Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

---

Tesi di Laurea Magistrale in Ingegneria Informatica

Title...

Candidato
Lorenzo Niccolai

Relatore
Prof. Enrico Vicario

Correlatore
???

Anno Accademico 2019/2020

Indice

Introduzione

L'obiettivo di questo lavoro è esporre un servizio su cloud in forma di *software as a service* che implementi una funzione basata su Oris nel contesto del intelligent transportation per tramvie.

Sarà progettata e realizzata una moderna applicazione a servizi che implementa una serie di casi d'uso, e successivamente verrà proposta una metodologia che guida la rifattorizzazione di tale software verso la forma di microservizi, applicando a vari livelli il proncipio di "separazione delle responsabilità".

Sarà inoltre posta attenzione a varie tecnologie che potrebbero essere adottate nelle fasi di progettazione, sviluppo e rilascio.

Capitolo 1

Applicazione

In questo capitolo saranno esposti i requisiti dell'applicazione, seguiti da una breve analisi e progettazione di un sistema che li implementa.

1.1 Requisiti

L'obiettivo principale è quello di poter definire e analizzare modelli virtuali di incrocio stradale, in modo da riuscire ad effettuare aggiustamenti sul traffico e quindi minimizzare l'impatto che può avere un mezzo come la tramvia sulla circolazione degli altri veicoli.

Per mantenere generale l'applicazione, dovrà essere possibile la descrizione di porzioni più ampie di mappa stradale in modo da supportare analisi di tipo diverso, nonché eventuali simulazioni.

In questo lavoro saranno realizzate le seguenti analisi esemplificative:

- Disponibilità di un incrocio: la probabilità che le auto siano libere di procedere, ovvero che il semaforo sia verde.

- Lunghezza media di una coda: Considerando alcuni fattori come il tempo di attraversamento, si vuol sapere quale sarà la dimensione della coda delle auto.
- Tasso di overflow: Caso in cui un'auto non riesce ad entrare nella via a causa del traffico eccessivo.

Tramite il software sarà possibile definire, oltre alla topologia della rete stradale, una serie di parametri necessari all'analisi di essa. Alcuni dei valori gestiti saranno:

- Periodo dell'analisi: cadenza con la quale i treni arrivano in prossimità dell'incrocio;
- Eventuale ritardo di arrivo del tram;
- Tasso di arrivo delle auto;
- Tempo di attraversamento delle auto;
- Dimensione massima della coda (calcolabile dalla lunghezza del tratto di strada e dalla dimensione dei veicoli);
- Tempo di attraversamento del tram;
- Tempo di anticipo del semaforo rispetto all'arrivo; effettivo del treno;

Sempre con lo scopo di supportare analisi non ancora stabilite, sarà possibile impostare sui vari elementi della rete un numero arbitrario di parametri. Questi saranno suggeriti dagli stessi moduli di analisi.

1.2 Casi d'uso

Sono state individuate tre categorie di utilizzatori (vedi figura 1.1):

- Amministratore
- Esperto
- Cliente

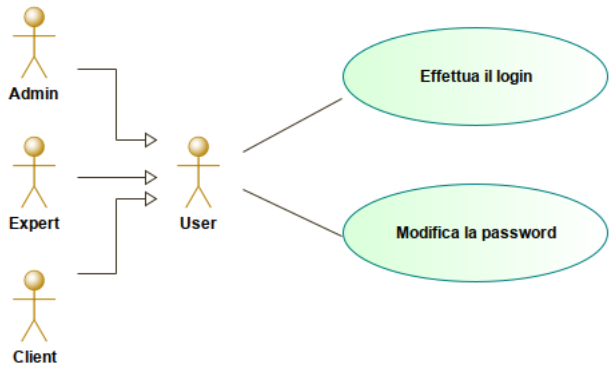

Figura 1.1: Classificazione degli utenti

Ogni utente potrà accedere all'applicazione tramite una funzionalità di login, così da avere accesso alle aree del software di propria competenza, nonchè alla sezione di gestione del profilo.

Gli utenti di tipo *amministratore* hanno la possibilità di gestire gli accounts aggiungendone di nuovi o eliminandone di esistenti (figura 1.2). Per semplicità sarà lo stesso amministratore a stabilire le password iniziali, che potranno essere modificate in seguito dai proprietari.

Gli utenti *esperti* conoscono a fondo il dominio e sono in grado di configurare un modello specificando le varie proprietà. Per questo hanno il permesso di definire valori di default o complete configurazioni per gli utenti meno esperti (figura 1.3).

Figura 1.2: Casi d'uso di un amministratore

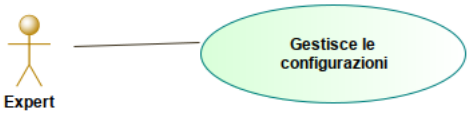

Figura 1.3: Casi d'uso di un utente esperto

Gli utenti semplici possono creare dei progetti personali. Questi raggruppano una serie di topologie di rete. Una volta creata e validata una mappa, sarà possibile lanciare delle analisi che rimarranno legate ad essa e delle quali sarà possibile monitorare lo stato ed accedere ai risultati.

In figura 1.4 è mostrato un semplice diagramma concettuale che sarà poi raffinato nelle fasi successive. É possibile apprezzare le relazioni tra le varie entità nominate precedentemente.

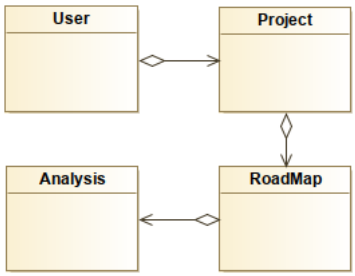

Figura 1.4: Schema concettuale

1.3 Modellazione della rete stradale

Interesse di questo lavoro è lo studio del traffico nei pressi di un incrocio tra auto e tramvia. Per supportare una visione più ampia, ed altre eventuali tipologie di analisi, sarà proposto un modello in grado di descrivere una generica rete di trasporti.

