Contents

[Abstract 3](#_Toc468638614)

[Introduzione 3](#_Toc468638615)

[1 MyData, Big Data 4](#_Toc468638616)

[2 Smart Mobility 6](#_Toc468638617)

[2.1 Mobility as a Service 6](#_Toc468638618)

[2.1.1 Mobility Profile e Journey Planner 7](#_Toc468638619)

[2.2 Smart Mobility for All 7](#_Toc468638620)

[2 ½ GDPR?? 8](#_Toc468638621)

[3 Architettura MyData 8](#_Toc468638622)

[3.1 Entità fondamentali 8](#_Toc468638623)

[3.2 Service Registry, Service Linking 8](#_Toc468638624)

[3.2.1 OAuth 2.0 9](#_Toc468638625)

[3.3 Autorizzazioni e Consent 10](#_Toc468638626)

[3.3.1 Kantara Consent Notice Receipt Specification 11](#_Toc468638627)

[3.3.2 User Managed Access 12](#_Toc468638628)

[3.4 Personal Data Storage 13](#_Toc468638629)

[4 Analisi e Design 15](#_Toc468638630)

[4.1 Accounting e Servizi 15](#_Toc468638631)

[4.2 Operatore MyData 16](#_Toc468638632)

[4.3 Service Registry, Service Linking 16](#_Toc468638633)

[4.4 Consent 17](#_Toc468638634)

[Granularità di un Consent 17](#_Toc468638635)

[4.5 Personal Data Storage 18](#_Toc468638636)

[4.6 Rappresentazione di dati non noti a priori 19](#_Toc468638637)

[5 Progettazione 21](#_Toc468638638)

[5.1 Flusso del programma 21](#_Toc468638639)

[5.2 Accounting 21](#_Toc468638640)

[5.2.1 IUser, MyDataUser 22](#_Toc468638641)

[5.2.2 IAccount, Account 23](#_Toc468638642)

[5.3 SecurityManager 24](#_Toc468638643)

[5.4 IMyData, MyData 24](#_Toc468638644)

[5.6 Autorizzazioni e Consent 26](#_Toc468638645)

[5.6.1 ConsentManager, ConsentStatus 26](#_Toc468638646)

[5.6.2 ServiceConsent, DataConsent 28](#_Toc468638647)

[rappresentazione dei dati 29](#_Toc468638648)

[5.7 PersonalDataVault 32](#_Toc468638649)

[5.8 Metadata 33](#_Toc468638650)

[DataSet 34](#_Toc468638651)

[5.9 IService, AbstractService, MostLikelyNextTrip 36](#_Toc468638652)

[5.10 ServiceRegistry 38](#_Toc468638653)

[5.11 Uso delle eccezioni 39](#_Toc468638654)

[Graphical User Interface 40](#_Toc468638655)

[Conclusioni e sviluppi futuri 42](#_Toc468638656)

# Abstract

# Introduzione

Si provi a pensare alla quantità di informazioni che uno smartphone raccoglie ogni giorno sul suo proprietario. Dati di localizzazione insieme a statistiche di utilizzo possono chiarire in quale parte del mondo ci troviamo e, ad esempio, che tipologia di occupazione abbiamo. Applicazioni di social networking tengono conto di etnia, orientamento politico, sessuale, religioso, insieme alla rete di amici e conoscenti acquisita dalla lista dei contatti. Calendario ed email possono aiutare in caso di impegni di lavoro; ci sono poi app per il monitoraggio dell’attività fisica che contengono grandi quantità di informazioni biometriche. Per concludere non si può dimenticare la presenza di app di e-commerce, gestione di account bancari e genericamente finanziarie. Il tacito accordo che porta avanti questa relazione fra gli utenti e le aziende che erogano servizi accessibili via Internet consiste nella rinuncia da parte dell’utilizzatore del controllo sui propri dati e sull’uso che ne viene fatto in cambio di servizi gratuiti o a basso prezzo. La quantità di informazioni raccolte non si limita a ciò che l’utente sceglie di condividere (foto, email, transazioni online), ma comprende anche dati osservati (abitudini di navigazione, dati di geolocalizzazione) e deduzioni (*targeted advertising*, previsioni sul flusso del traffico).

Tuttavia, in questo contesto è in crescita una contro-tendenza che vede un numero sempre più crescente di utenti insoddisfatti dell’uso dei propri dati personali \cite \cite. L’esplorazione in ambito scientifico di soluzioni che offrono una maggiore tutela della privacy si affianca ad una crescente attenzione da parte dell’opinione pubblica e delle istituzioni (adozione del regolamento generale sulla protezione dei dati da parte dell’Unione Europea \cite{}).

In questo contesto si sviluppa il modello *MyData*, fondato sull’idea secondo cui ogni utente ha il controllo sui propri dati personali, ed è a conoscenza dell’utilizzo che ne viene fatto, in modo trasparente. D’altra parte, l’architettura *MyData* vuole offrire al mercato e alle imprese un contesto di sviluppo di applicazioni software in cui vengono rispettate le leggi in materia di protezione dei dati sensibili, favorendo l’interoperabilità fra di esse.

L’approccio *MyData*, seppur innovativo, non è senza precedenti. Molti dei suoi concetti fondamentali sono anche alla base dei *personal cloud*, come ad esempio la possibilità di contenere in modo sicuro i dati personali dell’utente e offrire interoperabilità fra diversi servizi, utilizzando concretamente i dati memorizzati.

Questa tesi si propone di studiare i concetti fondamentali alla base del modello *MyData*, e implementare un prototipo di gestore di dati personali che ne rispetti le specifiche.

A titolo di esempio, si è scelto di utilizzare come servizio “consumatore” di dati personali un sistema di previsione del viaggio più probabile.

L’obiettivo è quello di realizzare un sistema che permetta all’utente di monitorare l’utilizzo dei suoi dati personali in tempo reale ed in modo trasparente, attraverso l’implementazione di una politica di controllo degli accessi. Inoltre, il gestore opera in un contesto in cui il rispetto delle politiche di sicurezza è fondamentale: anche se la realizzazione di un sistema sicuro non rientra negli obiettivi del lavoro di tesi, sono presenti alcuni accorgimenti che puntano in questa direzione.

# 1 MyData, Big Data

Attualmente, grandi quantità di dati vengono continuamente raccolti senza utilizzarne pienamente la ricchezza dell’informazione. Il loro percorso e le operazioni di processamento a cui vengono sottoposti sono sempre più numerose e cresce la difficoltà nel conservarne le tracce. Inoltre, l’utilizzo di software proprietari limita la possibilità di analisi dei dati a causa della scarsa interoperabilità fra le soluzioni adottate. Infine, va anche considerato l’impatto che tale modus operandi ha sugli utilizzatori finali: secondo un sondaggio riportato dal World Economic Forum, “Fully 78% of consumers think it is hard to trust companies when it comes to use of their personal data” (Orange, The Future of Digital Trust, 2014).

Il progetto MyData nasce in Finlandia, all’università di Aalto, dall’idea di rafforzare i diritti digitali dell’individuo, e di offrire un modello per la gestione di dati personali che aderisca alla stringente legislazione europea sulla loro tutela. Inoltre, *MyData* cerca di trovare una risposta a questi temi attraverso un cambio di paradigma che ponga l’utente al centro, attribuendogli la capacità di gestire i propri dati personali e di comprendere l’uso che ne viene fatto. Ulteriori benefici includerebbero l’aumento della qualità del servizio ricevuto, grazie ad una più consapevole condivisione (poiché informata) dei propri dati. Scenari futuri includono la possibilità di fornire conoscenze più approfondite del proprio comportamento da utenti (*self tracking*), anche attraverso la ricezione di un compenso per il processamento dei propri dati (*data monetization*).

Vi sono in aggiunta benefici anche in termini di sviluppo software. L’approccio a livello infrastrutturale permetterebbe l’indipendenza da specifici settori (sanità e salute, finanza, ecc.), favorendo una completa portabilità dei dati. Il rispetto e l’aderenza alle leggi verrebbe realizzato dall’infrastruttura stessa, consentendo alle aziende di sviluppare i servizi informatici in maniera meno vincolata, acquisendo al contempo la fiducia del cliente grazie alla trasparenza e all’affidabilità garantite da *MyData*.

Nel complesso, il paradigma *MyData* si contrappone all’attuale standard, Big Data, proponendo nuove soluzioni che non trascurino la realtà del mercato in cui il software viene usato.

# 2 Smart Mobility

## 2.1 Mobility as a Service

*MyData*, grazie al coinvolgimento del Ministero dei Trasporti, ha avuto un primo banco di prova all’interno del concetto di Mobility As A Service. Il progetto finlandese ha ricevuto un notevole slancio grazie al lavoro di tesi svolto da Sonja Heikkilä, nella quale l’autrice propone un nuovo concetto di mobilità per la città di Helsinki per rispondere alle sfide sempre più frequenti poste al settore dei trasporti. Le principali difficoltà risiedono nell’inaffidabilità dei mezzi pubblici, unico modo per raggiungere determinate mete, comparata con la difficoltà ad usare l’auto di proprietà, causa mancanza parcheggi o rischio di ingorghi durante il viaggio.

Il paradigma MaaS descrive un nuovo utilizzo delle tecnologie applicato ai mezzi di trasporto, che propone il passaggio dall’auto di proprietà a mezzi di trasporto condivisi. Non si tratta solo di una migliore gestione dei mezzi pubblici, che proponga ad esempio soluzioni di pagamento online o potenziamento delle corse in base alla richiesta: Mobility As A Service si applica anche ai taxi, biciclette o sistemi di car sharing.

Questo cambiamento consentirebbe di aumentare l’efficienza nell’uso dei mezzi di trasporto, eliminando gli sprechi che inevitabilmente derivano dal possesso di un autoveicolo e dal suo inutilizzo. L’accesso del privato ai mezzi di trasporto avverrebbe, in questa nuova ottica, attraverso un software in grado di calcolare una ottimizzazione per i mezzi condivisi, ad esempio raggruppando gli utenti per fasce orarie, tratte comuni e generiche preferenze. Grazie inoltre ad una gestione comune dei mezzi di trasporto diversificati fra loro, la pianificazione del viaggio può comprendere tratte percorse in modalità diverse (ad esempio automobile e bicicletta).

Da questa proposta è nata MaaS Global, una startup (?) finlandese che, durante l’autunno 2016, rilascerà l’app per smartphone “Whim”, con la quale sarà possibile muoversi all’interno della città di Helsinki secondo il paradigma Mobility As A Service. Inizialmente, si potrà fare uso di mezzi di trasporto quali il trasporto pubblico urbano, i taxi e le auto a noleggio, ma saranno presto integrati anche i servizi di bike e car sharing.

### 2.1.1 Mobility Profile e Journey Planner

Come *proof of concept*, sono state realizzate dall’università di Aalto due applicazioni, rilasciate su piattaforma Android e iOS, Mobility Profile e Journey Planner.

Mobility Profile raccoglie i dati personali dell’utente, li mantiene in un database relazionale e svolge le operazioni di calcolo del prossimo viaggio più probabile. Questa applicazione funziona come base di appoggio per Journey Planner, che riceve i suggerimenti calcolati e restituisce un feedback al processo sottostante, in modo da migliorarne l’accuratezza. Le API di comunicazione fra le due sono:

requestSuggestions(), requestTransportModePreferences(), e sendSearchedRoute(Place startLocation, Place destination).

