1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

UNIVERSITÀ DEGLI STUDI DI URBINO CARLO BO

DIPARTIMENTO DI SCIENZE PURE E APPLICATE

INSEGNAMENTO PROGRAMMAZIONE E MODELLAZIONE AD OGGETTI

Relazione del Progetto d'Esame
Sessione Autunnale

Battistelli Stefano

Grasso Emanuele

Rinaldi Simone

Scaramuzzino Elia

Anno Accademico 2021/2022

Indice

Capitolo 1

1 Analisi

Il gruppo si pone come obbiettivo quello di realizzare un applicativo attraverso l'utilizzo del linguaggio con paradigma di programmazione ad alto livello orientato agli oggetti(OOP), *Java*.

Questo applicativo si concentra sul facilitare lo scambio di materiale tra più direttori a capo di aziende con dei lavoratori all'interno, supervisionati da un unico Manager (ossia l'utente), sfruttando un mezzo di trasporto.

1.1 Requisiti del Progetto

L'applicativo dovrà essere in grado di permettere all'utente di creare il proprio impero di aziende e direttori, senza sforzi. L'utente si immedesimerà quindi in un Manager in grado di poter assumere e licenziare qualsiasi direttore in qualsiasi momento.

La creazione dei direttori verrà gestita totalmente dall'utente, non ci siamo posti limiti su quello che potrebbe creare.

I direttori lasceranno il libero controllo all'utente per tutta la parte relativa alla creazione ed accettazione delle richieste.

L'applicativo dovrà garantire un corretto circolo delle richieste tra i vari direttori spedendo le stesse solo a coloro in grado di soddisfarle.

L'applicativò si occuperà del trasporto del materiale attraverso un veicolo assimilabile ad un treno.

Il treno dovrà essere in grado di stilare autonomamente una lista delle varie tappe da raggiungere (in ordine FIFO) e maneggiare il proprio carico depositandolo o prelevandolo dai vari magazzini, lasciando però all'utente la scelta di quando spostarsi alla stazione successiva.

La lavorazione del materiale verrà gestita totalmente dall'utente, attraverso lo staff, il quale simulerà dei turni di lavoro che si suddivideranno in "Fase di inizio lavoro" e "Fase di fine lavoro"; durante i quali l'applicativo gestirà autonomamente il prelievo del materiale dal magazzino di carico, la lavorazione dello stesso ed il deposito del materiale lavorato nel magazzino di scarico.

