origen i el destí de la petició.

2. **Calcula el temps i la distància totals**
   Amb el camí ja calculat, es recorren els trams un a un per acumular:

   - la distància total del trajecte,
   - i el temps estimat que es trigarà a completar-lo.

3. **Construeix la ruta**
   Amb tota aquesta informació es crea un objecte Ruta, indicant que no és una ruta de càrrega. L'hora d'inici de la ruta és l'hora mínima de recollida de la petició.

4. **Assigna el nombre de passatgers**
   Finalment, s'afegeix a la ruta quants passatgers cal recollir, segons el que diu la petició.

Exemple dins del funcionament global de la simulació

Quan comencem una simulació i assignem peticions als conductors **voraços**, aquest mètode és el que utilitzen per planificar les seves rutes. Com que la decisió de "quin conductor fa quina petició" es fa externament (i aquí és on hi ha un petit problema de responsabilitat única), aquest mètode només s'encarrega de **generar una ruta senzilla** per aquella petició.

Un cop generada, el simulador afegeix un esdeveniment IniciRuta i el vehicle comença a executar-la.

Notes de disseny

- El mètode **compleix el principi de responsabilitat única (SRP)**: només planifica rutes senzilles per a una petició. No assigna conductors ni controla l'estat de la simulació.

- No comprova si el vehicle té prou bateria o si el temps és suficient per arribar a l'hora. Aquestes comprovacions es fan **abans** de cridar aquest mètode (des del controlador o gestor de simulació).

FUNCIONAMENT DEL MÈTODE executarRuta(...) Voraç

Aquest mètode s'encarrega de transformar una **ruta ja planificada** en una seqüència d'**esdeveniments** dins del simulador. Un cop un conductor té assignada una ruta, aquest mètode s'executa per simular el seu recorregut pas a pas: recollir passatgers, moure's per la ciutat, deixar-los al destí, o bé carregar el vehicle si és una ruta de càrrega.

Què fa exactament?

1. **Inicia el recorregut**
   Agafa la llista de llocs de la ruta i l'hora d'inici. Es comença a simular a partir d'aquest punt.

2. **Recorre tram a tram la ruta**
   Per cada parell consecutiu de llocs (origen → destí):

   - Calcula quant temps i quants km caldran per arribar al següent lloc.

   - Si és el **primer tram d'una ruta de passatgers**, programa un esdeveniment de **recollida de passatgers**.

   - Afegeix un petit marge d'1 minut i crea un **esdeveniment de moviment** del vehicle.

   - Si és l'**últim tram d'una ruta de passatgers**, programa un esdeveniment per **deixar els passatgers** al destí.

3. **Finalització segons el tipus de ruta**

   - Si és una **ruta de càrrega**:
     Comprova si l'últim lloc és un Parquing amb un punt de càrrega disponible. Si n'hi ha, s'hi programa un esdeveniment de **carrega de bateria**.

   - Si és una **ruta de passatgers**:
     Un cop finalitzat el recorregut, s'afegeix un esdeveniment de **FiRuta**, que indica que el conductor ja ha acabat i pot ser assignat a una nova ruta.

Exemple dins del funcionament general

Quan s'assigna una petició a un conductor (voraç o planificador), i aquest planifica una ruta, el simulador crida aquest mètode per **programar tot el viatge**. A partir d'aquí, el Timer de la simulació s'encarrega d'anar executant els esdeveniments segons l'hora.

·Mètodes principals de la classe LectorCSV:

carregarLlocs(String pathFitxer)

Aquest mètode llegeix el CSV de **llocs**. Per cada línia:

- Si és un lloc normal (tipus L), es crea un objecte Lloc.
- Si és un parquing (P), es llegeixen els carregadors públics i privats, es creen, i s'afegeix un Parquing amb aquests punts.
- Si el tipus no es reconeix, es mostra un error.

carregarCamins(String pathFitxer, Map<Integer, Lloc> llocsPerId)

Aquest mètode llegeix els **camins** entre llocs (com connexions entre carrers).

- Llegeix origen, destí, distància i temps.
- Només afegeix camins si els dos llocs existeixen.
- Si falta algun lloc, s'avisa per consola.

Serveix per crear el mapa de la ciutat on es mouran els vehicles.

carregarPeticions(String pathFitxer, Map<Integer, Lloc> llocsPerId)

Carrega les **peticions** dels passatgers.

- Per cada línia es llegeix: origen, destí, horaris, nombre de passatgers i si volen vehicle compartit.
- Es comprova que els llocs d'origen i destí existeixin.
- Si tot és correcte, es crea una Peticio.

carregarConductors(String pathFitxer, Map<Integer, Vehicle> vehiclesPerId, Map<Integer, Lloc> llocsPerId)

Llegeix la informació dels **conductors**.