In questo esempio, Mobility Profile non rispetta pienamente le specifiche dettate per *MyData*, ma è comunque possibile riconoscerne alcune caratteristiche all’interno del progetto. Ne sono esempio la richiesta esplicita di un permesso (revocabile in ogni momento) per l’utilizzo di dati personali, come gli impegni del calendario e lo storico delle posizioni GPS, e anche lo sviluppo di una applicazione separata per l’utilizzo dei risultati (Journey Planner) rispetto a quella che raccoglie i dati e li processa.

## 2.2 Smart Mobility for All

Rimanendo sul filone della mobilità intelligente, il progetto Smart Mobility for All si sviluppa in modo indipendente da MaaS.

L’idea centrale su cui si fonda la piattaforma SMAll è la costruzione di un insieme di servizi di mobilità, potenzialmente anche per mezzi di trasporto molto diversi fra loro, che gestisca l’intero ciclo di vita di un viaggio, dalla prenotazione all’effettivo spostamento e all’arrivo a destinazione. Si tratta di Smart Mobility poiché attraverso la raccolta e l’analisi dei dati di un utente è possibile fare inferenze sugli spostamenti futuri, proponendo anche l’acquisto dei titoli di viaggio più adatti alle abitudini e alle necessità dell’individuo.

Ancora in via di sviluppo, si basa sul concetto di modularità che punta a fare di ogni servizio un modulo separato e indipendente. In questo modo si applica efficacemente il principio di suddivisione delle responsabilità e migliora la manutenibilità sia del sistema nel complesso che dei singoli servizi.

All’interno dei vari componenti dell’infrastruttura si potrebbe prevedere un gestore di dati personali del quale si porta, in questa tesi, un esempio semplificato. In questo contesto risulta evidente l’importanza della gestione dei dati personali, non solo in termini di protezione della privacy ma anche con uno sguardo all’interoperabilità e al corretto funzionamento di un insieme di servizi eterogenei.

# 2 ½ GDPR??

# 3 Architettura MyData

Al fine di comprendere appieno le scelte effettuate all’interno del progetto, si evidenziano brevemente le componenti e la struttura del modello *MyData*.

## 3.1 Entità fondamentali

L’architettura di *MyData* si costruisce su quattro componenti base: l’utente finale, detto anche *Account Owner*, l’operatore *MyData*, o *Operator*, e due generiche entità che, da una parte, “producono” dati, e, dall’altra, li “consumano”. Esse sono definite rispettivamente *Source* e *Sink*. Mentre i ruoli di *Account Owner* e di *Operator* sono generalmente statici, quelli di *Source* e *Sink* sono fortemente variabili nel tempo e si possono applicare anche ad entità molto diverse fra loro, poiché definiti con un alto livello di astrazione. Convenzionalmente, si può identificare un servizio come “consumatore” di dati, mentre l’account dell’utente può essere un “produttore” di dati personali. In *MyData*, è possibile altresì che un servizio occupi entrambi i ruoli, o anche che l’*Operator* stesso rientri in questa classificazione quando si trova a compiere operazioni sui dati.

Il ruolo di *Operator* comprende operazioni di vario genere fra i quali vi sono la gestione degli utenti, dei servizi e delle interazioni che avvengono fra le due parti. Esso si occupa anche di gestire l’Audit Log di tutte le operazioni che coinvolgono tali interazioni.

## 3.2 Service Registry, Service Linking

Con *Service Registry* viene indicata quella parte dell’*Operator* che contiene un database di tutti i servizi registrati presso quell’operatore. Esso contiene anche la funzionalità di *Service Discovery*, utilizzata dagli utenti per trovare nuovi servizi da utilizzare. In particolare, gli utenti possono venire a conoscenza di un nuovo servizio tramite un suggerimento, calcolato in base a corrispondenze fra caratteristiche dell’utente e del servizio, oppure tramite ricerca diretta.

Ogni nuovo servizio che vuole essere utilizzabile all’interno dell’architettura MyData deve quindi sottoporsi ad una procedura di registrazione al termine della quale, in caso di successo, viene inserito all’interno del *Registry*.

Durante questa procedura, il servizio deve fornire almeno una descrizione del suo comportamento in formato machine-readable e human-readable: la prima permette a procedure automatiche una elaborazione corretta di suggerimenti, la seconda è rivolta direttamente all’utente finale.

L’iscrizione di un utente presso un servizio avviene tramite un processo chiamato *Service Linking*, in cui l’Operatore MyData si occupa di realizzare una identificazione mutua delle parti. Tutti i token e le firme digitali scambiate durante il procedimento sono espresse in notazione JSON.

Al termine del *Service Linking* viene prodotto un *Service Link Record*, necessario per ogni futura interazione fra l’utente ed il servizio.

### 3.2.1 OAuth 2.0

OAuth è un protocollo per il controllo dei flussi di autorizzazioni fra mobile applications, desktop applications, dispositivi mobili, ecc. \cite{}. Se ne dà un breve accenno in questa sede in quanto esso viene preso come esempio di implementazione di Service Linking all’interno delle specifiche di MyData \cite{mydatastack}.

Le entità coinvolte nel protocollo di autorizzazione le seguenti:

\begin{itemize}

\item Un’applicazione Client, che richiede l’accesso ad un account utente;

\item Un Resource Owner, spesso coincidente con l’utente finale, che accorda o nega l’accesso ad una porzione del suo account;

\item Un Resource Server dove sono memorizzati gli account, che espone determinate API per l’accesso e discrimina le richieste legittime in base ad access token;

\item Un Authorization Server che emette gli access token dopo aver provato l’identità dell’utente finale, e solo in caso quest’ultimo abbia acconsentito all’utilizzo dei suoi dati. Può coincidere con il Resource Server.

\end{itemize}

È facilmente riscontrabile un’analogia con le entità descritte all’interno dell’architettura MyData, ad esempio fra Sink e applicazione Client, fra Source e Resource Server, fra Account Owner e Resource Owner, e infine fra l’Operatore MyData e Autorization Server.

In figura \ref{} è mostrato il flusso di autorizzazioni implementato dal protocollo OAuth 2.0, dal quale è stato preso spunto per la definizione di Service Linking e delle interazioni in MyData. Le analogie possono essere riassunte all’interno dei seguenti punti:

\begin{itemize}

\item I punti (A) e (B) corrispondono alla registrazione di un utente presso un servizio; questo stato viene spesso indicato come precondizione per il Service Linking all’interno della documentazione MyData.

\item I punti (C ) e (D) riassumono i passi che l’Operatore compie per identificare il servizio, emettere ID surrogati e token key e svolgere le operazioni di autenticazione e firma da parte di Account Owner e del servizio stesso.

\item i punti (E) e (F) descrivono come avviene effettivamente il recupero dei dati una volta ottenuta l’autorizzazione necessaria. In MyData ciò corrisponde alla “connessione dati” che si instaura fra Source e Sink.

Vista la forte analogia con MyData, in fase di Progettazione ho scelto di prendere spunto dal protocollo OAuth al momento di definire un algoritmo per l’autenticazione e le relazioni fra le parti.

## 3.3 Autorizzazioni e Consent

Come specificato dal GDPR, ogni operazione svolta sui dati personali di un utente deve essere stata autorizzata dallo stesso tramite un permesso che acquista in questo contesto una valenza legale.

La funzione di un permesso, o *Consent*,è particolarmente rilevante: definisce quali dati possono essere utilizzati e in che modalità, e identifica le entità *Source* e *Sink* fra le quali avviene lo scambio. Il processo di *Service Linking* deve essere stato completato con successo affinché sia possibile fare richiesta di autorizzazione, e ciò viene verificato tramite ispezione del *Service Link Record*.

Il ruolo dell’Operator in questa situazione è quello di recuperare le informazioni corrispondenti al servizio presso il *Service Registry*: queste vengono presentate all’utente che decide se acconsentire o meno al processamento di un determinato insieme di dati da parte di un servizio specificato. È possibile per l’utente chiedere una ridefinizione delle richieste del servizio, ma non al di sotto del limite previsto per un corretto svolgimento del servizio stesso.

Nel caso la procedura di autorizzazione si concluda con successo, viene prodotto un *Consent Record* che contiene tutte le specifiche negoziate fra le parti insieme agli identificatori dell’utente e del servizio. Esso viene memorizzato all’interno dell’account, ma è possibile che il servizio o l’Operatore ne facciano richiesta successivamente.

Per dare la possibilità all’utente di ritirare il permesso accordato, un *Consent* ha tre stati possibili: *Active*, *Disabled* e *Withdrawn*. Il primo è lo stato standard di funzionamento, in cui l’accesso ai dati è consentito; si hanno poi gli stati “disabilitato” e “ritirato”, in cui l’accesso è impedito. Nel caso in cui il permesso sia stato ritirato è necessario provvedere all’emissione di una nuova autorizzazione, mentre in caso di stato disabilitato, è possibile attuare un cambiamento di stato, riportandolo al valore attivo.

Questo protocollo di autorizzazione all’utilizzo dei dati rispetta quindi quanto affermato dal GDPR, poiché il permesso viene dato volontariamente e in modo chiaro. Non è ambiguo, è informato, grazie alla specifica delle risorse necessarie, ed è possibile ritirarlo in ogni momento.

### 3.3.1 Kantara Consent Notice Receipt Specification

All’interno della documentazione di MyData \cite{} viene indicato il documento di specifica Consent Notice Recipt Specification (CNRS) elaborato da Kantara \cite{} come esempio di Consent Record. Il contesto di sviluppo alla base di questo documento è simile a quello di MyData, descritto nei capitoli \ref{capitolo1} e \ref{capitolo2}, e consiste nell’offrire agli utenti la possibilità di venire a conoscenza dell’uso che viene fatto dei propri dati e di poter intervenire attivamente nella sua gestione.

Pertanto, l’obiettivo del CNRS è quello di definire uno standard di implementazione per record di autorizzazioni e transazioni di dati, al fine di supportare l’interoperabilità fra sistemi diversi, offrire una prova credibile di autorizzazione ricevuta e dare il via a “buone pratiche” e consuetudini coerenti con il rispetto della privacy e la tracciabilità delle trasmissioni di dati.

All’interno del documento viene dettagliata la terminologia specifica del settore relativo a privacy e autorizzazioni, ad esempio mediante la definizione del termine “Consent”, che riporto di seguito perché particolarmente rilevante all’interno del contesto:

\begin {quote}

A Personally identifiable information (PII) Principal’s freely given, specific and informed agreement to the processing of their PII.

\end{quote}

Altri esempi comprendono la definizione di Personally Identifiable Information e Consent Recipt.

Successivamente, viene descritta la struttura dati proposta come standard di Consent Receipt, dettagliando ogni campo presente al suo interno. Ho preso spunto da questo documento al momento di realizzare i Consent all’interno del gestore di dati personali (sottosezione \ref{}).

### 3.3.2 User Managed Access

\\figure

User Managed Access \cite{} è un protocollo basato su OAuth (del quale si è parlato in \ref{}) il cui scopo è definire l’accesso a risorse protette (potenzialmente disponibili su infrastruttura distribuita) da parte di applicazioni client. Le modalità di accesso alle risorse vengono definite dall’utente finale e messe in atto da un’autorità centrale di autorizzazione. Anche il protocollo UMA, come il Consent Receipt Specification sopra citato, viene indicato all’interno della documentazione MyData come “Key Related Standard”.