1.3.1 Il grafo dei trasporti

Si può pensare di modellare una rete stradale con un grafo in cui gli archi e i nodi rappresentino rispettivamente le carreggiate delle strade ed i punti di interesse come gli incroci. Questi due concetti saranno rappresentati dagli elementi *RelevantPoint* e *LaneSegment*. Nel suo caso più basilare tale definizione corrisponde precisamente con quella matematica di grafo orientato, ovvero:

$$G = (V, E)$$

Con V insieme dei vertici ed $E \subseteq V \times V$ insieme di coppie ordinate di vertici.[8] In figura 1.5 sono mostrate le entità e i legami basilari tra esse: l'unico vincolo importante è che un segmento di strada deve avere come sorgente e destinazione un punto rilevante.

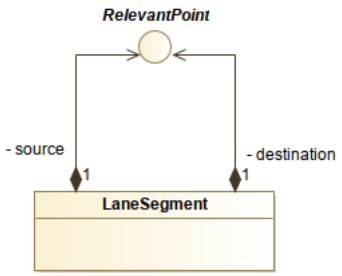

Figura 1.5: Entità basilari in un sistema stradale

Si definisce grado di entrata in un nodo $\delta_i(v_0)$ la cardinalità del seguente insieme:

$$\{(v, v_0) \in E\}$$

Mentre il grado di uscita $\delta_o(v_0)$ è la cardinalità di:

$$\{(v_0, v) \in E\}$$

I nodi con grado di entrata o di uscita nullo hanno speciale significato in questo ambito, ovvero se dato $v_s \in V$, $\delta_i(v_s) = 0$, allora v_s è un nodo di entrata nel sistema, mentre se dato $v_d \in V$, $\delta_o(v_d) = 0$, allora v_d è nodo di uscita dal sistema (1.6).

Siano S e D i rispettivi insiemi di nodi sorgente e destinazione, è stata introdotta la seguente limitazione per semplificare il lavoro di analisi:

$$v_s \in S \Rightarrow \delta_o(v_s) = 1$$

Ovvero un nodo sorgente può essere collegato ad una sola strada.
Questo non limita l'espressività del sistema in quanto è sufficiente dividere un nodo sorgente per ottenere la rappresentazione desiderata (1.7).

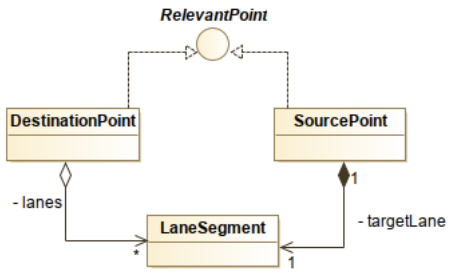

Figura 1.6: Sorgenti e destinazioni

I nodi interni alla rete invece sono realizzati da istanze della classe *CrossingPoint*, che fornisce il supporto di base alla modellazione di un incrocio

Figura 1.7: Divisione di un nodo sorgente

generico, definendo una mappa che associa ad ogni linea in entrata una o più linee in uscita: in questo modo sono coperti numerosi casi reali anche particolari, tra cui:

- Impossibilità di accedere ad una strada a partire da una certa corsia.
- Impossibilità di effettuare inversioni ad U.
- Impostare alcune tratte come più o meno trafficate.

Oltre alla versione base di *CrossingPoint*, che rappresenta un incrocio non arbitrato, è stata definita una versione *TrafficLightCrossingPoint* che invece introduce il concetto di semaforo (*TrafficLight*). Diventa possibile associare ad ogni linea in ingresso all'incrocio un semaforo che la gestisce, in particolare sono state implementate le varianti temporizzate e a sensore: quest'ultimo caso è utile quando per esempio una linea automobilistica è interrotta da una linea tramviaria. Vedi figura 1.8.

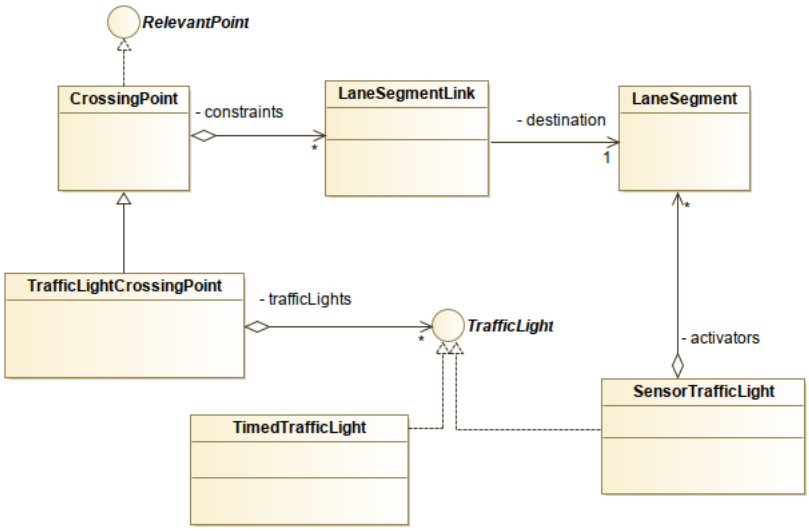

Figura 1.8: Modellazione di un incrocio

1.3.2 Configurazione

Per gli scopi a cui è destinato, il modello mostrato non prevede particolari funzionalità operative: il suo scopo è principalmente quello di *descrivere* una rete stradale. Per poter facilmente estendere il software (per esempio con un modulo user friendly per la creazione della rete o nuove analisi non ancora previste), le entità precedentemente descritte hanno la possibilità di essere configurate con una serie arbitraria di proprietà tipizzate chiave-valore (*Property*). Sarà possibile da parte dell'utente creare una serie di proprietà e configurazioni predefinite da applicare poi ad una eventuale rete. Vedi figura 1.9. Lo scopo di questo livello intermedio è quello di scaricare l'utente dalla necessità di conoscere tutte le possibili proprietà di configurazione.

