Agenzia nazionale per le nuove tecnologie,
l'energia e lo sviluppo economico sostenibile

MINISTERO DELLO SVILUPPO ECONOMICO

Sviluppo di applicazioni basate sull'Ontologia

Michela Milano, Federico Chesani

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Report RdS/PAR2017/041

SVILUPPO DI APPLICAZIONI BASATE SULL'ONTOLOGIA

Michela Milano, Federico Chesani (Università di Bologna - Dipartimento di Informatica, Scienza e Ingegneria)

Settembre 2018

Report Ricerca di Sistema Elettrico

Accordo di Programma Ministero dello Sviluppo Economico - ENEA
Piano Annuale di Realizzazione 2017
Area: Efficienza energetica e risparmio di energia negli usi finali elettrici e interazione con altri vettori energetici
Progetto: D.6 Sviluppo di un modello integrato di smart district urbano
Obiettivo: Piattaforma ICT per la gestione di smart district
Responsabile del Progetto: Claudia Meloni, ENEA

Il presente documento descrive le attività di ricerca svolte all'interno dell'Accordo di collaborazione "*Sviluppo di applicazioni basate sull'Ontologia"*
Responsabile scientifico ENEA: Nicola Gessa
Responsabile scientifico UNIBO: Michela Milano

Indice

Sommario

Il presente documento è uno dei risultati del terzo anno del progetto del MiSE per lo sviluppo di un modello integrato di Smart District Urbano nel Piano Annuale di Realizzazione ENEA 2017 sulla Ricerca di Sistema Elettrico. L'idea centrale è la realizzazione di una piattaforma software in grado di raccogliere dati e dataset di un distretto urbano al fine di creare servizi che sfruttando queste informazioni possano incrementare l'efficienza e la qualità della vita di una Smart City.

Una delle azioni chiave per raggiungere tale risultato è la descrizione delle informazioni provenienti dai diversi ambiti organizzativi di un distretto come un edificio, un singolo appartamento o una strada. Per facilitare e uniformare lo scambio di informazioni è stata ipotizzata la realizzazione di uno strumento in grado di descrivere le informazioni da scambiare secondo una semantica condivisa. A tal fine negli scorsi anni è stata realizzata una ontologia sulla base delle tecnologie del web semantico, che permetta una uniformità di ricerca dei dati e una organizzazione uniforme delle conoscenze al fine di semplificare lo scambio degli stessi.

Intorno a questa ontologia sono stati poi costruiti tutta una serie di strumenti software in grado permettere l'accesso alle informazioni presenti nell'ontologia, di generare schemi di messaggi per lo scambio di informazioni e verificare che i messaggi da scambiare sulla piattaforma siano stati realizzati in maniera conforme alle definizioni delle informazioni. Infine, è stato realizzato un servizio web in grado di fornire un facile accesso alle precedenti applicazioni.

Questo documento descrive il lavoro svolto per realizzare le applicazioni costruite intorno all'ontologia a partire dalle librerie di accesso che erano già state definite negli anni precedenti. Ovvero, sono state realizzate le applicazioni per la generazione dei template dei messaggi da scambiare e per la loro validazione ed il servizio web che fornisce un comodo accesso alle applicazioni ed alle informazioni presenti nell'ontologia.

1 Introduzione

La keyword “Smart City” viene spesso usata per indicare una delle misure prioritarie per affrontare la problematica energetico-ambientale propria di una città, ovvero il luogo in cui si concentra il maggiore consumo di risorse energetiche poiché vi si concentra l’attività insediativa, produttiva e di massimo impatto sull’ambiente.