Il protocollo UMA si articola in tre momenti principali, dai quali si è preso spunto nell’implementazione del flusso del programma all’interno del gestore di dati personali. Per agevolare la descrizione del protocollo, si definiscono i termini Authorization Server, per descrivere il server che si occupa di policy enforcement, e Resource Server, che indica il server presso cui vengono depositate le risorse da proteggere.

Come prima operazione, l’utente finale deposita i suoi dati presso un Authorization Server, definendo in modo sicuro le regole per la gestione dell’accesso. Questo è regolato per mezzo delle API esposte dall’Authorization Server, che fanno uso di OAuth (o sue implementazioni).

In un secondo momento, una applicazione client fa richiede un set di risorse ad un Resource Server protetto mediante protocollo UMA tramite API OAuth.

Durante il protocollo vi è lo scambio e l’emissione di tre tipi di token (Protection API Token, Requesting Party Token, Authorization API Token), utilizzati per provare l’affidabilità, l’identità e l’autorizzazione garantita al servizio per l’accesso alle risorse.

## 3.4 Personal Data Storage

Nonostante non venga dato un peso rilevante a questa componente all’interno dei documenti di *MyData*, non è possibile prescindere dall’esistenza di un database che mantenga tutti i dati relativi ad un *Account Owner*. Non si tratta infatti solo di dati personali, ma di ogni dato utilizzato da un generico servizio o ad esempio inserito volontariamente dall’utente.

Non è specificato se il Personal Data Storage (spesso chiamato anche Personal Data Vault) faccia parte dell’ecosistema dell’operatore: ciò è possibile, ma non sono da escludere implementazioni alternative che prevedono il salvataggio delle informazioni presso il dispositivo dell’utente.

L’accesso ai dati contenuti all’interno del Personal Data Storage è possibile solo in presenza di un adeguato *Consent Record*, ma le modalità di accesso non vengono regolamentate in maniera dettagliata. Si parla infatti genericamente di “Data API”, con le quali un *Sink* ottiene (in caso di richiesta legittima) un determinato *Resource Set* da un *Source*.

Poiché l’applicazione Mobility Profile è stata sviluppata in un secondo momento rispetto alla prima pubblicazione delle specifiche tecniche suddette, si è provveduto all’interno della relativa documentazione \cite{} ad aggiungere una breve menzione del Personal Data Storage, spiegando il suo ruolo all’interno dell’applicazione.

Dall’analisi del codice disponibile sul repository github \cite{} è emerso che i dati personali vengono mantenuti all’interno del dispositivo mobile e gestiti attraverso un database relazionale <https://github.com/satyan/sugar> . Questa implementazione differisce leggermente da quella proposta inizialmente, in quanto la computazione del servizio avviene localmente ai dati invece che localmente al servizio. All’interno del progetto del gestore di dati personali ho scelto di restare aderente alle prime specifiche di MyData \cite{}. Questa problematica viene analizzata più in dettaglio prima in fase di Analisi (sezione ) e successivamente in fase di Progettazione (sottosezione ).

# 4 Analisi e Design

Il progetto del gestore di dati personali è cominciato con un’analisi delle specifiche *MyData* all’interno di uno studio di fattibilità. Le soluzioni adottate sono generalmente di portata “enterprise” e come tali inadatte ad un progetto di tesi. Per questo motivo ho cercato di seguire, durante il progetto, le stesse linee guida e i principi posti alla base di *MyData* preferendo, quando possibile, una implementazione più semplice e adatta al contesto.

Da quanto emerso nei paragrafi precedenti, si rende necessaria la presenza dei componenti di alto livello per la realizzazione del gestore di dati personali. Questi sono:

* Un utente finale che disponga di un account presso l’Operatore *MyData* e di un account presso il servizio di calcolo del prossimo viaggio più probabile;
* Il servizio che realizzi la logica di business, ad esempio un servizio che calcola il prossimo viaggio più probabile (*Most Likely Next Trip*);
* L’Operatore *MyData*;
* Un *Service Registry*, presso cui il servizio è registrato;
* Un gestore dei permessi per l’utilizzo del servizio e l’accesso ai dati dell’utente;
* Un Personal Data Vault per la memorizzazione dei dati.

## 4.1 Accounting e Servizi

https://github.com/HIIT/mydata-sdk/tree/master/Account

Al fine di gestire gli utenti all’interno del progetto, sono stati previsti due diversi tipi di account. Il primo si ottiene per mezzo dell’iscrizione dell’utente finale al servizio *MyData*. Ciò richiede l’inserimento obbligatorio di alcuni dati anagrafici come nome, cognome e data di nascita, che è possibile tuttavia modificare e arricchire in seguito. Questo tipo di account permette di associare ad ogni utente un Personal Data Vault, e costituisce punto di riferimento per tutti i singoli account creati presso i servizi. Non è possibile creare account duplicati.

Il secondo tipo di account fa riferimento alla registrazione che avviene al primo utilizzo di un servizio. Nel caso considerato, ad esempio, l’utente creerà un account specifico per il servizio Most Likely Next Trip, associato al suo account presso *MyData*; pertanto, ogni utente potrà avere un numero di account specifici pari ai servizi presso cui si è registrato. Inoltre, viene mantenuto l’elenco di tutti i permessi che l’utente ha generato per il servizio corrispondente.

Per quanto riguarda il servizio Most Likely Next Trip, si è scelto di utilizzare un componente già pronto, sviluppato da Nicola Ferroni nel suo lavoro di tesi “Un sistema di previsione degli itinerari per applicazioni di smart mobility”. All’interno di questo progetto, si svilupperà un esempio di come sia possibile far interagire le due entità, regolando lo scambio di dati in modo quanto più possibile aderente al modello *MyData*.

## 4.2 Operatore MyData

https://github.com/HIIT/mydata-sdk/tree/master/Operator\_Components

La realizzazione di un completo *MyData* Operator comporta la costruzione di diversi componenti. Le funzioni dell’Operatore comprendono la gestione del *Service Registry*, degli account utente e del processo di *Service Linking*. L’Operatore deve inoltre occuparsi della reciproca identificazione fra le parti nel processo di autorizzazione del servizio, durante la firma dei *Consent*, e autorizzare il flusso di dati quando ne viene fatta richiesta.

In questo lavoro si è preferito suddividere le responsabilità corrispondenti a queste operazioni in una molteplicità di classi, invece che un’unica entità, in conformità a principi di semplicità come il rasoio di Occam o l’acronimo KISS (*Keep It Simple, Stupid*). Pertanto, si realizzeranno separatamente:

* Un gestore di *Consent*, che si occupi dell’emissione dei permessi e del cambio di stato degli stessi in caso di richiesta da parte dell’utente.
* Un *Service Registry* semplificato, dove ogni servizio si registra per poter essere accessibile all’interno di MyData.
* Un’entità che metta in comunicazione il Personal Data Storage con il servizio, recuperando i dati necessari (punto di *policy enforcement*).
* Un gestore per le operazioni di autenticazione fra utente e servizio.

Per questo motivo, si sceglie nel seguito del progetto di non citare più il concetto di Operator, facendo invece riferimento alle entità particolari.

## 4.3 Service Registry, Service Linking

Una implementazione proposta dal team di sviluppo finlandese può essere trovata all’indirizzo ( <https://github.com/digitalhealthrevolution/serviceregistry> ) [ to fix ].

Inizialmente non era stata prevista alcuna implementazione di *Service Registry*, principalmente a causa della complessità non solo della struttura del componente, ma anche delle operazioni di registrazione di un nuovo servizio e di *Service Discovery*.

Come verrà descritto meglio nella sezione dedicata alla Progettazione, questo componente si è rivelato indispensabile per il funzionamento del sistema, ma la sua complessità è stata notevolmente ridotta. Infatti, si è scelto di non includere nessuna delle due descrizioni del servizio menzionate nelle specifiche, ma solo una indicazione sui tipi di dato necessari al suo funzionamento.

Questa scelta ha avuto una ripercussione anche sul protocollo di *Service Linking*, che ha perso di significato in mancanza dell’entità *Service Registry*. Tuttavia, vista la sua importanza concettuale, ho scelto di proporne comunque una variante all’interno del meccanismo delle autorizzazioni e del Consent.

## 4.4 Consent

A questo punto, si presenta la necessità di realizzare due tipi diversi di permessi.

Il primo contiene la semantica associata al procedimento di *Service Linking*: descrive quali sono le entità in gioco, fornendone una mutua autenticazione e contiene uno stato per indicare la validità del permesso considerato. I valori possibili di questo stato ricalcano quelli indicati nel modello *MyData*: *Active*, *Disabled*, *Withdrawn*. Questo Consent viene emesso ogni volta che l’utente fa richiesta di utilizzare un servizio, e in base al valore del suo stato permette (o meno) l’effettivo processamento dei dati.

Il secondo tipo di permesso viene utilizzato come token di accesso al Personal Data Storage ogni volta che si instaura un flusso di dati, sia esso in ingresso o in uscita dal PDS. Questo tipo di Consent contiene un riferimento ai dati scambiati durante la transazione (il Resource Set Identifier delle specifiche di MyData); inoltre, può essere utilizzato una volta sola ed è possibile ottenerlo solo se il permesso che lega l’utente ed il servizio corrente ha stato attivo.

Entrambi i Consent vengono memorizzati all’interno dell’account utente corrispondente al servizio che ne ha fatto richiesta.

### Granularità di un Consent

Un aspetto importante da considerare in Analisi è la granularità con cui i Consent permettono l’accesso ai dati.

All’interno delle specifiche di MyData non si trovano linee guida precise riguardo questo aspetto, mentre vengono espresse delle considerazioni riguardo la frequenza di accesso ai dati e la quantità di dati prelevati. L’interesse in questo senso è motivato dalla volontà di proteggere la privacy dell’utente, che potrebbe essere messa a rischio nel caso un servizio effettuasse un gran numero di richieste legittime per ottenere dati personali sempre diversi. All’interno dello sviluppo del gestore di dati personali si è scelto di privilegiare l’aspetto del controllo degli accessi, piuttosto che rispondere a quanto puntualizzato in altre implementazioni \cite mobility profile.

In una prima ipotesi, ho preso in considerazione la possibilità di garantire l’accesso ai dati in base a criteri che comprendono la loro tipologia, la quantità di dati richiesta e la loro “età”. L’implementazione di questa scelta avrebbe previsto, a grandi linee, l’utilizzo di un database per la gestione di dati con un gran numero di proprietà (come ad esempio la data di acquisizione sopra citata). Poiché tale soluzione avrebbe richiesto notevole impegno ed avrebbe sconfinato all’esterno dell’ambito della tesi, questa possibilità è stata scartata.

Come seconda opzione, sulla scia del Mobility Profile e della gestione delle autorizzazioni in Android, si è optato per il seguente compromesso: la granularità per il controllo degli accessi viene imposta a livello di tipi di dato di alto livello, come ad esempio Calendario, storico delle posizioni GPS, ecc. Per questo motivo, si è scelto di sostituire al generico identificatore di un set di risorse sopra menzionato un insieme di specifici tipi di dato, al fine di poter implementare un adeguato controllo degli accessi.

## 4.5 Personal Data Storage

L’ipotesi iniziale per il Personal Data Storage era quella di realizzare un componente indipendente dalle scelte dell’utente o dalle necessità del servizio, come anche dalle scelte implementative.