La classe aggiuntiva *PropertyMetadata* potrà essere utilizzata per fissare i nomi di alcune proprietà notevoli in modo da evitare errori nel tipo o nel nome al momento della definizione.

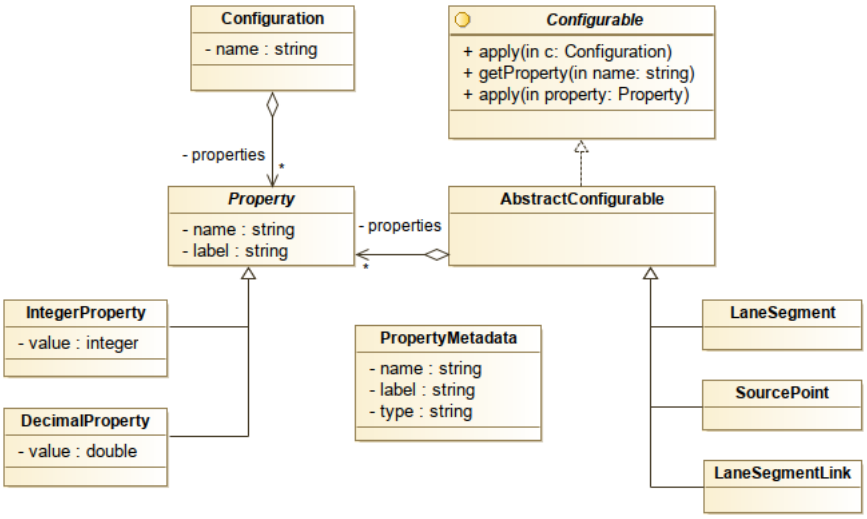

Figura 1.9: Configurazione

1.4 Schermate dell'applicazione

Di seguito saranno riportate alcune schermate di esempio.