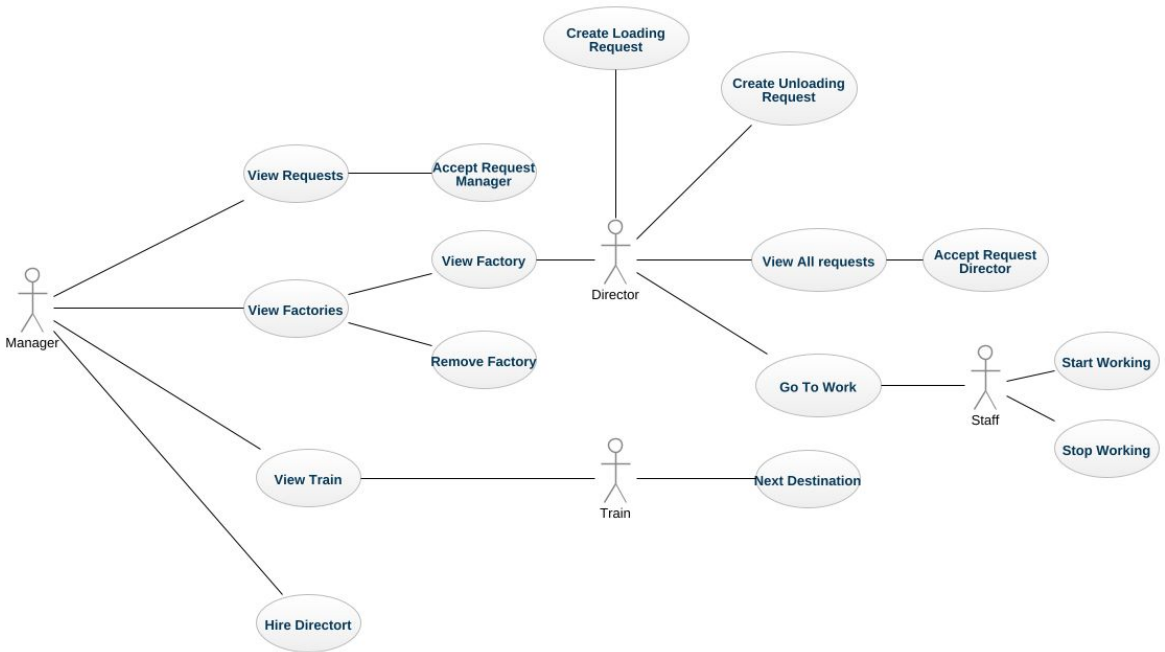

Figura 1: Use Case Diagram

1.2 Modello del Dominio

Inizieremo analizzando le figure adottate per l'implementazione di questo programma:

- `Manager`: Figura principale utilizzata per il coordinamento delle varie componenti e la comunicazione tra di essi.

  In particolare detiene il controllo totale sulla figura del direttore.
- `Director`: Figura secondaria, a capo di una singola azienda, che verrà utilizzata per la gestione della stessa.

  In particolare si occuperà di gestire la produzione di un materiale; dalla richiesta di materia prima sino alla vendita del prodotto lavorato.
- `Factory`: Figura assegnata ad un unico direttore dove avverrà la lavorazione del materiale.
- `Material`: Rappresentazione di ciò che viene lavorato all'interno di un'azienda.
- `Staff`: Figura che rappresenta il personale dell'azienda utilizzato per la lavorazione vera e propria.
- `Warehouse`: Figura posseduta da ogni azienda sotto forma di "magazzino di carico" e "magazzino di scarico".

  In particolare verrà utilizzata come deposito per il materiale.
- `Train`: Figura attua a simulare un mezzo di trasporto attraverso il quale la merce verrà spostata per essere caricata e scaricata dai magazzini delle varie aziende.
- `Request`: Figura astratta utilizzata al fine di semplificare la procedura di trasporto merce.
- `Store`: Figura alla quale verrà inviata la merce in eccesso.