Un esempio di indipendenza dalle scelte dell’utente è il seguente. Il servizio Most Likely Next Trip utilizza al suo interno i dati di un calendario, secondo l’assunto per cui l’utente deve compiere un viaggio per portarsi nel luogo dell’evento inserito. La soluzione più immediata è quella di salvare all’interno del PDS gli impegni necessari, o la struttura dati del calendario in un file specifico per l’applicazione di Calendario utilizzata dall’utente. Questa scelta però non solo è poco efficiente, ma vincola all’utilizzo di un particolare tipo di calendario, impedendo successive modifiche o estensioni. Una soluzione alternativa potrebbe essere quella di utilizzare uno standard di rappresentazione di calendari, attualmente il formato .ics, anche se è ancora diffusa la versione precedente, .vcs. Questa scelta, seppure legata ancora ad un tipo di implementazione a file, offre maggiore interoperabilità grazie alla scelta di un formato supportato anche da servizi esterni.

Alternativamente, è possibile astrarre ad un livello ancora maggiore l’implementazione del calendario, ad esempio utilizzando le API di servizi online (come quelle di Google Calendar, <https://developers.google.com/google-apps/calendar/overview> ) per integrare quelli inseriti nel Personal Data Storage. Attraverso l’utilizzo di molteplici livelli di astrazione sarebbe possibile soddisfare la richiesta dell’utente di utilizzare uno specifico calendario, senza però cablarne questa scelta nell’implementazione.

In ultima analisi, anche a causa di una implementazione già presente all’interno del progetto Most Likely Next Trip si è scelto di adottare una persistenza che faccia uso di file di testo. Tuttavia, attraverso l’uso di interfacce ho cercato di non propagare le dipendenze dall’implementazione scelta all’esterno della classe, in modo da permettere un più semplice refactoring in caso di sviluppi futuri.

## 4.6 Rappresentazione di dati non noti a priori

Una delle maggiori difficoltà incontrate durante lo studio di *Service Registry* e Personal Data Storage riguardava la necessità di descrivere tipi di dato non noti a priori, e di possedere strumenti per il loro processamento. Questa problematica interessa il *Service Registry* al momento dell’iscrizione del servizio presso *MyData* e il Personal Data Storage quando questo prende parte ad uno scambio di dati con il servizio.

Le specifiche del modello *MyData* consentono di ottenere l’indipendenza da tipi di dato specifici tramite l’utilizzo linguaggi che descrivono la tassonomia e le strutture dati necessarie al funzionamento del servizio. In particolare, viene fatto uso di RDF (Resource Description Framework), che permette la rappresentazione di informazioni all’interno del web ( <https://www.w3.org/TR/rdf11-concepts/> ).

Alcuni esempi utilizzati in *MyData* sono W3C’s Data Catalog Vocabulary e The RDF Data Cube Vocabulary, utilizzati all’interno del *Service Registry* nella descrizione del servizio.

Per ottenere l’indipendenza e l’interoperabilità al centro del modello di MyData, utilizzando allo stesso tempo una soluzione più adatta ad un contesto ridotto, ho preso considerato la possibilità di trasmettere i dati necessari in linguaggio JSON.

# 5 Progettazione

Nei paragrafi successivi verranno illustrate le fasi di implementazione del gestore di dati personali, motivando le scelte implementative e le eventuali differenze che si creano rispetto a quanto detto in Analisi.

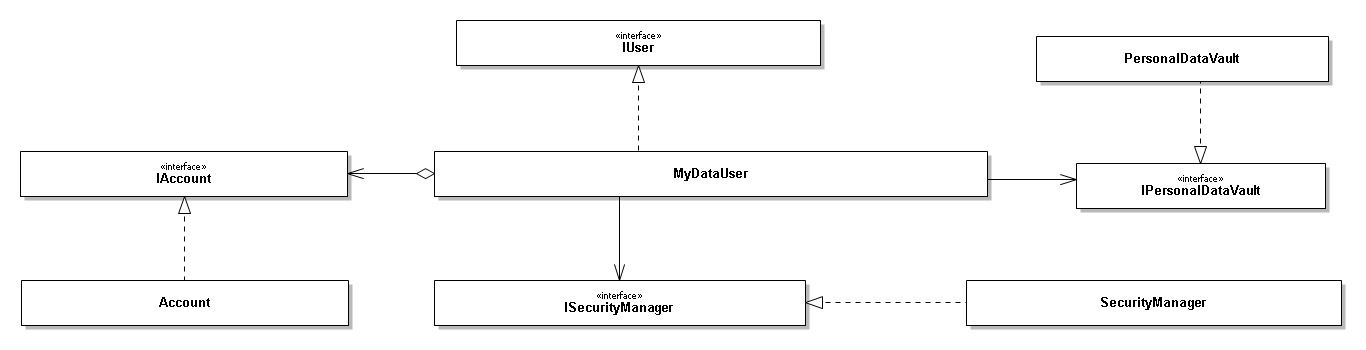
Per la realizzazione del gestore è stato utilizzato il linguaggio Java.

Fra i principi generali seguiti in Progettazione troviamo l’inversione delle dipendenze, la separazione delle responsabilità, il principio di sostituibilità di Liskov e il già citato rasoio di Occam. Secondo il principio di inversione delle dipendenze, è necessario che le dipendenze presenti all’interno del codice non siano fra classi ma fra interfacce, in modo da evitare che la struttura possa risentire di cambiamenti che avvengono a basso livello. Il principio di separazione delle responsabilità stabilisce che ogni classe deve avere un solo compito, da svolgere interamente, ma mai più di uno: lo sviluppo di classi con più di una responsabilità genera dipendenze non volute fra le classi, rendendo il codice fragile. Infine, il principio di sostituibilità di Liskov si applica ai casi di ereditarietà fra classi, e ne regola il rapporto: ogni sottoclasse deve poter essere utilizzata al posto della classe base senza che ci si accorga della differenza.

## 5.1 Flusso del programma

//inserire grafico in latex

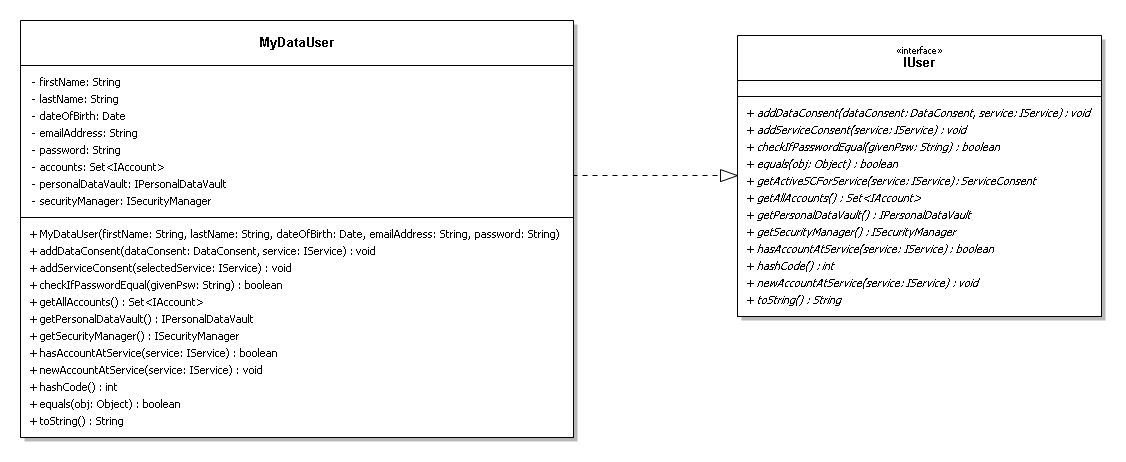
## 5.2 Accounting



Figura

Al centro è collocata la classe corrispondente all’utente MyData, che, confermando quanto detto in Analisi, è collegata agli account dei servizi, e al Personal Data Vault dell’utente. Una novità è invece la coppia ISecurityManager, SecurityManager creata per soddisfare i requisiti di sicurezza relativi alla mutua autenticazione fra utente e servizio. Si è deciso di sviluppare separatamente questa classe per un principio di separazione delle responsabilità, e per permettere una più semplice estendibilità in caso di sviluppi futuri.

### 5.2.1 IUser, MyDataUser



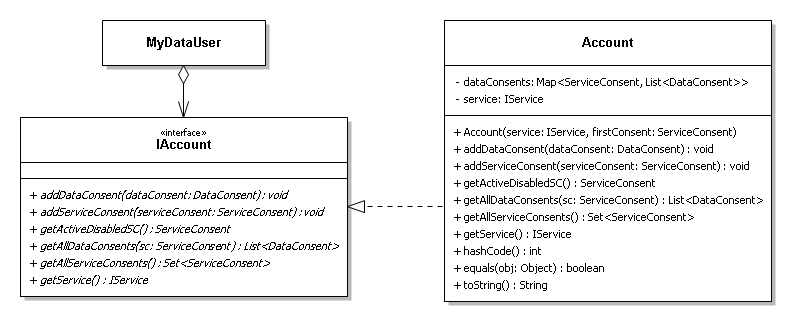
Figura

Questa classe modella un generico utente dell’architettura MyData. I field al suo interno sono un esempio delle caratteristiche che si è scelto di modellare, e fra di essi i più rilevanti sono indirizzo email e password, in quanto permettono il login per utenti già registrati. L’indirizzo mail è stato adottato inoltre come identificatore unico di un utente all’interno di MyData, e questa caratteristica è stata implementata mediante l’override della funzione equals(Object obj).

Si evidenzia inoltre la presenza di un Set<IAccount> accounts, che realizza l’associazione fra un utente e gli account presso i servizi a cui si è registrato. La scelta di un Set permette di implementare il vincolo secondo cui ogni utente può avere un solo account presso un certo servizio, ed è adatta anche in quanto non è necessario mantenere un insieme ordinato di account.

Questa classe ha inoltre la funzione di interfacciare gli altri componenti del gestore, compresa la GUI, con gli account utente. A tal fine, presenta i metodi addServiceConsent(IService service), addDataConsent(DataConsent dataConsent, IService service), hasAccountAtService(IService service). La classe Account è stata infatti modellata con visibilità package protected, per impedire l’accesso a classi esterne al package users: di conseguenza, anche la creazione di nuovi account avviene attraverso questa classe, in particolare nella funzione newAccountAtService(IService service). All’interno del metodo troviamo l’istanziazione di un nuovo account, insieme ad una chiamata alla classe ConsentManager, che realizza quanto anticipato al paragrafo 4.4 Consent. In questo modo si realizza un esempio di *Service Linking*, e l’esito di questa operazione viene concretizzato in un oggetto ServiceConsent. Si rimandano però ulteriori dettagli a quanto spiegato nella sezione 5.6 Autorizzazioni e Consent.

### 5.2.2 IAccount, Account



Figura

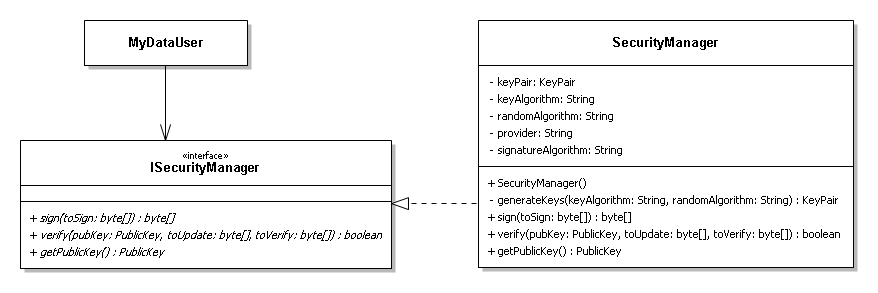
La classe Account è abbastanza semplice, poiché si occupa semplicemente di implementare la logica di basso livello nelle operazioni di gestione degli account.