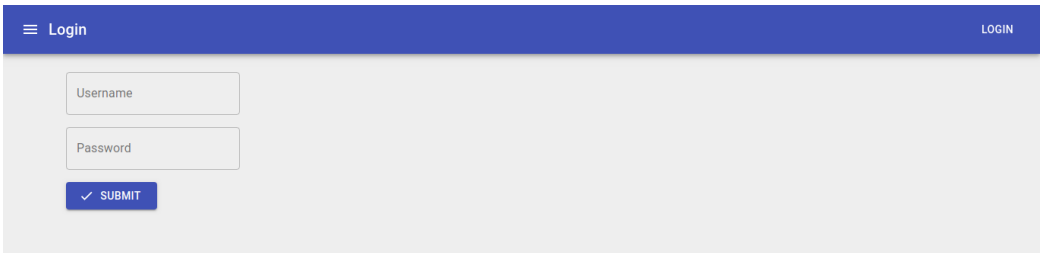

Figura 1.10: Form di login

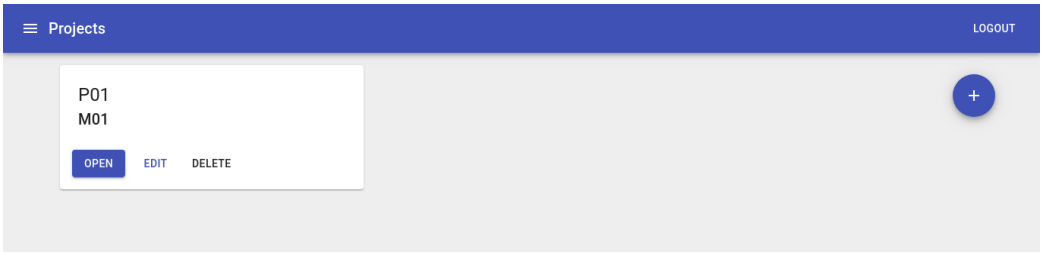

Figura 1.11: Elenco dei progetti

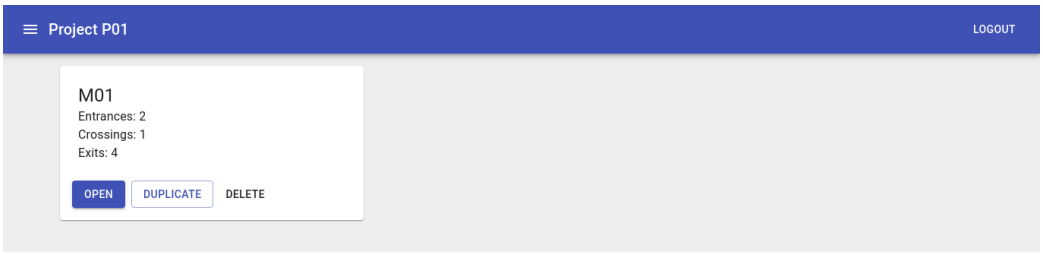

Figura 1.12: Contenuto di un progetto

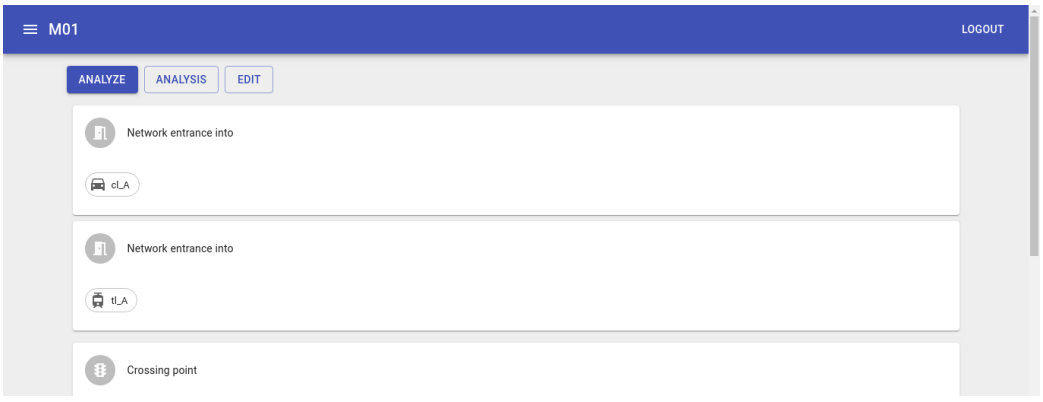

Figura 1.13: Visualizzazione di una mappa

Figura 1.14: Analisi effettuate

Capitolo 2

Architetture

2.1 Introduzione

Il cuore di un software object-oriented si trova nel modello di dominio. Questo raccoglie i concetti estratti dall'analisi dei requisiti ed è composto da una rete di oggetti dotati di attributi e metodi che rispecchiano i *dati* da memorizzare e le *operazioni* necessarie alla loro elaborazione. Questi elementi non dovrebbero avere al loro interno codice di supporto (finalizzato alla memorizzazione su un database, alla visualizzazione su interfaccia, all'invio su un canale ecc...), e in oltre questo non dovrebbe contenere frammenti di logica di dominio. Il rischio altrimenti è quello di abbassare in maniera critica la manutenibilità del codice: le varie componenti sono accoppiate tra loro, e devono evolvere necessariamente in contemporanea. Il test automatico diventa complesso, e si rischia di raggiungere presto lo stato di *big ball of mud*.[7] Scrivere codice il più possibile disaccoppiato dalle tecnologie di contorno lo rende più manutenibile e testabile, ma d'altra parte può diminuire la produttività del team di sviluppo, soprattutto quando l'applicazione da sviluppare non è di eccessiva complessità.

Nel caso del lavoro corrente l'obiettivo è simulare l'evoluzione di un software da una versione monolitica ad una a microservizi: per far questo sono state sperimentate alcune architetture riconosciute, applicando quando possibile il principio di separazione delle responsabilità, fondamentale quando si parla di microservizi.

La prima architettura presa in esame è quella a livelli.

2.2 Architettura a livelli

Questo tipo di architettura è uno degli standard più affermati nello sviluppo di applicazioni enterprise.
Il codice è organizzato a livelli sovrapposti, ognuno dei quali può sfruttare i servizi esposti **soltanto** dai quelli sottostanti. Esistono una serie di layers standard, di seguito se ne riportano alcuni dei più comuni:[1]

- Presentation: Fornisce informazioni e interpreta i comandi provenienti dall'esterno, in modo da interfacciarsi con un utente o un'altro software.
- Application: Sottile strato che dirige la logica di dominio delegando le azioni ai business objects.
- Model: Mantiene lo stato della logica di dominio e contiene le regole che la governano. É il cuore del software.
- Infrastructure: Fornisce funzionalità utili agli altri strati come persistenza, invio di messaggi, creazione interfaccia grafica ecc...

In questa visione il livello di business logic è consapevole dei livelli sottostanti, e può interagire direttamente con essi. Dovendo gestire più ad alto livello la logica di dominio, per esempio aprendo e chiudendo una transazione, anche il livello application necessita di interagire con lo strato di infrastruttura.

Le linee guida per realizzare un'applicazione seguendo questa architettura prevedono di partizionare il codice in livelli coesi e dipendenti solamente da quelli sottostanti.

Esistono varie versioni di questa architettura, in cui i livelli di contorno variano: una di queste è la *three layered architecture* (vedi figura 2.1), che prevede i livelli di *Presentation* per la gestione della UI, *Business Logic* per la logica di dominio e *Persistence* per dialogare con i database. Questi sono sufficienti per gran parte della applicazioni enterprise, in particolare se l'approccio usato è quello del monolite o del monolite a servizi.

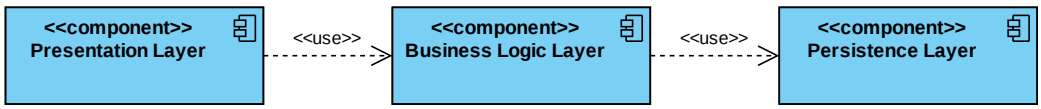

Figura 2.1: Architettura a tre livelli

Ogni livello può avere più istanze o essere ulteriormente separato, per esempio quando vi sono più interfacce grafiche verso la stessa applicazione, o più database che forniscono il servizio di persistenza.
Gli oggetti del modello di dominio devono essere, come già affermato, indipendenti dalle varie rappresentazioni, siano queste destinate ad UI, persistenza o altro. Un indice di cattiva separazione delle responsabilità può essere per esempio la necessità di duplicare codice, o di modificare la logica di dominio quando si aggiunge una nuova interfaccia all'applicazione.
La separazione in layers permette inoltre di poter effettuare il rilascio delle varie parti dell'applicazione su macchine differenti, favorendone l'evoluzione e la scalabilità.