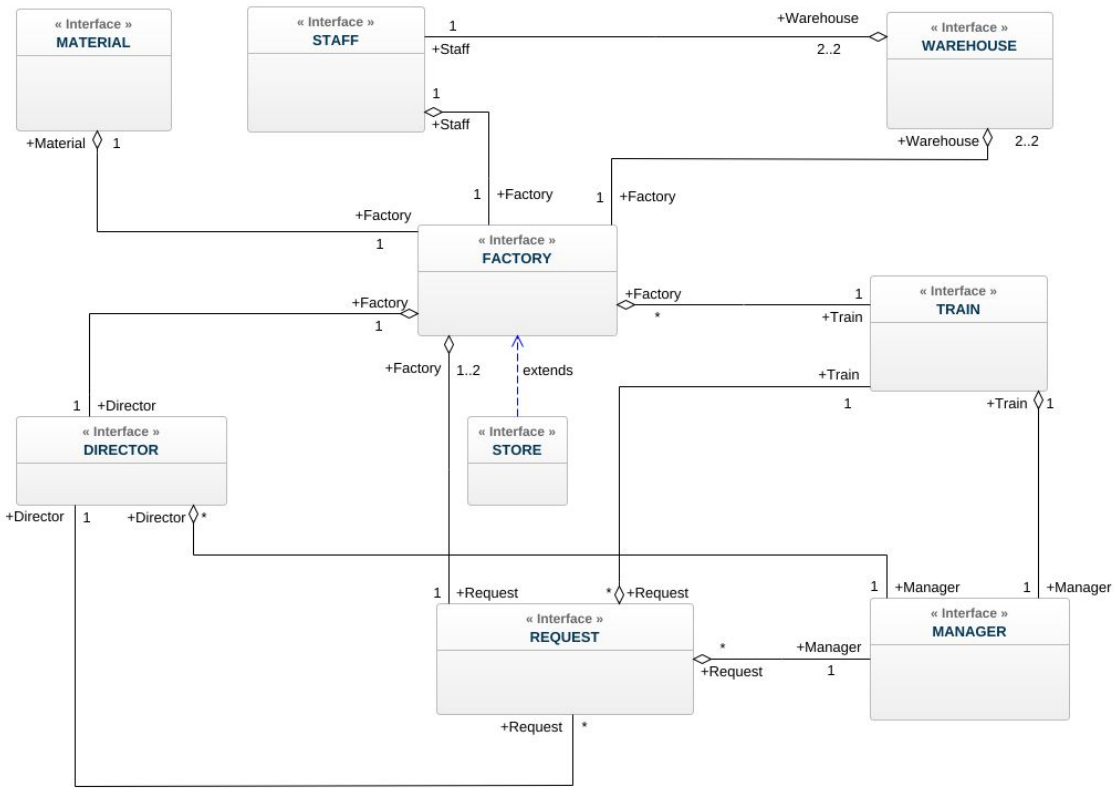

Figura 2: UML Diagram

Capitolo 2

2 Design

La classe principale dell'applicazione è **ManagerImpl**: in essa vengono istanziati i vari direttori (ai quali verrà associata una delle aziende) rappresentati attraverso **DirectorImpl**.

Le aziende sono rappresentate dalla classe **FactoryImpl** e da una sua estensione, denominata **StoreImpl**, la quale è unica.

La classe **FactoryImpl** è composta dallo staff, ovvero gli operatori che vi lavorano all'interno, rappresentati dalla classe **StaffImpl**;

Inoltre la classe **FactoryImpl** possiede un magazzino di carico e uno di scarico, rappresentati dalla classe **WarehouseImpl**; in essi verranno depositati materiale grezzo e lavorato, a seconda del riferimento del materiale dettato dalla classe **MaterialImpl**.

I direttori, per interagire tra di loro e col manager, si avvalgono dell'uso della classe **RequestImpl** che permette l'invio di una richiesta di carico o scarico merce.

Le richieste di carico, potranno essere risolte dal Manager in maniera diretta per conto di terzi, risolvendola immediatamente.

Nel caso venga risolta per conto di altri direttori, questi dovranno accettare la richiesta e spedire il materiale tramite l'utilizzo della classe **TrainImpl** che estrapolerà dall'oggetto 'richiesta' l'azienda mittente e destinataria, così da sapere dove dirigersi per caricare e scaricare la merce.

Per quanto riguarda le richieste di scarico, se non è possibile scaricare all'interno del magazzino di scarico dell'azienda, il treno imposterà tra le sue destinazioni il negozio rappresentato dalla classe **StoreImpl**.

2.1 Architettura

Per la realizzazione di questo progetto, ci siamo basati sull'utilizzo di due pattern ben conosciuti: **MVC**(Model-View-Controller) Design Pattern e **Singleton** Design Pattern.

2.1.1 *MVC Design Pattern*

Tramite l'utilizzo del pattern MVC, abbiamo deciso di assegnare il ruolo di **Controller** alla classe `ManagerImpl`, essendo colui che detiene idealmente il controllo su tutte le altre componenti ed, inoltre, garantisce lo scambio di informazioni tra la view ed il model.

La **View** rappresenta la visualizzazione dei dati contenuti nel Model, la quale avrà il compito di gestire l'interazione tra utente e dati/funzionalità, interfacciandosi ad esso attraverso diverse GUI. Quest'ultime si aggiorneranno progressivamente interagendovi ed effettuando le opportune operazioni consentite.

Il **Model** è la parte di MVC che implementa la logica di dominio. In parole povere, questa logica viene utilizzata per accedere ai dati utili dell'applicativo.

2.1.2 *Singleton Design Pattern*

Per quanto riguarda l'ottenimento dell'unicità degli oggetti della classe `ManagerImpl` e dello `Store` è stato adottato l'utilizzo del Singleton Design Pattern per garantirvi questa proprietà.