L’approccio Smart City mira, quindi, al raggiungimento di traguardi di abbattimento dei consumi energetici (in primo luogo elettrici) molto più consistenti di quelli ottenuti finora attraverso strategie basate essenzialmente sulla sostituzione di componenti con altri “a maggiore efficienza energetica”. Il principio organizzativo da utilizzare è quello del “resource on demand”. Tale approccio richiede però una tecnologia di sistema avanzata che coinvolge e integra: i) una sensoristica urbana e sistemi di interazione per comprendere esattamente la necessità dell’utente, ii) sistemi di trasmissione e raccolta integrata dei dati (cloud urbani), iii) sistemi a elevata intelligenza, per analisi, diagnostica, elaborazione e ottimizzazione che fondono dati provenienti da diversi canali informativi, e infine iv) servizi urbani capaci di adattare il proprio comportamento in base ai dati ricevuto da altri ambiti.

Il principale obiettivo di una Smart City sta nella capacità di mettere insieme gli elementi energetico-ambientali e quelli di carattere sociale (come la consapevolezza energetica, la partecipazione e coesione sociale e la qualità della vita) attraverso l’uso di tecnologie e di applicazioni e sfruttando l’interconnessione tra reti, ottenendo lo sviluppo di “servizi innovativi multifunzionali” partendo dalle conoscenze messe a disposizione degli enti. Ovvero attraverso la pubblicazione di dataset contenenti informazioni utili allo sviluppo di servizi grazie all’integrazione di dati precedentemente separati e non pubblici, e quindi allo sviluppo di nuovi servizi dalla creazione di nuova conoscenza derivante dall’integrazione delle diverse sorgenti.

Il presente lavoro si inserisce nel progetto D.6 - SVILUPPO DI UN MODELLO INTEGRATO DI SMART DISTRICT URBANO. Il principale obiettivo del progetto D.6 consiste nello sviluppo di un modello di “distretto urbano intelligente” che coniughi aspetti tecnologici e aspetti sociali, finalizzati al miglioramento dei servizi erogabili ai cittadini in quanto più efficienti dal punto di vista energetico e funzionale.

Nel distretto è prevista infatti l’implementazione di tecnologie e metodologie per la raccolta e distribuzione di informazioni per garantire questo approccio. La soluzione proposta prevede inoltre una combinazione tra tecnologie, modelli di business e coinvolgimento dei cittadini in un approccio innovativo di rigenerazione urbana.

L’attività si focalizza sullo sviluppo integrato di infrastrutture pubbliche urbane, sistemi per la modellazione e gestione della rete energetica del distretto (Smart District), sistemi centralizzati per l’analisi dei dati provenienti dalle abitazioni ed edifici con feedback all’utente per orientarlo a un uso consapevole (Smart home e Smart building) e sistemi di supporto alle decisioni per la valutazione del rischio del patrimonio edilizio e delle infrastrutture.

Obiettivo finale dell’attività è lo sviluppo di un modello di Smart District come distretto urbano intelligente in cui tutti i servizi di quartiere siano gestiti in maniera ottimale, sinergica e interoperabile. Grazie alla definizione e utilizzo di specifiche standard e tecnologie open, si agevolerà la replicabilità dei modelli sviluppati come primo step di una roadmap per la realizzazione delle Smart City; di fatto si realizzerà uno strumento a servizio delle amministrazioni locali e dei cittadini per evitare il locked-in dei vendors.

Uno degli obiettivi del progetto è lo sviluppo della Smart Platform in grado di raccogliere e integrare tra loro i dati dai diversi ambiti applicativi: Building Network, Lighting, Smart Home Network, sicurezza delle infrastrutture e coinvolgimento dei cittadini. La Piattaforma ICT deve essere interoperabile e aperta per la gestione delle infrastrutture fisiche di distretto attraverso l’integrazione di applicazioni verticali relative ai servizi di distretto. La Piattaforma ICT, denominata Smart Platform, è un livello software orizzontale, trasversale a tutte le applicazioni verticali da cui riceve i dati. Tali dati devono essere elaborati, forniti e resi fruibili dai diversi attori che interagiscono con il distretto (gestori, amministratori comunali, utenti).