2.3 Ports and adapters

Una delle alternative all'architettura organizzata a *layers* è l'architettura chiamata *ports and adapters* o *hexagonal architecture*.[7]

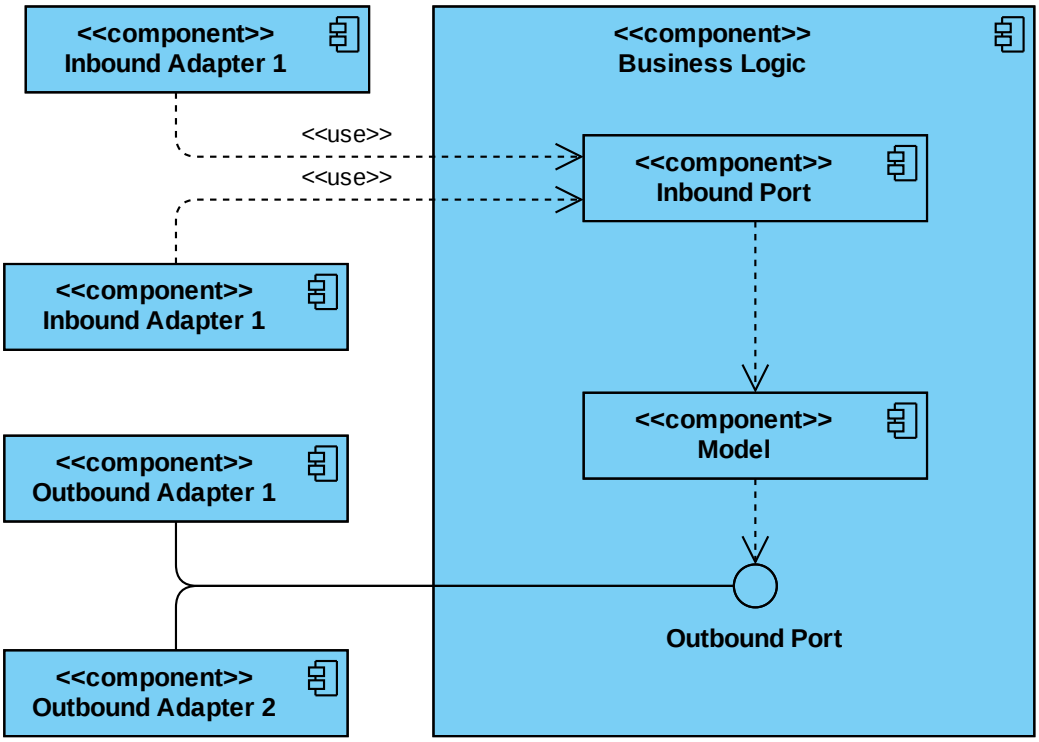

Figura 2.2: Architettura ports and adapters

Questo modo di organizzare i componenti del sistema è focalizzato sull'isolamento della logica di dominio.
Le comunicazioni di input/output con l'esterno sono guidate da interfacce **definite dal nucleo** e chiamate *porte*: queste possono essere di tipo *inbound* quando un modulo esterno ha necessità di invocare la logica di dominio, o di tipo *outbound* quando il nucleo ha la necessità di invocare servizi esterni. Ogni porta può avere più *adattatori* associati, elementi che servono per collegare realmente i componenti esterni al nucleo. La logica di dominio implementa le porte di tipo *inbound*, che saranno usate dagli adattatori in ingresso

per richiedere servizi all'applicativo. Gli adattatori in uscita implementano le porte di tipo *outbound*.

Partendo da un sistema costruito a livelli è possibile derivare l'equivalente in architettura esagonale considerando i seguenti fatti:

- Se il client di un'applicazione a livelli è un'altro software, il livello di presentazione del primo può essere molto legato al livello di persistenza del secondo: ciò che è esterno alla logica di dominio quindi puòò essere visto semplicemente come un'interfaccia esterna

- In un'architettura a livelli moderna difficilmente la progettazione inizia dal livello di persistenza: spesso questo è realizzato da classi DAO (Data Access Object) che implementano un'interfaccia definita sulla base delle necessità del livello di business logic: questo crea una dipendenza in qualche modo inversa rispetto alla filosofia dell'architettura, enfatizzando il fatto che la logica di dominio è il centro dell'applicativo.

Rifattorizzare un progetto basato su architettura three tier per renderlo compatibile con quella ports/adapters a volte potrebbe limitarsi alla riorganizzazione dei package, senza necessitare modifiche al codice: questo perché i componenti di un'architettura a livelli spesso hanno un equivalente in architettura esagonale (vedi figura).

Può capitare che data un'applicazione enterprise moderna Anche in questo caso si cerca di rendere il più possibile indipendente il nucleo di un'applicazione dalle tecnologie di contorno, come database, sistema di scambio messaggi ecc...

Convertire un'applicazione service oriented realizzata a layers in una aderente all'architettura esagonale non è un'operazione complessa: se per esempio lo stack è di tipo 3-tier con una serie di servizi REST esposti che invocano

internamente la logica di dominio ed uno strato di persistenza (es: JPA), il mapping è immediato (vedi figura 2.3).