2.2 Design Dettagliato

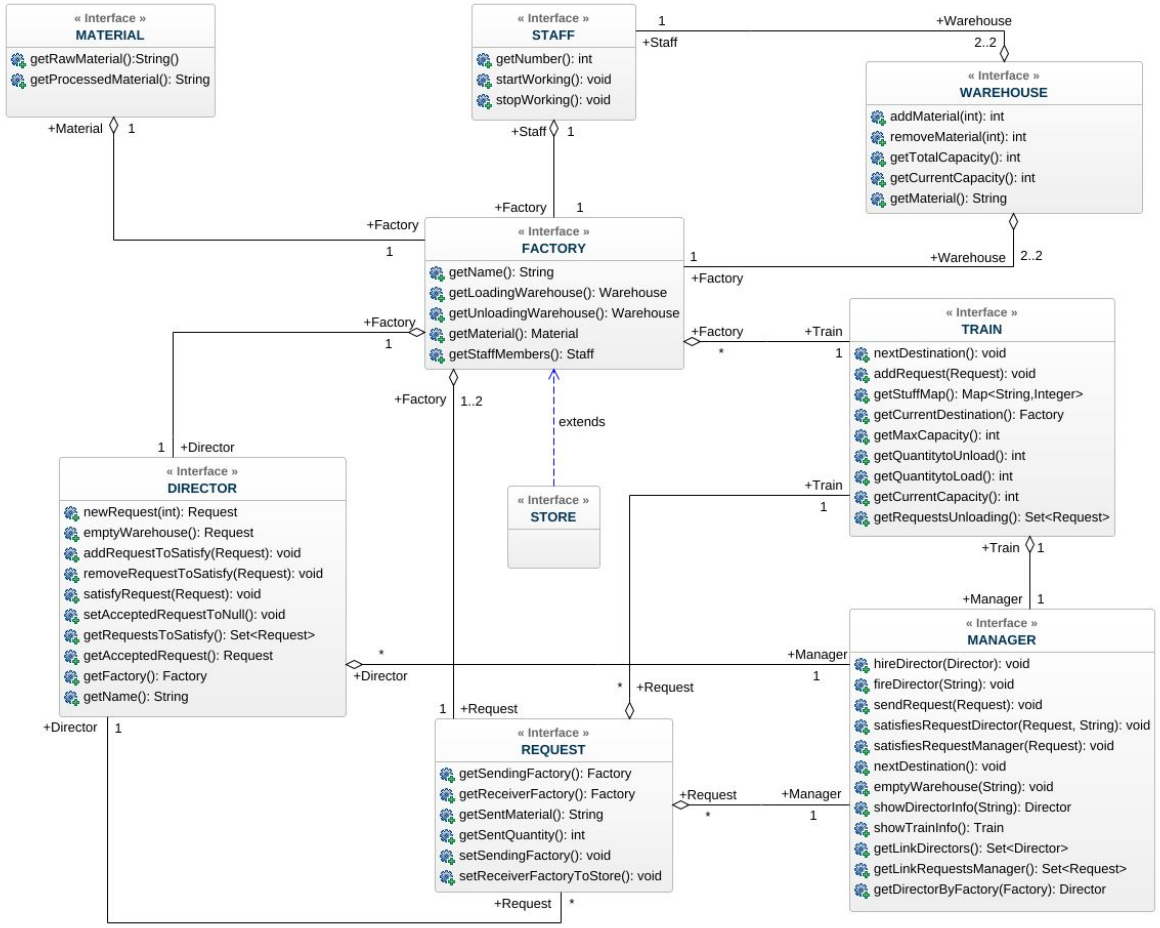

Figura 3: UML Diagram

In questa sezione andremo ad approfondire alcuni elementi del design con maggior dettaglio.

Ricordiamo inoltre che, in questa sezione, ogni membro del team ha scritto la propria sottosezione in maniera autonoma.

2.2.1 Battistelli Stefano

Interfacce Realizzate:

- Manager.java

Classi Realizzate:

- ManagerImpl.java
- FirstPageFrame.java
- FactoryFrame.java
- ManagerFrame.java
- HireDirectorFrame.java
- ViewRequestFrame.java

Il controller di questo progetto è il manager che detiene il controllo su tutte le figure e, pertanto, è un'entità unica all'interno del nostro software.
Per renderlo tale, abbiamo adottato il 'SingleTon' design pattern, che permette di istanziare il manager una ed una sola volta, garantendogli il controllo totale dell'applicativo.