Fra queste vi sono i controlli sullo stato dei Consent memorizzati, la gestione dello storico di tutti i Consent emessi per quel servizio service, o ancora il matching fra i due tipi di Consent (ServiceConsent, DataConsent, dettagliati al paragrafo 5.6.2 ServiceConsent, DataConsent).

La memorizzazione dei Consent all’interno della classe è stata ottenuta mediante l’utilizzo combinato delle strutture dati Map<ServiceConsent, List<DataConsent>>. Questa scelta permette di esprimere diversi concetti a livello semantico. Come prima cosa, per i Consent sul flusso di dati si è scelto di utilizzare la classe base DataConsent invece che le sue due implementazioni, in modo da poterli memorizzare indiscriminatamente. Ciò verifica l’utilizzo del principio di sostituibilità di Liskov. Inoltre, la scelta di una List come value all’interno di una mappa permette di descrivere un flusso di dati (all’interno dello stesso ServiceConsent) per il quale si sono rivelate necessare una molteplicità di interazioni fra *Source* e *Sink*, ognuna delle quali modellata da un DataConsent.

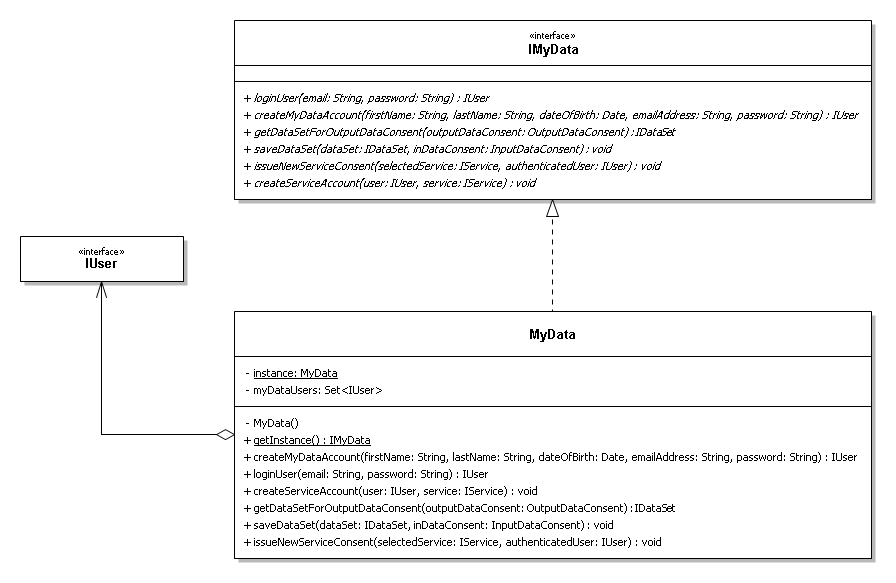
Ad ogni istanza di ServiceConsent corrisponde quindi una Collection dei DataConsent emessi durante il periodo di validità dello stesso, ed è possibile avere un unico ServiceConsent attivo in un determinato istante di tempo.

## 5.3 SecurityManager



Figura

## 5.4 IMyData, MyData



Figura

La classe MyData svolge all’interno del gestore di dati personali un importante ruolo di coordinazione fra le parti, poiché realizza al suo interno una parte dell’Operatore MyData, secondo quanto specificato in 4.2. Essa è un punto di riferimento per l’interfaccia utente, alla quale fornisce i dati da elaborare e mostrare a video, e dalla quale riceve le richieste effettuate dall’utente.

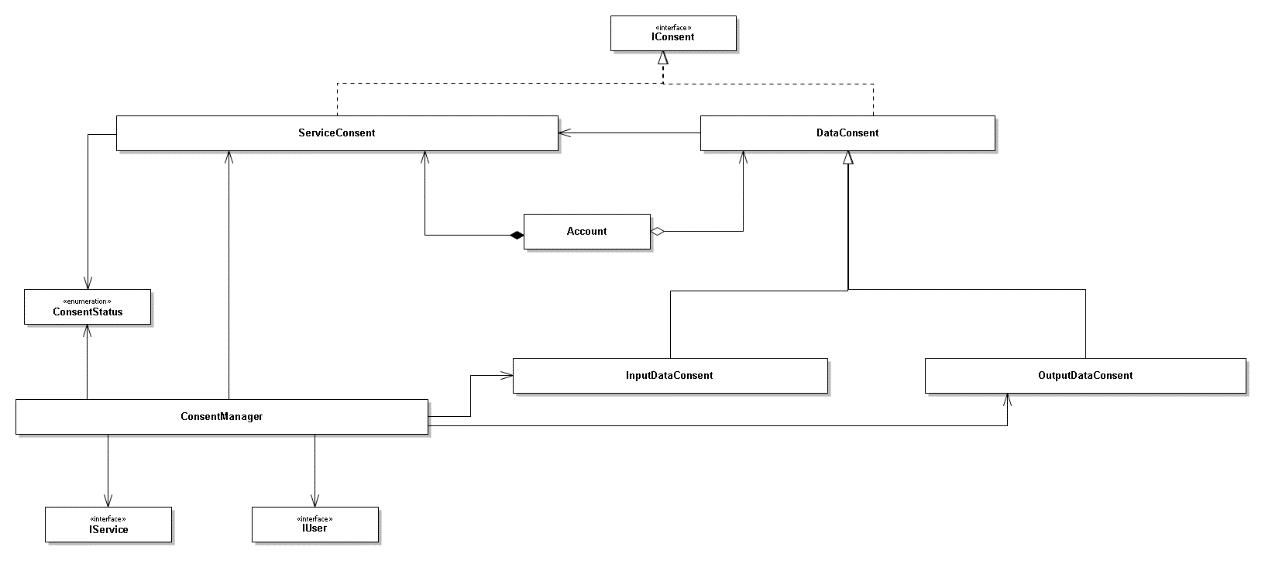
Si occupa quindi come prima cosa di registrare e autenticare gli utenti (metodi createMyDataAccount e loginUser) in modo da impedire la creazione di duplicati. I controlli in questo senso vengono effettuati su un Set<IUser> contenuto all’interno della classe (si sfrutta la proprietà della struttura dati Set di non ammettere duplicati). La creazione di un nuovo account presso un determinato servizio viene gestita da questa classe, tramite invocazione dell’opportuno metodo esposto dall’interfaccia IUser, insieme alla richiesta di nuovi ServiceConsent in caso di utente già registrato.

In secondo luogo, ogni servizio che ha richiesto e ottenuto il permesso di accedere a uno specifico insieme di dati personali di un utente fa riferimento alla classe MyData, che si occupa di intercedere presso il Personal Data Vault per ottenere quanto richiesto. L’operazione si svolge sia per i dati in ingresso che per i dati in uscita dal Vault mediante i metodi getDataSetForOutputDataConsent(OutputDataConsent outputDataConsent) e saveDataSet(IDataSet dataSet, InputDataConsent inDataConsent). In entrambi i casi, viene controllata la validità del Consent emesso prima di effettuare la richiesta di dati personali.

Infine, si evidenzia la realizzazione della classe MyData come Singleton mediante l’utilizzo di un costruttore privato e di un campo instance di tipo MyData. Ciò assicura la presenza di un unico Operatore di questo tipo all’interno del programma

## 

## 5.6 Autorizzazioni e Consent



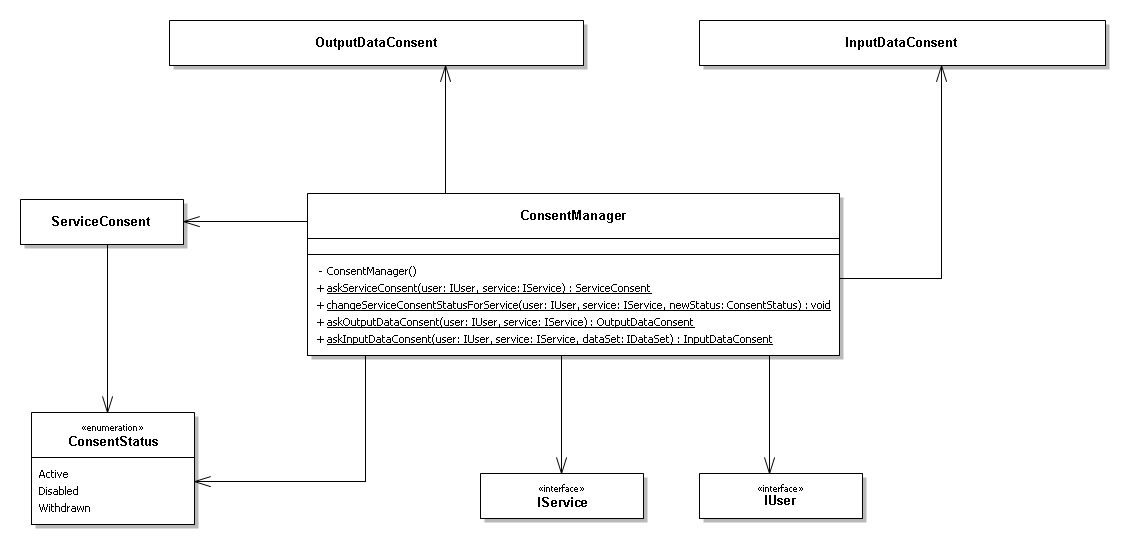
Figura

Per quanto riguarda la gestione delle autorizzazioni e dei permessi, si procede come prima cosa ad uno studio di alto livello.

Le classi più importanti all’interno di questa architettura sono ConsentManager, che realizza il gestore di permessi previsto in 4.2, e le due tipologie di permessi, anch’esse previste in fase di Analisi (4.4).

È presente infine anche l’enumerativo ConsentStatus, attraverso il quale si rappresentano gli stati del rapporto fra un utente ed un servizio generici.