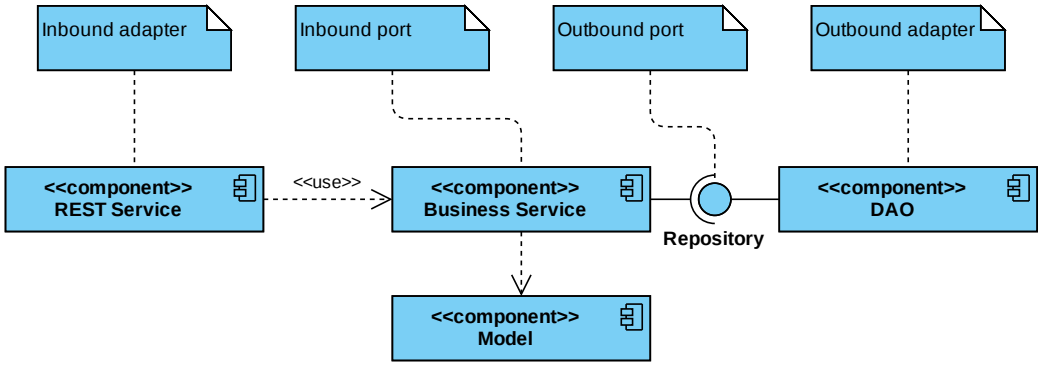

Figura 2.3: Confronto architettura a livelli e ports/adapters

I benefici di tale organizzazione diventano evidenti quando utilizzata all'interno di un software a microservizi, in quanto si è obbligati a identificare con chiarezza quali sono i confini del servizio, favorendo manutenibilità e testabilità.

2.4 Logica di dominio

Quando si parla di realizzare la logica di dominio per un'applicazione enterprise vi sono numerosi approcci che si possono adottare: di seguito riportiamo alcuni patterns notevolu evidenziando quali sono i punti di forza e le debolezze di ognuno.

2.4.1 Transaction script

Il pattern Transaction Script[3] prevede che la logica di dominio sia organizzata in base alle transazioni necessarie al sistema. Per ognuna di esse vi sarà un frammento di codice procedurale che si occupa di recuperare i dati

necessari all'elaborazione, effettuare validazioni, modifiche ed aggiornare il database. Questo approccio è particolarmente adatto per casi d'uso in cui la logica è semplice: si tratta di organizzare correttamente il codice in moduli evitando duplicazione. In riferimento alla figura 2.3, i componenti *Business Service* sono in effetti le classi *Transaction Script*, prive di stato e ricchie di logica; i componenti *Model* sono classi di modello che contengono lo stato vero e proprio, ma povere di comportamento (es. POJO in Java).
Uno dei casi in cui questo approccio può essere utile, è quando si utilizza un framework come JPA per la persistenza: si ha infatti il vantaggio di ridurre drasticamente il numero di righe di codice di tale livello, accettando il compromesso di "sporcare" le classi di modello con alcune annotazioni accoppiate ad una tecnologia. Rendendo la separazione tra livelli (o tra nucleo e porte) più labile, questo pattern è adatto quando la logica di dominio non è particolarmente complessa e si prevede che non evolverà molto.

2.4.2 Domain model

Il pattern Domain Model,[3] al contrario di Transaction Script, organizza la logica di dominio all'interno degli oggetti del modello: *Model* in figura 2.3. Le classi della categoria *Business Service* sono presenti e presentano come in *Transaction Script* un metodo per ogni richiesta/transazione, ma in questo caso la maggior parte del comportamento viene delegato.

In un'applicazione Object Oriented, gli oggetti cercano di modellare dati e comportamenti caratteristici dell'area di business di cui tratta il software. Quando si ha a che fare con un modello ad oggetti ricco, forzarlo a rispecchiare semplicemente tabelle di un database vanifica i benefici della coesione tra dati ed operazioni forniti dalla OOP. Senza contare il fatto che non esiste sempre una corrispondenza esatta tra le associazioni possibili tra oggetti e

quelle tra entità memorizzate su database (basti pensare al tema dell'Object Relational Mapping o ai Design Patterns[4]).
Utilizzare un vero e proprio modello ad oggetti permette di avere codice più comprensibile e manutenibile, preferendo numerose piccole classi ognuna con le proprie responsabilità.

Quando si parla di architettura a microservizi anche quest'ultimo approccio può mostrare alcuni punti di debolezza. Può essere utile effettuare un ulteriore raffinamento, arrivando al concetto di Domain-driven design.

2.4.3 Domain-driven design

Questo raffinamento dello sviluppo orientato agli oggetti[1][9] è stato pensato per applicazioni dotate di logica di business complessa.

DDD prevede alcuni patterns utilizzabili come elementi base per costruire un modello di dominio. Di seguito i principali:

- *Entity:* Oggetto che ha un'identità persistente identificate da un ID. Sono le classi che in JPA sono annotate come *@Entity*.
- *Value object:* Collezione di valori priva di un'identità persistente. Non possiedono un'ID, quindi due istanze diverse con lo stesso stato interno possono essere usate in modo intercambiabile.
- *Factory:* Oggetto o metodo che si occupa di creare oggetti complessi.
- *Repository:* Oggetto che permette l'accesso alle entità persistenti e nasconde i meccanismi di accesso al database.
- *Service:* Oggetto che implementa la logica di business che non è interna alle entities e ai value objects.

Un altro concetto importante in questo contesto è quello di *Aggregato*.

2.4.4 Aggregati e microservizi

In un'architettura a microservizi ogni servizio deve necessariamente avere il proprio modello di dominio. Un tema interessante che verrà approfondito in questo paragrafo è quello del partizionamento di un modello di business in *sottodomini* utilizzando il concetto di *Business Context*.

2.4.5 Architettura del software

Il sistema è stato realizzato separando lo strato di UI da quello di business logic, realizzando secondo i principi dell'architettura *ports/adapters*.
In particolare sono esposti verso l'esterno (inbound) una serie di servizi REST necessari all'interazione, ed è utilizzato un servizio di persistenza relazionale (outbound) necessario alla memorizzazione di informazioni come il profilo degli utenti, i progetti e le analisi effettuate. In figura 2.4 è mostrato lo schema generale.

I requisiti permettono di dividere naturalmente i servizi, le porte e gli adattatori in tre categorie:

- Gestione degli utenti;
- Gestione delle configurazioni;
- Gestione dei progetti;

Un altro modulo piuttosto indipendente è quello relativo alla gestione delle analisi, ma essendo queste legate strettamente ad un progetto, saranno considerate parte dello stesso servizio.