La classe fornisce dei metodi per poter interagire con tutte le altre entità, ad esempio, per comunicare con i suoi direttori sfrutta una struttura dati di tipo `HashSet()` che contiene i riferimenti ad essi ed alle loro informazioni; servendosi di questa struttura il manager può assumere e/o licenziare i vari direttori.

La scelta ricaduta su questa struttura dati è stata dettata dal fatto che non si necessita di un ordine determinato e non si ritiene necessario mantenere un riferimento all'ordine di inserimento;
inoltre si è optato per tale struttura data la necessità di gestire elementi unici.

Il manager ha il compito di gestire la comunicazione tra i direttori ed il treno attraverso le richieste.

2.2.1.1 Punti Critici: Manager

Qualora e solo nel caso in cui non sia presente neanche un direttore in grado di soddisfare una richiesta, questa verrà considerata di emergenza ed inviata al manager.
La spedizione della merce da parte del manager avviene per conto di terzi, non passerà tramite il treno.

Tutte le entità dell'applicativo verranno istanziate, in seguito al Manager ad eccezione della figura del treno, la cui creazione avverrà in contemporanea.

Ulteriori punti critici si sono riscontrati durante l'implementazione della figura del manager all'atto dell'assunzione e del licenziamento dei direttori; in quanto si è ritenuto necessario dover gestire e aggiornare le richieste già in circolo nel programma.

A fronte di ciò si è optato per una gestione interna delle richieste attraverso due strutture dati ben distinte, una per le richieste soddisfacibili dai direttori `linkGlobalRequests` e l'altra per le richieste unicamente accettabili dal manager `linkRequestsManager`.
Anche in questo caso si è optato per due strutture dati di tipo HashSet, poichè non si necessita di un ordine determinato e non si ritiene necessario mantenere un riferimento all'ordine di inserimento;
inoltre si è optato per tale struttura data la necessità di gestire elementi unici.

Un ulteriore punto critico lo riscontriamo nel metodo `hireDirector()` il quale simula l'assunzione di un nuovo direttore, aggiungendo tale entità al set `linkDirectors`, attribuendogli le richieste da lui soddisfacibili già presenti all'interno dell'applicativo.

Analogamente all'assunzione, il metodo `fireDirector()` simula il licenziamento di un direttore attraverso la rimozione di esso dal set 'linkDirectors'; anche in questo caso, aggiornando i parametri delle richieste, nello specifico:

- Rimuovendo dall'applicativo le richieste precedentemente create dal direttore appena licenziato;
- Indirizzando verso il negozio tutte le richieste precedentemente indirizzate all'azienda del direttore licenziato;
- Rimettendo in circolo le richieste presenti nel set 'requestsToSatisfy' del direttore appena licenziato rispettando le condizioni precedentemente indicate.

2.2.2 Grasso Emanuele

Interfacce Realizzate:

- Director.java
- Request.java
- Material.java

Classi Realizzate:

- DirectorImpl.java
- MaterialImpl.java
- RequestImpl.java
- DirectorFrame.java
- LoadingRequestPopup.java
- UnloadingRequestPopup.java

Non vengono segnalate particolari scelte di progetto relative alle richieste ed ai materiali.

Il direttore, invece, ricopre un ruolo molto importante nella gestione della componente comunicativa (in quanto protagonista degli scambi di richieste) e su questo si è concentrato maggiormente il lavoro relativo a tale sezione.

Sfruttando un set interno di tipo `HashSet()`, si riescono a manovrare le richieste in entrata e, attraverso una variabile di appoggio, si riescono anche a controllare le eventuali richieste accettate dall'utente attraverso il direttore di turno.

Per quanto riguarda lo scambio effettivo di queste, si è optato per l'impiego del manager che, per sua natura, è in grado di tenere sotto controllo tutti i direttori contemporaneamente e quindi favorire tale operazione.

Ricordiamo inoltre che ogni direttore è unico e differisce da altri direttori per il nome o per l'azienda ad esso assegnata.