### 5.6.1 ConsentManager, ConsentStatus



Figura

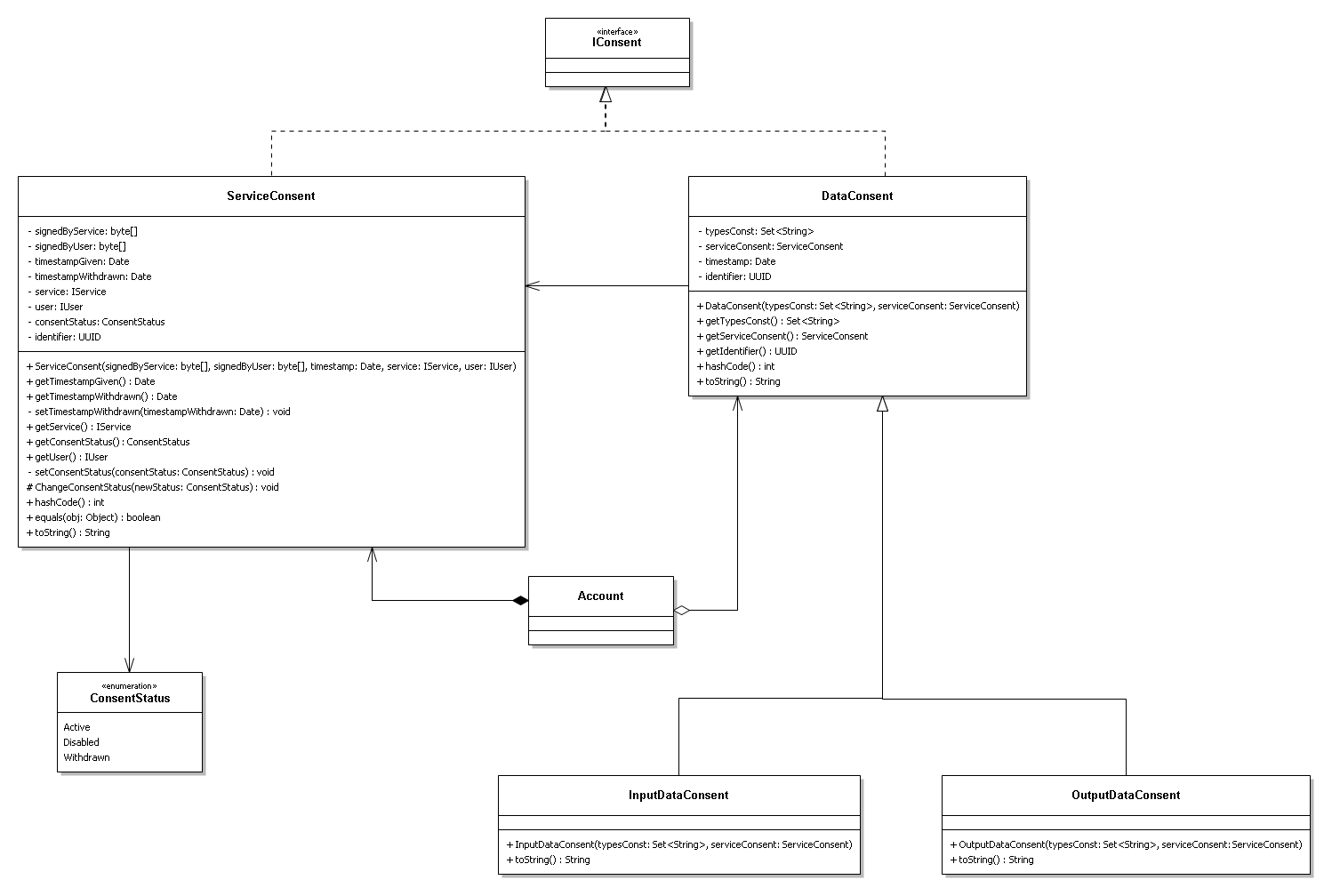
La classe ConsentManager si occupa dell’erogazione, in caso di richiesta legittima, di vari tipi di permessi ai servizi che ne fanno richiesta. Poiché il suo scopo è quello di garantire il rispetto di un determinato protocollo di assegnazione dei permessi, essa è prima di tutto una classe implementativa e per questo motivo non è previsto alcun tipo di astrazione (ad esempio tramite interfaccia).

Si potrebbe considerare di rendere la classe final, per impedire estensioni o ridefinizioni del comportamento. Le motivazioni a supporto di questa scelta risiedono nella garanzia di una maggiore sicurezza. Tuttavia, nel complesso ciò avrebbe portato ad una eccessiva rigidità del codice, impedendo aggiornamenti del protocollo anche in casi di legittima necessità.

Per la sua implementazione, ho cercato di fare in modo che essa realizzasse un servizio il più possibile indipendente dal resto dell’architettura circostante. Pertanto, la classe ConsentManager non mantiene alcuno stato interno, presenta costruttore privato e non ha dipendenze rilevanti. Inoltre, poiché la procedura di verifica dei requisiti è costante e indipendente dai parametri di ingresso, ogni metodo è stato realizzato come static. È

I tipi di Consent erogati dalla classe ConsentManager sono ServiceConsent e le classi “figlie” InputDataConsent, OutputDataConsent, ognuno con un metodo dedicato. È presente anche la procedura changeServiceConsentStatusForService (IUser user, IService service, ConsentStatus newStatus), che permette all’utente di cambiare lo stato del Service Consent correntemente attivo o disabilitato, secondo quanto previsto dalle specifiche di MyData e successivamente in fase di Analisi (sez. 4.4).

### 5.6.2 ServiceConsent, DataConsent



Figura

I permessi utilizzati all’interno del gestore di dati personali ed erogati dalla classe ConsentManager sono istanze delle classi ServiceConsent, InputDataConsent e OutputDataConsent. Come è possibile osservare dal diagramma UML in figura, InputDataConsent e OutputDataConsent estendono la classe DataConsent: la loro funzione è principalmente semantica, in quanto non aggiungono logica al programma ma descrivono il verso del flusso di dati che si crea con il Personal Data Vault. Pertanto, descriverò principalmente le caratteristiche delle classi ServiceConsent e DataConsent, che costituiscono il punto focale della realizzazione dei permessi descritti in MyData.

Nonostante la differenza di realizzazione e di utilizzo, entrambe le classi ServiceConsent e DataConsent implementano una interfaccia comune IConsent. Questa è una interfaccia “marker” necessaria per esprimere una somiglianza a livello semantico, in quanto entrambe le classi descrivono un tipo di autorizzazione.

Dal diagramma UML è possibile dedurre il ruolo della classe Account rispetto ai due tipi di Consent. Come accennato infatti in 5.2.2, essa mantiene al suo interno una mappa di corrispondenze fra ServiceConsent e liste di DataConsent.

Nel primo caso, la relazione è rappresentata mediante il simbolo “rombo nero”, che qualifica la classe Account come “contenitore” di istanze della classe ServiceConsent. In particolare, il rombo nero descrive un tipo di relazione molto stretta fra le due parti, e la scelta è dovuta alle specifiche di MyData, secondo cui non è possibile registrarsi presso un servizio senza ottenere un Consent. Ciò è stato implementato mediante l’emissione di un ServiceConsent prima della creazione effettiva dell’account, e il legame fra i due avviene tramite il passaggio di questo permesso al costruttore della classe Account.

La classe ServiceConsent realizza il primo – e il più rilevante – dei due tipi di permessi previsti per il gestore di dati personali. Al suo interno troviamo i token firmati da utente e servizio per la mutua autenticazione, insieme ai rispettivi riferimenti; vi sono inoltre anche alcuni campi per l’identificazione del Consent stesso e la sua collocazione temporale.

Per quanto riguarda invece i DataConsent, essi si comportano come access token per il Personal Data Vault validi una sola volta e conservati come storico dell’accesso ai dati personali. A tal fine, un DataConsent contiene al suo interno il Set<String> che contiene l’elenco dei tipi di dato a cui il servizio beneficiario può accedere. Per impedire accessi illegittimi, viene sempre controllata la corrispondenza fra i tipi di dato dichiarati in fase di registrazione e quelli richiesti alla creazione del DataConsent. Poiché il servizio non può interrogare direttamente il Personal Data Vault, gli scambi avvengono in base a quanto dichiarato all’interno del Set<String>.

Infine, si evidenzia dal diagramma UML che ogni DataConsent mantiene un riferimento al ServiceConsent attivo al momento della sua emissione, al fine di avere una migliore tracciabilità delle transazioni di dati, e che il costruttore della stessa classe ha visibilità package - protected in modo da obbligare l’uso delle sottoclassi alle classi esterne.

## rappresentazione dei dati

Si presenta a questo punto la problematica di rappresentazione dei dati, non solo all’interno del Personal Data Storage ma anche durante le transazioni fra Source e Sink, e all’interno del servizio stesso.

La soluzione implementata all’interno del Mobility Profile prevede l’uso di oggetti JsonArray e GeoJSON, come indicato nella documentazione \cite{ githubmobilityprofilespecification}, tuttavia questa implementazione differisce dal caso di studio poiché il servizio che utilizza i dati per calcolare il prossimo viaggio più probabile e il Personal Data Storage si trovano all’interno della stessa app. Come anticipato in 2.1.1 infatti, l’applicazione Journey Planner è solamente un frontend che non esegue alcuna computazione ma presenta un risultato già pronto.

All’interno del gestore di dati personali in oggetto si vuole invece permettere un flusso di dati fra Source e Sink generici in modo che la computazione avvenga presso il servizio che richiede i dati, non localmente al Personal Data Storage.

In fase di Analisi (sezione 4.6) il problema è stato preso in considerazione, e si è proposta come soluzione l’utilizzo del formato JSON per la comunicazione e il trasferimento di dati.

In questo senso, come prima cosa ho studiato i componenti Java disponibili nell’architettura per realizzare la conversione in stringhe JSON: JsonArray, JsonObject e altre sotto-interfacce di JsonValue. Il motivo per cui ho scelto di non utilizzarli è che questi non offrono alcun metodo di utilità per la conversione da oggetto a stringa, e la trasposizione di ogni field va realizzata manualmente sia in serializzazione che in deserializzazione. <http://docs.oracle.com/javaee/7/api/javax/json/JsonObject.html> Una scelta di questo tipo implicherebbe un precedente accordo fra le parti per stabilire come interpretare le stringhe inviate, e una forte dipendenza dalla particolare implementazione dei dati utilizzati.

Come seconda opzione ho considerato di utilizzare la libreria Google Gson <https://github.com/google/gson> , in quanto essa risolve i problemi incontrati nel corso del primo tentativo grazie ai metodi gson.toJson(obj) per la serializzazione e gson.fromJson(json, obj.class) per la deserializzazione. Questo approccio funziona perfettamente in caso di oggetti che contengono tipi primitivi, ma mostra qualche limitazione quando si introducono oggetti di tipo generico e Collection di oggetti, siano esse di oggetti di un unico tipo o di tipi diversi. Il motivo risiede nell’implementazione della Java Virtual Machine, e in particolare nella sua caratteristica di Type Erasure per la quale ogni oggetto a basso livello “perde” il suo tipo particolare per diventare un Object. Questo non crea problemi in serializzazione ma in deserializzazione, quando risulta impossibile recuperare il tipo originario dell’oggetto da deserializzare. Poiché l’utilizzo della libreria Gson nel contesto del PersonalDataVault si sarebbe collocato all’interno dei casi non completamente supportati, ho scelto di scartare anche questa seconda possibilità.

Questa problematica (l’impossibilità di prevedere con sufficiente precisione tutte le possibili situazioni legate alla rappresentazione dei dati in ogni momento) riveste un serio aspetto di complessità e articolazione tali da assumere una rilevanza non contenibile nei limiti del presente contesto di lavoro di tesi.

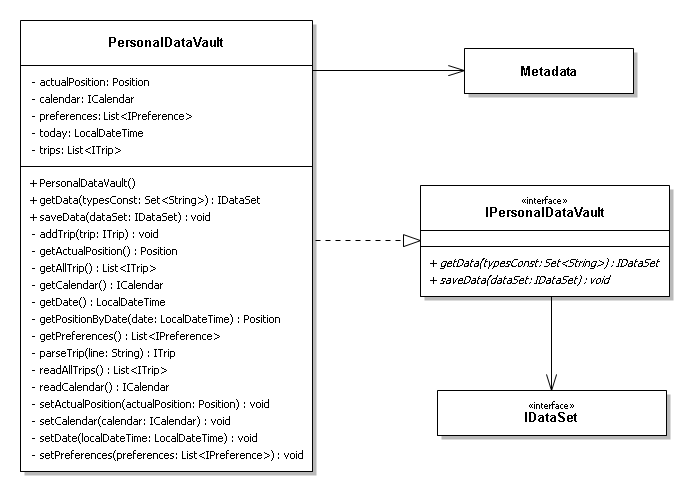
Necessiterebbe, viceversa, di organizzazione, mezzi e tempistiche - oltre che conoscenze - non immediatamente disponibili nell'attuale contesto.