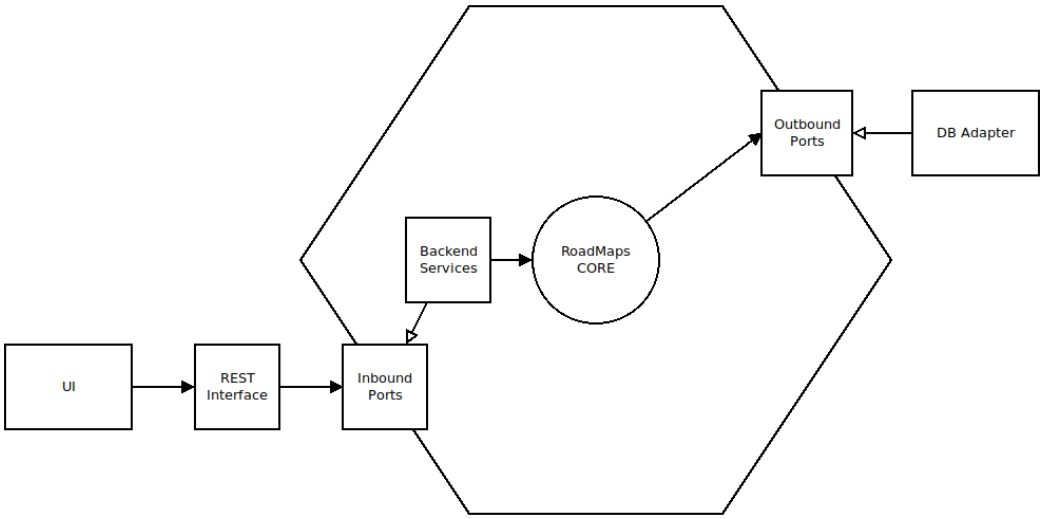

Figura 2.4: Architettura generale

Ognuno di questi moduli darà luogo ad un endpoint REST corrispondente ad uno o più casi d'uso, ad almeno un'entità principale e ad almeno una tabella su RDB. Di seguito verrà riportata la descrizione di ognuno di questi blocchi partendo dal centro del modello di dominio e spostandosi all'esterno.

Gestione degli utenti

Il frammento di modello di dominio relativo alla gestione utenti è quello in figura 2.5.

Dallo schema è possibile individuare quali siano le varie componenti associabili ai concetti dell'architettura esagonale:

- UserService: Rappresenta una *inbound port* che espone all'esterno le funzionalità relative ad un utente;
- UserServiceImpl: Implementa l'interfaccia *UserService*, ed è quindi un *inbound adapter*;

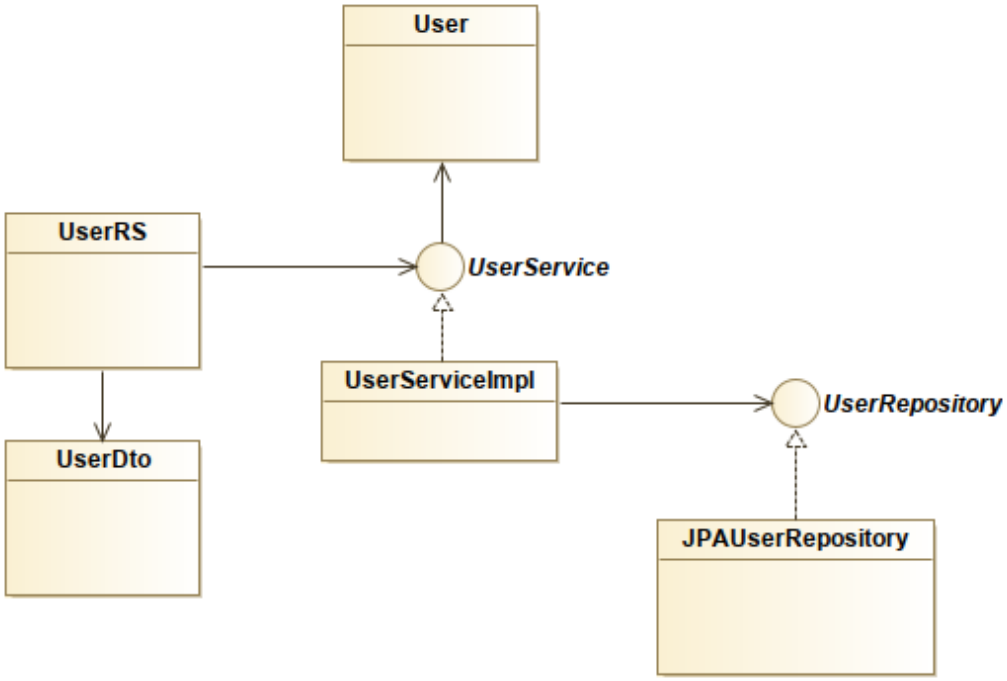

Figura 2.5: Diagramma delle classi relative alla gestione degli utenti

- UserRepository: É una *outbound port* necessaria per fornire alla logica di dominio le funzionalità di persistenza;
- JPAUserRepository: Implementazione di *UserRepository* e quindi *outbound adapter*, dipendente da una tecnologia specifica (Database relazionali + JPA), fornisce il collegamento con un RDBMS.
- UserRS: É un controller REST che espone verso il sottosistema UI i dati e le funzionalità di backend.

Gestione delle configurazioni

Il frammento di modello di dominio relativo alla gestione delle configurazioni è quello in figura 2.6.