All'interno di questo task si sviluppa il lavoro condotto dal Dipartimento di Informatica – Scienza e Ingegneria (DISI) dell'Università di Bologna. Il lavoro svolto consiste nell'analisi dei dati per poter definire uno strato di interoperabilità semantica che ha come scopo quello di semplificare l'implementazione di algoritmi intelligenti facilitando la selezione e aggregazione dei dati provenienti dai diversi ambiti applicativi.

Dal momento che i dati che confluiscono sulla piattaforma possono provenire da fornitori diversi che gestiscono parti diverse dell'infrastruttura e raccolgono informazioni provenienti da varie attività della città o del distretto e depositati in luoghi diversi, è stato previsto l'uso di uno strato semantico di interoperabilità basato su di un'ontologia. Questo è un requisito fondamentale per evitare di legare le amministrazioni cittadine, utenti ideali della piattaforma ICT di distretto, a tecnologie proprietarie di singoli fornitori. Come abbiamo detto, una Smart City può definirsi tale se gestisce e integra efficacemente informazioni che provengono da diversi ambiti applicativi della città. La combinazione di queste informazioni è spesso limitata dal modo in cui sono rappresentate tali informazioni e ai concetti a cui si riferiscono, non necessariamente interpretati in maniera univoca.

Lo scopo di un'ontologia è proprio quello di appianare tali differenze al fine di facilitare l'analisi delle informazioni permettendo una più facile elaborazione automatica, e in questo caso, col fine di progettare un'infrastruttura che limiti i tempi di sviluppo e al tempo stesso faciliti il lavoro di estensione dell'ontologia stessa per favorire una implementazione futura di nuovi servizi da integrare nella piattaforma della Smart City.

Oltre all'ontologia, sono necessari anche strumenti per estrarre le informazioni in essa presenti. Per questo motivo sono state sviluppate applicazioni che ne utilizzano le informazioni in essa contenute per costruire template dei messaggi usati per lo scambio di informazioni tra gli attori che operano sulla piattaforma e per validare la correttezza formale e semantica di tali messaggi. Inoltre, è stata realizzata un'applicazione web in grado di fornire accesso a questi strumenti, oltre che visualizzare il contenuto dell'ontologia e navigare i vari concetti.

Nel seguito del documento sono descritte le applicazioni realizzate. Nel capitolo 2 verranno descritti brevemente l'ontologia e la libreria di accesso, definita per integrare più facilmente l'ontologia in altri prodotti software, sviluppate negli anni precedenti di progetto. Nel capitolo 3 si presenteranno alcune delle principali tecnologie software utilizzate per questo progetto. Nei capitoli 4, 5 e 6 verranno presentate le applicazioni software rispettivamente per la generazione di template XML, generazione di file di validazione Schematron [1] ed un trasformatore XML-JSON. Nel capitolo 7 verrà presentata l'interfaccia web sviluppata che permette la visualizzazione delle informazioni presenti nell'ontologia e l'accesso alle applicazioni per le applicazioni presentate nei capitoli precedenti. Nell'ultimo capitolo verranno tratte le conclusioni del lavoro svolto.

2 Modello semantico di riferimento

In questo capitolo sono ripresi i concetti sviluppati durante gli anni precedenti del progetto. Nel capitolo successivo verrà presentato il lavoro svolto quest'anno.

2.1 Il modello dell'ontologia

Nel corso del primo anno del progetto, è stato definito il concetto di Urban Key Application Indicator (UKAI), quale indice aggregato che riassuma, su un certo periodo di tempo e ambito di spazio, informazioni significative su un aspetto ritenuto importante dal punto di vista della gestione dello Smart District (per esempio un indice di consumo energetico). Nel corso del secondo anno si è deciso di trasformare gli UKAI in dataset di informazioni relative alla Smart City da pubblicare sul web con il nome di Urban Dataset

In particolare (Figura 1):

- Il livello di aggregazione spaziale può andare dal singolo dispositivo (item) all'intera città;
- l'orizzonte temporale può andare dalla misura istantanea effettuata al valore medio sul breve-medio o lungo periodo, fino al valore anagrafico inviato una tantum per identificare una risorsa una volta per tutte;