Si rende necessario, pertanto, ricercare una soluzione diversa, più concreta e adeguata alle attuali circostanze, più ammissibile anche rispetto agli obiettivi di questo lavoro.

L'ipotesi che viene proposta, che si può ritenere più accettabile, consiste

La soluzione da me proposta a questo problema consiste nell’utilizzo di una struttura dati DataSet che permetta lo scambio di dati di qualunque tipo all’interno del sistema, purché il loro tipo sia stato preventivamente dichiarato in fase di registrazione (sezione 5.9). La classe DataSet è descritta in dettaglio nella sezione 5.7.

### 5.7 PersonalDataVault



Figura

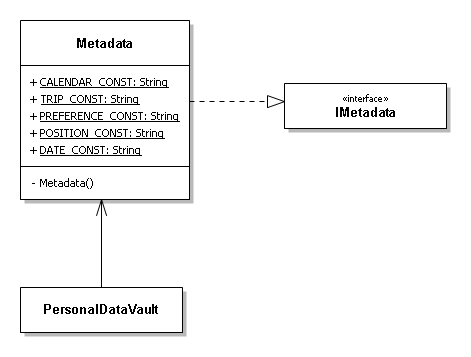
In questa sezione si presenta la classe PersonalDataVault, in cui vengono mantenuti i dati personali dell’utente. Essa corrisponde direttamente al Personal Data Storage proposto all’interno della documentazione MyData.

L’interfaccia \texttt{IPersonalDataVault} espone i metodi \texttt{getData (Set<String> typesConst)} (che restituisce un \texttt{IDataSet}) e \texttt{saveData (IDataSet dataSet)} che permettono l’accesso ai dati. Essi si occupano di prelevare dati, o salvarli, in base a quanto specificato dal \texttt{Set<String>} contenente l’insieme dei tipi di dato. Inoltre, la signature garantisce l’indipendenza dai tipi di dato in ingresso o in uscita dal Vault grazie all’uso di oggetti di tipo \texttt{DataSet} incapsulati in opportune interfacce \texttt{IDataSet}.

All’interno della classe \texttt{PersonalDataVault} troviamo alcuni metodi privati necessari per la gestione dei dati memorizzati (ad esempio la lettura e la scrittura su file, con opportune funzioni di utility necessarie per il parsing). Le scelte effettuate in questo senso derivano da una precedente implementazione e sono state adattate solo in minima parte per motivi di compatibilit\`a. Per quanto riguarda invece i metodi ereditati dall’interfaccia, si evidenzia l’eventualità di una RuntimeException (ClassCastException): il verificarsi di questo evento non è auspicabile, ma rappresenta comunque uno stato del sistema previsto e voluto. Qualora infatti un oggetto dovesse rivelarsi di un tipo diverso rispetto a quello dichiarato all’interno del dataSet ricevuto in ingresso, è opportuno che il programma segnali questa grave inconsistenza, terminando la sua esecuzione. Incidentalmente, il verificarsi di una situazione di questo tipo implica la non osservanza delle politiche e dei protocolli di sicurezza, pertanto si ritiene motivato l’utilizzo di questa eccezione di basso livello.

Si può osservare che l’implementazione del PersonalDataVault è particolarmente dipendente dai tipi di dato utilizzati dal servizio Most Likely Nex Trip scelto come caso di studio. Tale caratteristica costituisce una limitazione del gestore di dati personali realizzato in questa tesi e trova un riscontro all’interno delle problematiche evidenziate all’interno della sezione 5.6. Anche la scelta di memorizzare i dati acquisiti all’interno di file di testo, derivata da una precedente implementazione, si colloca all’interno di un orizzonte limitato: alcune proposte di aggiornamento sono state discusse in 4.5, e saranno oggetto di ulteriore discussione all’interno del capitolo 6.

### 5.8 Metadata



Figura

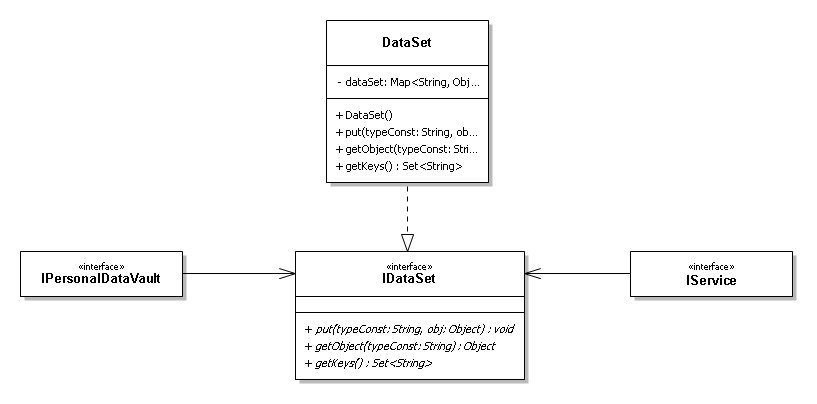
Come già accennato in precedenza, un approccio diverso, più completo ed esaustivo, avrebbe comportato mezzi più consistenti e conoscenze di cui - al momento - non si dispone.

Si è ritenuto, in alternativa, di dover ricorrere alla definizione preventiva dei tipi di dato disponibili all’interno del sistema, quali ad esempio Position e ICalendar, utilizzando dove possibile le interfacce al posto delle classi, onde diminuire la forte dipendenza dalla particolare implementazione.

Questa modalità potrebbe forse apparire come una eccessiva semplificazione del contesto per la scelta arbitraria dei tipi di alto livello sopra menzionati e delle loro implementazioni. La soluzione proposta, però, è da ritenersi plausibile in quanto valuta che i componenti indicati siano tali da poter rendere risultati accettabili anche in presenza delle limitazioni assunte.

Pertanto, la classe Metadata ha il compito di definire a priori i tipi di dato disponibili all’interno dell’Operatore MyData. A livello implementativo, la classe espone un certo numero di stringhe static final, ognuna delle quali è inizializzata con il fully qualified name della classe (o interfaccia) che rappresenta, al fine di conservare l’informazione completa riguardo al tipo di dato. All’interno del programma, esse sono accessibili mediante la notazione Metadata.DATATYPE\_CONST, ad esempio al momento della costruzione dei Set<String> necessari per la registrazione presso il ServiceRegistry (sezione ), o nel controllo della legittimità dell’emissione di un permesso (sezione ).

### DataSet



Figura

Come anticipato nelle sezioni precedenti, la classe DataSet viene utilizzata per il trasferimento di dati fra Source e Sink. Dal grafico UML in figura si nota che essa viene sempre referenziata per mezzo dell’interfaccia IDataSet, ed è inoltre possibile riconoscere IPersonalDataVault e IService nei ruoli (non statici) di Source e Sink.

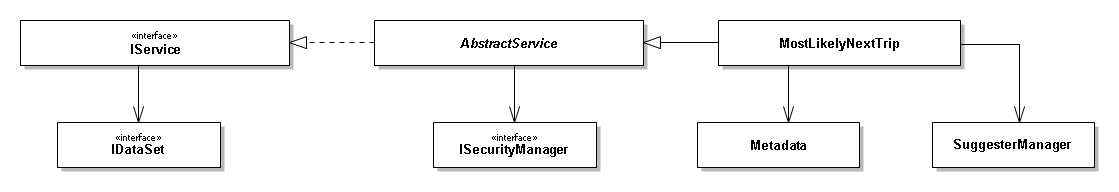
L’oggetto DataSet contiene i dati da trasferire mediante l’utilizzo di una Map<String, Object>, dove le chiavi sono le costanti di tipo stringa esposte dalla classe Metadata e i value corrispondenti sono generici Object (non sono permessi valori null). In altre parole, questa struttura dati associa ad ogni oggetto il suo tipo, espresso all’interno di una stringa.

L’obiettivo della classe DataSet è contenere un insieme di oggetti il cui tipo non è noto a priori senza perdere l’informazione sul tipo stesso dopo la conversione a Object.

In questo modo si trova una soluzione al problema della Type Erasure, poiché l’informazione sul tipo di dato originario viene conservata separatamente all’oggetto stesso. Il destinatario della trasmissione è in grado, attraverso le costanti di tipo stringa, di risalire al tipo dell’oggetto ricevuto, in modo da poterlo utilizzare in modo appropriato.

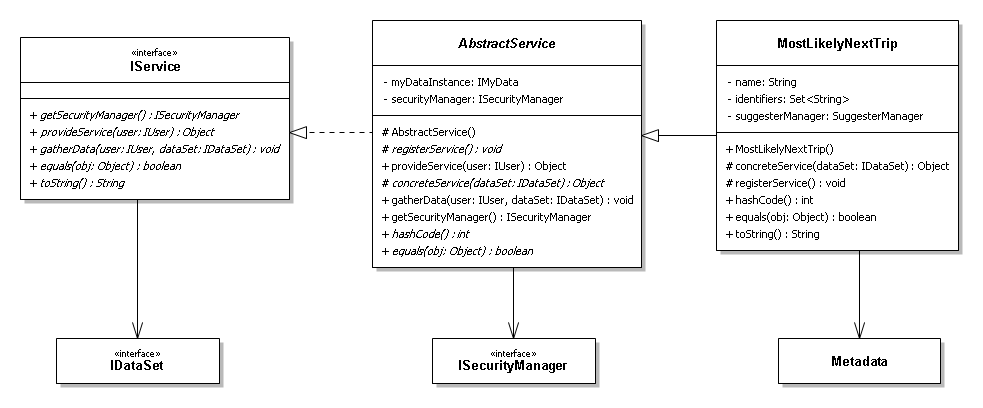
I metodi esposti dall’interfaccia ridefiniscono quelli propri della struttura dati Map che costituisce lo stato interno della classe, aggiungendo il controllo per i valori null e per la richiesta di oggetti non disponibili all’interno della mappa data.

## 5.9 IService, AbstractService, MostLikelyNextTrip



Figura

Il diagramma UML in figura mostra l’implementazione proposta per l’utilizzo di servizi all’interno dell’architettura MyData. L’interfaccia IService e la classe astratta AbstractService sono state realizzate al fine di mediare l’interazione fra servizio concreto (in questo caso MostLikelyNextTrip) e Operatore (ad esempio, classi ServiceRegistry o IMyData). In questo senso, si può dire che la classe AbstractService raccoglie a fattore comune le operazioni comuni a tutti i servizi, lasciando alle classi figlie il compito di realizzare solo le parti fortemente dipendenti dalla particolare business logic. Per realizzare un nuovo servizio è necessario quindi creare una classe che estende AbstractService: nel caso di studio, essa è la classe MostLikelyNextTrip, che si interfaccia con il servizio di calcolo del prossimo viaggio più probabile[cite] SuggesterManager.