2.2.2.1 Punti Critici: Director

Passando ad osservare il componente più nel dettaglio:

Il direttore è una fonte molto pregna di eccezioni in quanto queste potrebbero verificarsi ad ogni creazione di una nuova richiesta.

Altre eccezioni possono presentarsi durante la creazione dell'oggetto stesso in quanto, per non rovinare la grafica, si è optato di porre un limite massimo ai caratteri dedicati al nome (pari a 12) ed un limite minimo (pari a 1).

Essendo il nome una componente unica del direttore, questa sarà fondamentale per permettere al manager di gestire al meglio il proprio set `linkDirectors`.

2.2.3 Rinaldi Simone

Interfacce Realizzate:

- Train.java
- Factory.java

Classi Realizzate:

- TrainImpl.java
- FactoryImpl.java
- TrainFrame.java

Una figura fondamentale per la realizzazione di questo progetto è la Factory, ovvero l'oggetto dotato di due magazzini che interagisce con il treno per il carico e lo scarico del materiale, il quale verrà lavorato dai membri dello staff interni a quest'oggetto.

La classe Factory, d'altronde, non conterrà nessun particolare metodo.

2.2.3.1 Punti Critici: Factory

Ogni azienda è unica e differisce dalle altre per il nome.

Ritenendo tale componente essenziale, provvederemo a validarla attraverso un limite massimo di caratteri dedicati(pari a 12) ed un limite minimo (pari a 1).

Sfrutteremo quindi questa caratteristica per favorire l'unicità nella creazione dei direttori.

L'entità Treno è una delle figure più importanti di questo progetto, poichè è responsabile del trasporto del materiale tra le aziende.
All'interno di esso troviamo quattro strutture dati principali:

- `loadingRequests:` Set attuo a contenere le richieste di carico;
- `unloadingRequests:` Set attuo a contenere le richieste di scarico;
- `stuffMap:` Mappa utilizzata per tenere traccia del materiale a bordo del treno;
- `destinationsQueue:` Coda unica adottata per tenere traccia delle tappe che il treno dovrà raggiungere.

Si è ritenuto necessario l'impiego di due set contenenti le richieste data l'unicità di queste.
Per la gestione della `stuffMap` si è optato per l'utilizzo di una struttra dati `LinkedHashMap()` per mantenere il riferimento all'ordine di inserimento dei materiali.

La scelta della struttura dati è ricaduta sulla mappa poichè essa mette a disposizione una tecnica di salvataggio dei dati tramite l'utilizzo di un coppia $< chiave, valore >$.
Con questo meccanismo è possibile tener traccia del materiale a bordo del treno facilmente, associando alla chiave il nome del materiale e al valore il quantitativo di quest'ultimo.

La comunicazione che avverrà tra i set e la mappa è fondamentale per garantire un corretto funzionamento delle operazioni di carico e scarico materiale.

Si è ritenuto necessario implementare una coda di Factory di tipo `UniqueQueue()`, ossia una classe creata ad hoc, la quale sfrutta un set di tipo `HashSet()` ed una coda di tipo `LinkedList()` interni, così da garantire l'inserimento di aziende in ordine FIFO ed un'unicità sugli elementi.

2.2.3.2 Punti Critici: Train

All'interno della figura del treno, ciò che rappresenta una maggior presenza di punti critici è il metodo di gestione di carico e scarico della merce, ovvero `CargoManagement()`, in grado di generare diversi tipi di eccezione per via delle diverse situazioni che si ritrova a gestire.

Per non rendere scomodo all'utente l'utilizzo dell'applicativo a causa di un mezzo di trasporto poco capiente, abbiamo deciso di inserire un limite di capienza ritenuto minimo.

2.2.4 Scaramuzzino Elia

Interfacce Realizzate:

- Warehouse.java
- Staff.java
- Store.java

Classi Realizzate:

- WarehouseImpl.java
- StaffImpl.java
- StoreImpl.java
- StaffFrame.java

Lo Staff rappresenta il personale di ogni singola azienda, supervisionata da un direttore.
È la figura che ha il compito di produrre del materiale finito (conservato nel magazzino di scarico) a partire da del materiale grezzo (conservato nel magazzino di carico).