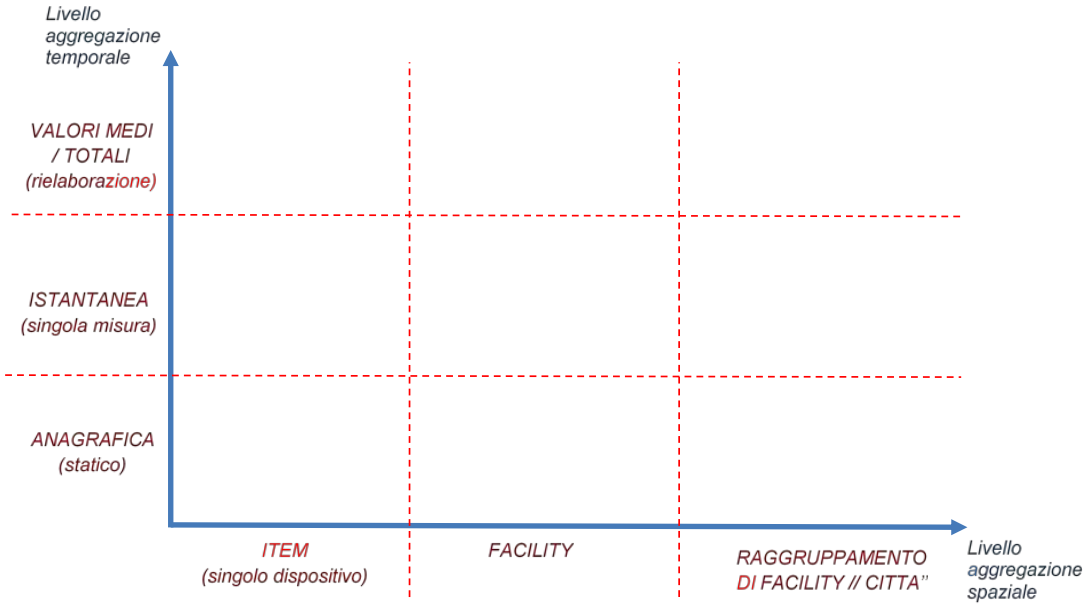

Figura 1. Classificazione degli Urban Dataset in base ad aggregazione spaziale e temporale

L'idea di partenza è di dare alle amministrazioni pubbliche uno strumento per monitorare i servizi affidati a fornitori. Supponiamo, per esempio, che un comune emetta il bando per la gestione dell'illuminazione pubblica. Potrebbe definire degli Urban Dataset per verificare il consumo energetico e i livelli d'illuminazione e richiederne il valore medio sulle ore di accensione, aggregate a livello di strada. Queste dovrebbero essere inviate al comune tramite la piattaforma orizzontale, che esporrà un Web Service REST a cui inviare tali dati in un determinato formato (per esempio JSON o XML).

Dunque:

- Tramite tali Urban Dataset, il comune potrà verificare che l'azienda rispetti i limiti di consumo energetico senza tenere le strade scarsamente illuminate.
- Al momento della scadenza del contratto di gestione dell'illuminazione pubblica e dei dati relativi, il comune, rifacendo la gara, potrà richiedere, nel bando, all'azienda vincitrice di fornire i dati allo stesso modo: dunque l'amministrazione pubblica potrà riutilizzare la stessa piattaforma, senza ricostruirla da zero, ma allo stesso tempo permettere all'azienda subentrante

di realizzare la propria infrastruttura informatica come preferisce a patto che vengano rispettate le specifiche di comunicazione con la piattaforma.

- Se altre città adottassero lo stesso sistema, la stessa utility potrebbe partecipare al bando, senza dover ricostruire la propria infrastruttura software.

Estendendo il ragionamento precedente, se si adottasse il sistema per le varie utility:

- L'amministrazione presenterebbe un'interfaccia unica verso tutti i fornitori.
- Le varie applicazioni potrebbero scambiarsi dati, tramite la piattaforma orizzontale.