- Per cada línia, es mira quin tipus de conductor és: vorac o planificador.
- Els voraços només necessiten vehicle.
- Els planificadors també necessiten un Parquing assignat.
- Si el vehicle o parquing no existeixen, es descarta aquell conductor.

carregarVehicles(String pathFitxer, Map<Integer, Lloc> llocsPerId)

Aquest mètode carrega tots els **vehicles**.

- Per cada vehicle, es llegeix la ubicació inicial, autonomia, capacitat i temps de càrrega (ràpida i lenta).
- Es crea l'objecte Vehicle i s'afegeix a la llista.

Serveix per inicialitzar la flota amb la seva ubicació, energia i capacitat.

En resum:

Tots aquests mètodes formen part del procés d'**inicialització de la simulació**, i són cridats des del mètode inicialitzar(...) quan carreguem els fitxers CSV. Un cop tot és carregat correctament (llocs, camins, vehicles, conductors i peticions), podem començar la simulació amb totes les dades preparades.

rutaParquingMesProper(...)

Aquest mètode serveix per trobar **el parquing públic més proper** al lloc on es troba actualment un conductor, per tal que pugui anar a **carregar la bateria**.

Què fa exactament:

1. Recorre **tots els llocs** del mapa.
2. Si el lloc és un Parquing i **no està ple**, es calcula el camí des de l'origen fins a aquest parquing.
3. Si el camí existeix i és el més ràpid trobat fins ara, es guarda com la millor opció.
4. Finalment, retorna la ruta cap al parquing més proper **amb lloc lliure**.

Quan es fa servir?
Quan un conductor voraç ha de carregar la bateria després de servir una petició.

rutaParquingPrivatMesProper(...)

Aquest mètode busca el **parquing privat assignat al conductor planificador**, i hi calcula la ruta més ràpida per arribar-hi, sempre que:

- El parquing **no estigui ple**, i
- Sigui el que li toca al conductor (coincideix per ID).

Què fa:

1. Igual que l'anterior, però limita la cerca **només al parquing privat assignat al conductor planificador**.
2. Si el camí és vàlid i el parquing no està ple, el guarda com la millor ruta.
3. Retorna aquesta ruta perquè el conductor hi pugui carregar.

Quan s'utilitza?
Quan un planificador acaba una ruta i necessita tornar a casa seva (el seu parquing privat) per carregar.

Notes

- Els dos mètodes estan ben separats segons el tipus de conductor.
- Retornen una Ruta que ja es pot executar (igual com qualsevol altra ruta).
- Tenen una **assumpció** important: que el camivorac(...) funciona bé i sempre torna camins viables (si no, pot fallar silenciosament).

Comentari afegit (console output)

Al final del mètode privat, hi ha un System.out.println(...) que imprimeix el camí trobat. És útil per depurar (sobretot si vols veure per on passa la ruta), però probablement voldràs treure-ho o controlar-ho amb un flag de debug en la versió final.

·Mètodes principals de la classe Optimitzador:

obtenirConductorsRedundants(File JsonFile)

Aquest mètode serveix per **detectar conductors que no han estat útils** durant la simulació, és a dir, que **no han deixat cap passatger**.

Com funciona?

1. Es carreguen **totes les dades** (llocs, vehicles, conductors i esdeveniments) des del fitxer JSON de la simulació.

2. A partir dels esdeveniments executats, es compta **quantes vegades cada conductor ha deixat passatgers**.

3. Els conductors que no apareixen en cap DeixarPassatgersEvent s'etiqueten com a **redundants**.

Ús pràctic: ajuda a veure si s'han assignat conductors que **no han fet res realment útil**, i per tant, es podrien eliminar en una futura simulació.

obtenirPuntsCarregaRedundants(File JsonFile)

Aquest mètode analitza **quins punts de càrrega no s'han fet servir** durant tota la simulació.

Com funciona?

1. Es carrega tota la configuració i esdeveniments des del fitxer JSON.

2. Es compta **quantes vegades s'ha utilitzat cada punt de càrrega** (públic o privat).

3. Si un punt de càrrega no ha estat usat, apareixerà com a **redundant** al mapa resultant.

Ús pràctic: permet veure si s'han creat carregadors innecessaris que ocupen espai i complexitat, però no han estat útils.

Notes finals

- Ambdós mètodes es poden cridar des del menú de l'aplicació amb les opcions:
    - "Optimitzar simulació vehicles redundants"
    - "Optimitzar simulació punts de càrrega redundants"
- Retornen un mapa amb els ID dels recursos **i quantes vegades s'han fet servir**.
- Ideal per fer anàlisis post-simulació i millorar el disseny per la pròxima execució

FUNCIONAMENT GENERAL DE LA CLASSE Simulador

La classe Simulador és la peça central de la simulació. S'encarrega de controlar l'execució del temps, gestionar els vehicles, conductors, peticions i afegir/treure esdeveniments a la cua d'execució.