Il numero di operatori all'interno dello staff, viene definito nell'atto di creazione dell'azienda ed impatta sulla quantità di materiale prodotto per ogni ciclo di lavoro.

2.2.4.1 Punti Critici: Staff

L'unico punto critico relativo allo staff è in fase di creazione in quanto il numero degli operatori non può superare la capienza dei magazzini.

L'entità Warehouse rappresenta la struttura dove verrà stipata la merce di carico `loadingWarehouse` e scarico `unloadingWarehouse` dell'azienda.
Ogni azienda possiede un unico magazzino di carico ed un unico magazzino di scarico.

La definizione della capacità dei magazzini, rapportata al numero di operatori dello staff dell'azienda, viene stipulata durante l'istanziamento della stessa.

2.2.4.2 Punti Critici: Warehouse

Anche in questo caso l'unico punto critico è in fase di creazione in quanto la dimensione dei magazzini dev'essere rapportata alla capienza del treno.

La figura Store all'interno del nostro applicativo è un'estensione della classe Factory.
Il compito di tale entità è quello di immagazzinare il materiale che il treno non è capace di scaricare nei magazzini di carico delle aziende (a causa del raggiungimento della capienza massima di questi ultimi) ed a seguito della scelta di voler svuotare un magazzino per conto del direttore.

2.2.4.3 Punti Critici: Store

L'entità Store non è dotata di capienza e, per questo, si è ritenuta accettabile l'idea di avere un'unica istanza di questa classe.
Per ovviare a questo problema abbiamo utilizzato il SingleTon Design Pattern.

Capitolo 3

3 Sviluppo

3.1 Testing Automatizzato

Per il testing automatizzato abbiamo scelto di utilizzare il framework di unit testing JUnit 5, portando avanti il progetto insieme all'idea di sviluppo guidato da test (Test Driven Development).

Il motivo principale che giustifica tale metodologia di sviluppo è dettato dal fatto che ad ogni nuovo metodo o feature aggiunta ad una classe è possibile verificarne subito il suo funzionamento in modo indipendente.

Attraverso i metodi forniti dalla suite di JUnit 5 abbiamo scritto i test per verificare passo a passo se i risultati restituiti da un metodo, oppure una nuova feature, fossero esattamente quelli che ci stavamo aspettando.

L'utilizzo di questo framework, si è rivelato molto efficiente per la verifica del funzionamento corretto delle interazioni tra le varie componenti del progetto, senza dover rilanciare l'interfaccia grafica simulando le situazioni da verificare svariate volte.

3.2 Metodologia di Lavoro

Per la realizzazione di questo progetto, inizialmente, abbiamo scelto collettivamente una specifica favorevole a tutti i componenti del team.

Ci siamo subito concentrati sull'analisi dell'applicativo da realizzare che ci ha portato alla definizione del dominio ed all'implementazione del diagramma dei casi d'uso(Use Case Diagram).

In esso, attraverso un linguaggio grafico, abbiamo descritto l'interazione dell'utente o degli utenti con l'applicativo.

La fase successiva è stata la costruzione del diagramma dell'UML (Unified Modeling Language) del progetto, attraverso il quale abbiamo definito le entità che comporranno il nostro applicativo, comunicando tra di loro.

Dopo aver definito il dominio e lo 'scheletro' del progetto, abbiamo iniziato la stesura dei test, seguita dalla suddivisione del lavoro tra i vari membri del team, per la realizzazione delle classi rappresentate nel diagrammaUML.

Abbiamo optato sin da subito per un lavoro in parallelo con tutti i membri del gruppo sfruttando la piattaforma *GitHub*.

In questa sezione ogni membro del team ha scritto la propria sottosezione in maniera autonoma:

3.2.1 Battistelli Stefano

Come descritto nella sezione di Design Dettagliato, il lavoro svolto è stato incentrato totalmente sulla creazione del controller.

Data la sua grande flessibilità è stata necessaria una grande comunicazione con i detentori delle classi relative ai Direttori, alle Richieste ed al Treno

La parte svolta in singolo è relativa al corpo dei metodi secondari, specialmente quelli attui ad ovviare la ripetizione di codice.