Figura

Il livello più astratto di definizione dei servizi in MyData è rappresentato dall’interfaccia IService, e permette il loro riferimento all’interno dell’architettura. Fra i metodi esposti si evidenziano provideService (IUser user) e gatherData (IUser user, dataSet IDataSet), che costituiscono i punti di ingresso e di uscita del flusso di dati utilizzati e prodotti dal servizio.

L’interfaccia viene implementata parzialmente dalla classe AbstractService.

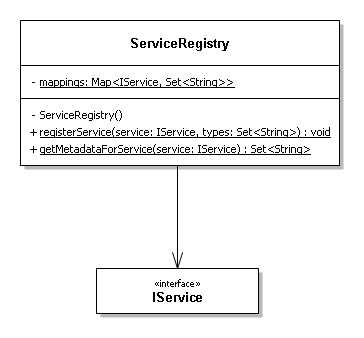
Il metodo provideService raccoglie a fattore comune la richiesta di un OutputDataConsent al ConsentManager, seguita dall’effettiva richiesta di dati inoltrata al PersonalDataVault tramite una richiesta all’operatore MyData. Infine, viene restituito il risultato della chiamata alla funzione abstract protected concreteService (IDataSet dataSet): la visibilità garantisce che dall’esterno venga chiamato solo il metodo esposto dall’interfaccia IService. La signature astratta indica inoltre che questa funzione contiene l’effettiva implementazione della business logic, delegata alle classi concrete che estendono AbstractService.

Il metodo gatherData segue la stessa sequenza di passi: la richiesta al ConsentManager di un InputDataConsent e l’interazione con il PersonalDataVault mediante Operatore MyData sono però preceduti da alcuni controlli sui dati in ingresso, al fine di verificare la legittimità della richiesta.

Fra i metodi astratti si evidenzia infine registerService(), da invocare all’interno del costruttore della classe figlia. Durante la chiamata a procedura, il servizio concreto specifica quali tipi di dato utilizzerà mediante l’utilizzo delle costanti descritte in dettaglio nella sezione 5.7.

Della classe implementativa MostLikelyNextTrip si evidenzia solo la presenza dello stato interno, dove sono contenuti il nome, identificatore unico del servizio, e il Set<String> che contiene i tipi di dato utilizzati dal servizio.

## 5.10 ServiceRegistry



Figura

Ogni servizio che voglia essere utilizzabile all’interno dell’infrastruttura MyData deve, per prima cosa, registrarsi all’interno del Service Registry. In fase di registrazione, ogni servizio dichiara inoltre i tipi di dato necessari per il suo funzionamento, al fine di agevolare gli scambi di dati con altre entità dell’infrastruttura, siano esse Source o Sink.

Per questo motivo, la classe ServiceRegistry ha come responsabilità fondamentale quella di mantenere al suo interno le corrispondenze fra i servizi registrati e i tipi di dato: ciò viene realizzato mediante l’utilizzo di una Map<IService, Set<String>>.

Come per la classe ConsentManager (analizzata più in dettaglio nella sottosezione 551), anche in questo caso ho cercato di rendere il Service Registry il più possibile indipendente tramite l’uso di metodi statici e la scelta di un costruttore privato. I metodi registerService(IService service, Set<String> types) e Set<String> getMetadataForService(IService service) incapsulano quelli di basso livello esposti dalla struttura dati Map, aggiungendo gli opportuni controlli sui parametri in ingresso. In particolare, è possibile registrare un servizio mediante la procedura registerService, e ottenere i tipi di dato registrati mediante la funzione getMetadataForService.

## 5.11 Uso delle eccezioni

All’interno del gestore di dati personali sviluppato nell’ambito della tesi in oggetto, è di importanza non trascurabile il controllo della legittimità delle operazioni da eseguire, degli argomenti in ingresso alle funzioni o degli stati in cui si trova il gestore stesso. Pertanto, è necessario uno strumento che permetta di descrivere il verificarsi di situazioni non previste o illegittime, impedendo la continuazione del flusso del programma. Questa situazione è diversa, ad esempio, dal caso in cui l’invocazione di un metodo è legittima ma non produce alcuni risultato, poiché è di fondamentale importanza che l’esecuzione termini.

Lo strumento utilizzato per ottenere questa caratteristica sono le Java Exception, ed in particolare le sottoclassi di RuntimeException IllegalArgumentException e IllegalStateException.

Fra di esse, IllegalStateException è descritta all’interno della documentazione Java nel modo seguente: “Signals that a method has been invoked at an illegal or inappropriate time. In other words, the Java environment or Java application is not in an appropriate state for the requested operation”. Ho valutato quindi che questa eccezione fosse adeguata in tutti quei casi in cui l’invocazione di un metodo non può essere portata a termine a causa di Consent non adeguati. Un esempio può essere la richiesta di un nuovo DataConsent nel caso in cui non vi sia un ServiceConsent attivo in quel momento.

L’uso di IllegalArgumentException è convenzionalmente limitato al controllo dei parametri in ingresso ai metodi, in aderenza a quanto dichiarato sulla documentazione Java. Fra le RuntimeException utilizzate è presente anche SecurityException, invocata ogni volta in cui un parametro o controllo di sicurezza non è verificato.

Un aspetto importante dell’uso delle eccezioni è stata la loro collocazione all’interno dello stack delle invocazioni di metodi. Si prenda nuovamente come esempio il caso di richiesta di un nuovo DataConsent: è noto a priori che la precondizione per ottenerlo richiede l’esistenza di un ServiceConsent attivo in quel momento. Questa eventualità è controllata all’interno della classe ConsentManager mediante l’uso di una IllegalStateException. Tuttavia esiste un sotto-caso che, pur rientrando a pieno titolo nell’insieme di situazioni non corrette (mancanza di un ServiceConsent attivo) merita di essere differenziato per il suo valore semantico. Si tratta infatti del caso in cui l’utente per cui si sta facendo richiesta non sia registrato presso il servizio: il ServiceConsent non ha stato Active perché in realtà non ne è stato emesso alcuno. Al fine di controllare anche questa eventualità è opportuno inserire una eccezione nella classe MyDataUser, non ConsentManager, in quanto essa rientra all’interno delle sue responsabilità.

## Graphical User Interface

//grafico uml

Per la realizzazione dell’interfaccia grafica del gestore di dati personali ho scelto il design pattern Model View Controller. Questo viene impiegato anche all’interno di Most Likely Next Trip \cite{}, del quale si è utilizzata la schermata di funzionamento per il servizio omonimo.

Ho ritenuto necessario realizzare due finestre con le seguenti funzioni:

\begin{}

\item Una schermata per login e registrazione;

\item Una schermata per il profilo utente, che mostri i servizi attivi e abiliti, o disabiliti, determinati bottoni a seconda dello stato corrente dell’esecuzione.

\end{}

A ciò si deve aggiungere la schermata del servizio, che viene mostrata ad ogni sua invocazione.

Il Controller ha lo scopo di gestire la View e di permettere una comunicazione con il Model: in questo caso, esso si interfaccia quasi unicamente con la classe MyData, confermando il suo ruolo centrale di gestione.

La View mostra il profilo dell’utente loggato, permettendogli la gestione dei servizi a cui si è registrato. È possibile aggiungere nuovi servizi (funzione implementata tramite mock per mancanza di Service Linking), visualizzare le autorizzazioni emesse modificare il loro stato.

\fig view profilo, \fig view consent

Nella figura \ref{} vediamo il profile utente: dal menù a tendina in alto è possibile scegliere un servizio all’interno di quelli registrati; subito sotto sono presenti alcuni comandi per la gestione del suo stato (è possibile impostare i valori Active, Disabled, Withdrawn) e per l’emissione di nuove autorizzazioni in caso di necessità. È presente inoltre una casella di testo in cui sono mostrati tutti i ServiceConsent relativi al servizio selezionato, e tramite la pressione di un bottone si apre una nuova finestra che mostra i DataConsent relativi \ref{}. Infine, si trovano i bottoni per invocare il servizio e il Service Registry.

# Conclusioni e sviluppi futuri

In questa tesi si presenta un prototipo funzionante di gestore di dati personali, che permette di monitorare e gestire in tempo reale l’accesso ai dati da parte di servizi “consumatori”. Per la realizzazione del progetto si sono considerate le linee guida descritte all’interno dei documenti di specifica rilasciati dal team di sviluppo dell’università finlandese di Aalto, insieme alle relative implementazioni pubblicate su repository Github. Ulteriori spunti provengono dai protocolli e dai modelli citati all’interno di questi documenti: OAuth 2.0, UMA e il Consent Receipt Notice Specification.

Per testare l’applicazione si è scelto un servizio di previsione del prossimo viaggio più probabile, Most Likely Next Trip \cite{}, ma è possibile aggiungere nuove funzionalità grazie alla separazione fra infrastruttura e logica di business del singolo servizio.

Anche se il gestore di dati personali è stato realizzato in Java come applicazione monolitica, la modularità con cui sono stati sviluppati i componenti e le API di interazione fra di essi (gestione dei permessi, registrazione dei servizi, punto di policy enforcement) è tale da motivare, in un possibile progetto di estensione, la loro separazione in servizi indipendenti.

Un altro esempio di possibile sviluppo futuro comprende l’implementazione di un adeguato gestore delle politiche di sicurezza. Il SecurityManager presente all’interno del gestore di dati personali si occupa infatti solo dell’autenticazione reciproca di Source e Sink, fornendo un esempio di un possibile protocollo in collaborazione con il ConsentManager.

Ci sono tuttavia molti aspetti dell’implementazione che necessiterebbero di alcune modifiche a causa di problematiche di sicurezza: un esempio banale è la gestione delle password di autenticazione degli utenti. Altri esempi riguardano la sicurezza della memorizzazione dei dati, la garanzia di autenticità ed integrità dei Consent emessi e il logging delle operazioni svolte e degli eventi.

L’infrastruttura Java offre diversi strumenti disponibili per realizzare funzionalità in ambito di sicurezza, come ad esempio la Java Cryptography Architecture e il package security utilizzato al suo interno.

Un altro limite evidente del gestore di dati personali è dovuto all’assenza di un adeguato strato di persistenza. Questa problematica riguarda sia la gestione degli account che quella dei dati immagazzinati all’interno del Personal Data Storage. L’implementazione attuale consente l’utilizzo di file di testo per la simulazione di uno stato del sistema, utile anche per motivi di testing, che si limita ai soli dati contenuti all’interno del PDS.

Alcuni possibili miglioramenti sono stati anticipati all’interno della sezione \ref{4.5}: un esempio interessante potrebbe essere l’implementazione di uno strato di astrazione che si occupa della gestione dei dati in ingresso e in uscita dal Vault, modellandoli come InputStream e OutputStream utilizzabili a basso livello da input e output di tipo diverso (file, database, socket, ecc.).

Si deve tenere in considerazione infine che il progetto MyData è stato avviato e viene portato avanti separatamente dal team di sviluppo finlandese, pertanto rientrano all’interno di queste considerazioni anche gli aggiornamenti di specifiche e implementazioni pubblicate attraverso i repository \cite{stack} \cite{mobprof} \cite{jplan} \cite{mydatasdk}.

Durante il periodo di lavoro sulla tesi, sia di stesura che implementazione, sono stati rilasciati aggiornamenti che hanno fornito spunti di lavoro per l’arricchimento del gestore di dati personali, come ad esempio l’aggiunta delle specifiche sull’account utente. Alcune però non hanno trovato spazio all’interno dell’implementazione, come ad esempio una recente specifica sulla modalità di connessione fra Source e Sink.