Hi ha dos constructors:

- Un que rep totes les dades carregades (per a una simulació nova).
- Un altre que reconstrueix una simulació des d'un fitxer JSON.

Un cop creada la instància, es poden assignar peticions, iniciar la simulació (amb Timer), visualitzar el mapa i generar estadístiques.

FLUX DE FUNCIONAMENT

- L'usuari carrega els fitxers (CSV/JSON).
- Es crea un Simulador.
- Es mostren els llocs i camins en un panell visual (MapPanel).
- Es poden afegir peticions aleatòries.
- En iniciar la simulació:
    - S'assignen peticions als conductors (voraços o planificadors).
    - Es generen esdeveniments que s'executen amb un Timer.
    -

- En acabar:
    - Es mostren estadístiques.
    - Es pot guardar la simulació i les estadístiques en JSON.

PUNTS DE MILLORA

- **Refactorització de responsabilitats**: delegar les funcions d'assignació i control d'esdeveniments en classes auxiliars.
- **Separació GUI/lògica**: el codi que mostra el JDialog d'estadístiques hauria d'estar fora del Simulador.
- **Millor testabilitat**: ara és molt difícil fer tests unitaris al Simulador perquè fa de tot.

CONCLUSIÓ

El disseny actual **funciona i és extensible**, però no compleix del tot els principis SOLID, especialment pel que fa a la responsabilitat única i la dependència. Malgrat això, es nota una bona organització general del codi, ús correcte del polimorfisme i separació entre parts com vehicles, peticions i conductors.

FLUX GENERAL DELS ESDEVENIMENTS

Cada esdeveniment hereta de la classe base abstracta Event, que defineix els atributs comuns (temps d'execució) i obliga a implementar el mètode executar(Simulador simulador).

Quan s'inicia la simulació, es planifiquen rutes i es generen esdeveniments com:

- IniciRutaEvent
- MoureVehicleEvent
- RecollirPassatgersEvent
- DeixarPassatgersEvent
- CarregarBateriaEvent
- FiRutaEvent
- FiCarregaEvent

Tots aquests són **executats en ordre cronològic** gràcies a una PriorityQueue.

FLUX GENERAL DE LA INTERFÍCIE

L'aplicació comença amb la **pantalla SelectorInicial**, on l'usuari pot carregar fitxers CSV i JSON. Des d'allà es poden iniciar o carregar simulacions, optimitzar punts de càrrega o conductors, o veure estadístiques.

Quan s'inicia la simulació, es mostra el MapPanel, que és el panell principal amb el mapa (llocs i camins), botons per afegir peticions i iniciar la simulació, i la llegenda (LegendPanel) i missatges en directe.

Quan finalitza, es mostren estadístiques dins d'un JDialog, o es poden veure gràficament amb EstadistiquesPanel, VehiclesComparisonPanel o VistaPuntsCarrega.

SOBRE CADA COMPONENT:

- **SelectorInicial.java**
  Vista principal per seleccionar fitxers d'entrada i accions. Bona separació de responsabilitats, tot i que es barregen decisions de lògica i vista en alguns punts (pot trencar el principi **S**).

- **MapPanel.java**
  Component principal de visualització del mapa. Dibuixa nodes, camins i vehicles. Té una mica massa responsabilitats (pinta, gestiona missatges, controla la llegenda), així que trenca parcialment **S (Single Responsibility)**.

- **LegendPanel.java**
  Bona aplicació de **S**: només mostra la llegenda del mapa. Simple i clar.

- **EstadistiquesPanel.java / VehiclesComparisonPanel.java / VistaPuntsCarrega.java**
  Panells que mostren gràfics amb estadístiques. Són vistes especialitzades, i **apliquen correctament el principi S**. Només mostren, no decideixen.

SOBRE ELS EVENTS

Tots els Event (com IniciRutaEvent, MoureVehicleEvent, CarregarBateriaEvent...) hereten de Event.java, i tenen un mètode executar(Simulador) que encapsula el seu comportament. **Aplicació correcta del principi O (Open/Closed)**: es poden afegir nous tipus d'event sense modificar la classe base.

Millorable: el Simulador coneix tots els detalls dels events i els seus efectes. Si es delegués més l'execució dins dels propis objectes implicats, s'aplicaria millor **D (Dependency Inversion)**.

POSSIBLES MAL DISSENYS DETECTATS

- El Simulador fa massa coses: controla la lògica, la GUI, les dades i l'estadística. Això **viola clarament el principi S (Single Responsibility)**.

- També coneix detalls concrets de molts altres objectes (vehicles, conductors, mapes, fitxers...), cosa que **viola el principi L (Law of Demeter)** i **I (Interface Segregation)** si afegim massa funcionalitat no relacionada.

- La classe Simulador també podria delegar l'assignació de peticions a estratègies externes (patró Strategy), aplicant millor el principi **O (Open/Closed)**.