La scrittura relativa al corpo dei metodi ritenuti principali (ossia `hireDirector()` e `fireDirector()`) è stata svolta in singolo e, per una maggiore pulizia del codice, revisionata da Emanuele Grasso prima della convalida.

Lo sforzo principale è stato riscontrato nelle ricerche ed approfondimento del Singleton Design Pattern.

Inoltre la scrittura della relazione, degli UML e dello UseCase è stata svolta in contemporanea con gli altri membri del team.

3.2.2 Grasso Emanuele

Come descritto nella sezione di Design Dettagliato, il lavoro svolto è stato incentrato principalmente sulla creazione del Direttore, delle Richieste e del Materiale.

La parte svolta in singolo è stata quella relativa alla scrittura delle classi di Richieste e Materiale, in quanto dotate unicamente di metodi Getter e Setter.

L'implementazione relativa alla classe del Direttore è stata discussa con gli altri membri del team, in quanto strettamente collegata ad ognuno di loro.

Lo sforzo principale è stato riscontrato nel coordinamento dei lavori del team, nella stesura dei test e nella riscrittura dei metodi di `hashCode()`.

Inoltre la scrittura della relazione, degli UML e dello UseCase è stata svolta in contemporanea con gli altri membri del team.

3.2.3 Rinaldi Simone

Come descritto nella sezione di Design Dettagliato, il lavoro svolto è stato incentrato principalmente sulla creazione del Treno e delle Aziende.

La parte svolta in singolo è stata quella relativa alla scrittura della classe delle Aziende, in quanto dotata unicamente di Getter e Setter.

L'implementazione relativa alla classe delle Aziende e del Treno è stata discussa con gli altri membri del team, in quanto strettamente collegate ad ognuno di loro.

La scrittura relativa al corpo del metodo `cargoManagement()`, ritenuto il principale metodo del treno, è stata svolta in singolo e, per una maggiore pulizia del codice, revisionata da Emanuele Grasso prima della convalida.

La creazione della classe `UniqueQueue()` e la riscrittura del metodo `hashCode()` relativa all'azienda sono state svolte collaborando assieme al collega Grasso Emanuele.

Lo sforzo principale è stato riscontrato nell'ideazione ed astrazione del metodo principale del treno, in quanto molto delicato ed ampiamente discusso con i colleghi Battistelli Stefano e Grasso Emanuele.

Inoltre la scrittura della relazione, degli UML e dello UseCase è stata svolta in contemporanea con gli altri membri del team.

3.2.4 Scaramuzzino Elia

Come descritto nella sezione di Design Dettagliato, il lavoro svolto è stato incentrato principalmente sulla creazione dei Magazzini, del Negozio e dello Staff.

La parte svolta in singolo è stata quella relativa alla scrittura del Negozio in quanto non strettamente collegata al resto del progetto.

La scrittura relativa alla classe dei Magazzini e dello Staff è stata discussa in parallelo con i detentori delle classi relative ai Direttori ed al Treno.

Lo sforzo principale è stato riscontrato nelle ricerche ed approfondimento del Singleton Design Pattern e nell'astrazione del Negozio come magazzino infinito.

Inoltre la scrittura della relazione, degli UML e dello UseCase è stata svolta in contemporanea con gli altri membri del team.

3.3 Note Di Sviluppo

Funzionalità avanzate adottate per lo sviluppo del progetto da:

3.3.1 Battistelli Stefano

- Lambda Expression;
- Utilizzo di Stream;
- Scrittura degli algoritmi relativi all'assunzione ed al licenziamento dei direttori.

3.3.2 Grasso Emanuele

- Lambda Expression;
- Utilizzo di Stream;
- Utilizzo di id statici per le richieste;
- Ridefinizione del metodo `hashCode();`
- Ridefinizione del metodo `equals()`.
- Creazione `UniqueQueue();`

3.3.3 Rinaldi Simone

- Lambda Expression;
- Utilizzo di Stream;
- Ridefinizione del metodo `hashCode();`
- Ridefinizione del metodo `equals();`
- Scrittura dell'algoritmo relativo allo scarico ed al carico;
- Creazione `UniqueQueue();`
- Utilizzo di Generici.

3.3.4 Scaramuzzino Elia

- Creazione del Negozio.