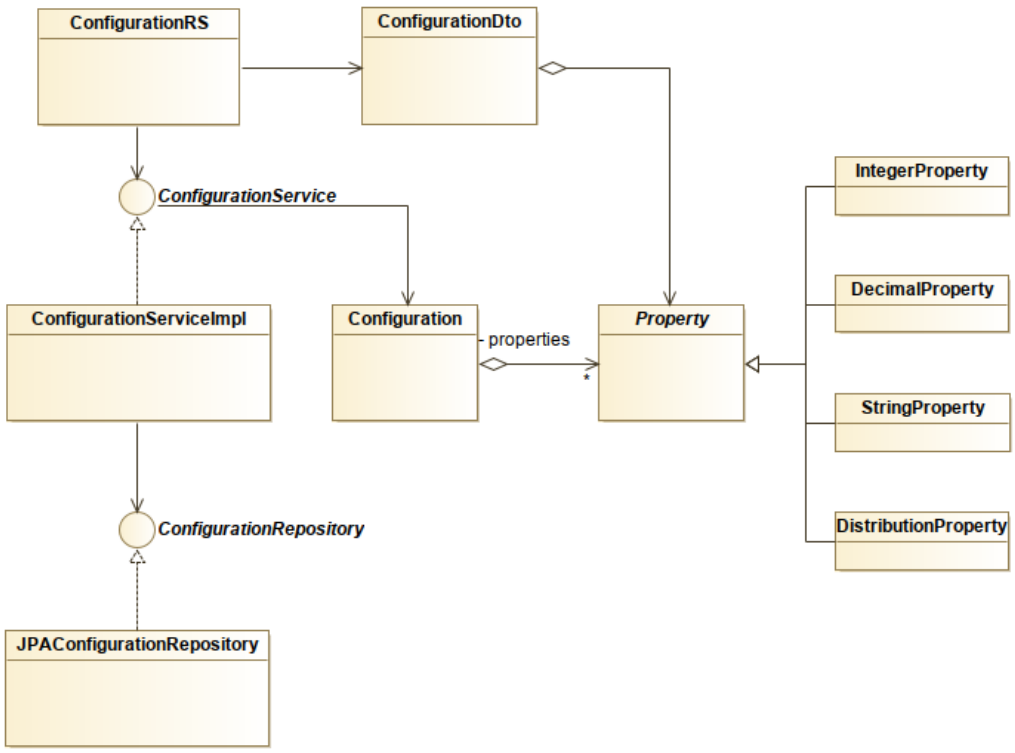

Figura 2.6: Diagramma delle classi relative alla gestione delle configurazioni

La struttura generale è equivalente a quella della sezione utenti: si hanno due interfacce *ConfigurationService* e *ConfigurationRepository* corrispondenti alle porte *inbound* e *outbound*, con i relativi adattatori *ConfigurationServiceImpl* e *JPAConfigurationRepository*.
Vi è sempre un controller REST per ricevere comandi dall'esterno del sistema, utilizzando in questo caso direttamente alcune entità del modello di dominio.

Gestione dei progetti

Il frammento di modello di dominio relativo alla gestione utenti è quello in figura 2.7.

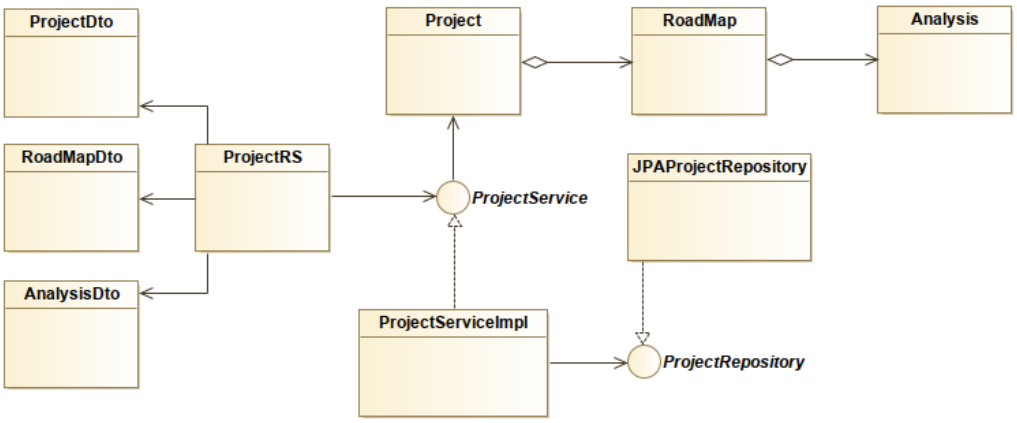

Figura 2.7: Diagramma delle classi relative alla gestione dei progetti

Anche in questo caso vi sono le stesse porte e adattatori più il controller REST relativo.

Capitolo 3

Tecnologie

TODO

3.0.1 UI

La parte di interfaccia è stata realizzata nei linguaggi HTML/CSS/JS, utilizzando in particolare il framework *React*[2] e la libreria di componenti per quest'ultimo *Material-UI*,[6] che implementano le linee guide di Material Design.[5]

Bibliografia

[1] Eric Evans. *Domain-Driven Design.* Addison-Wesley Professional, 2003.

[2] Facebook. React.

[3] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, 2002.

[4] Johnson Vlissides Gamma, Helm. *Design Patterns.* Pearson, 2002.

[5] Google. Material design.

[6] Material-UI. Material-ui.

[7] Irakli Nadareishvili. *Microservice Architecture.* O'Reilly, 2016.

[8] Roberto Grossi Pierluigi Crescenzi, Giorgio Gambosi. *Strutture di dati e algoritmi.* Pearson, 2006.

[9] Chris Richardson. *Microservices Patterns.* Manning, 2018.