Per esempio, se s'implementasse sui lampioni un sistema di monitoraggio del traffico, per regolare la luminosità sulla base delle necessità, questi dati potrebbero essere inviati all'applicazione di monitoraggio del traffico, gestita da un altro soggetto.

Per raggiungere questi scopi, non è sufficiente definire il singolo indice su cui effettuare le analisi ed eventualmente prendere decisioni, ma occorre definire anche la rappresentazione ontologica del modello dati e dei vari Urban Dataset che permettano, in modo semplice, di passare da un formato a un altro, di costruire una serie di strumenti che facilitino l'inserimento dei dati nei vari formati e, comunque, di avere una comprensione semantica del singolo Urban Dataset.

Gli Urban Dataset sono, quindi, strutture dati che trasmettono informazioni legate a uno specifico dominio applicativo, pensati per fare da tramite fra applicazioni verticali e l'applicazione orizzontale di gestione della città (Smart City Plaform) o fra diverse applicazioni verticali (per esempio singoli sensori/dispositivi ma anche piattaforme di gestione locale di un dominio che elaborano i dati grezzi e forniscono servizi).

Figura 2. Scambio di Urban Dataset fra applicazioni verticali

Precedentemente si era pensato di fare in modo che l'ontologia contenesse anche le informazioni degli Urban Dataset, cioè tutti i dati generati dai sensori e dai sistemi che elaborano informazioni a partire dai dati dei sensori. Una scelta del genere, data la mole di dati prevista in ingresso sarebbe stata insostenibile da gestire da parte di un sistema basato su tecnologie del web semantico quali RDF e SPARQL. Per questo si è deciso che i dati vengano memorizzati in database esterni all'ontologia. Lo schema di riferimento dello scambio degli Urban Dataset è quello mostrato in Figura 2**Errore. L'origine riferimento non è stata trovata.**. Un applicativo v erticale usa il formato dati (sia esso JSON, XML) per inviare verso la Smart City Platform gli Urban Dataset e questa lo memorizza nel proprio DB e/o lo invia a uno o più applicativi connessi a essa.

Nel precedente schema, gli Urban Dataset possono essere definiti, coerentemente con le definizioni di aggregazione spaziale e temporale:

- A livello di dispositivo/sorgente (detto, in questo progetto, anche livello di campo): questi saranno tipicamente Urban Dataset di Item in Real-Time. Per esempio, il sensore di una Smart Street segnala il numero di macchine parcheggiate in un determinato momento.
- All'interno della piattaforma locale, ovvero i dati aggregati nelle piattaforme locali, per la gestione interna del contesto applicativo (con protezione dei dati a tutela della privacy). Saranno Urban Dataset con aggregazione spaziale a livello di facility, su un tempo medio. Per esempio, il numero medio auto nella fascia oraria 17-18 dei giorni lavorativi in una particolare strada monitorata dalla piattaforma Smart Street.
- A livello dell'Area Smart City Platform, ovvero quei dati elaborati nelle piattaforme locali per descrivere un aspetto a livello di distretto o città: Per esempio, numero medio auto nella fascia oraria 17-18 dei giorni lavorativi in tutte le strade monitorate del distretto dalla piattaforma smart street.

2.1.1 Concetti principali

L'ontologia che è stata definita ruota intorno al concetto di Urban Dataset. Con tale concetto si riferisce ogni dato, aggregato o meno, che un contesto applicativo è in grado di elaborare a partire dai dati raccolti dai sensori dislocati nello Smart District. Questo è il nodo nevralgico di tutta la comunicazione e l'informazione principale che deve essere scambiata all'interno dell'infrastruttura.

Di seguito sono spiegate le varie parti che compongono l'ontologia. Il tutto è accompagnato da grafici che ne mostrano le relazioni. In tali immagini le parti in grassetto sono state definite per l'occasione mentre le altre sono pezzi di ontologie importate che completano e aiutano la sua definizione.

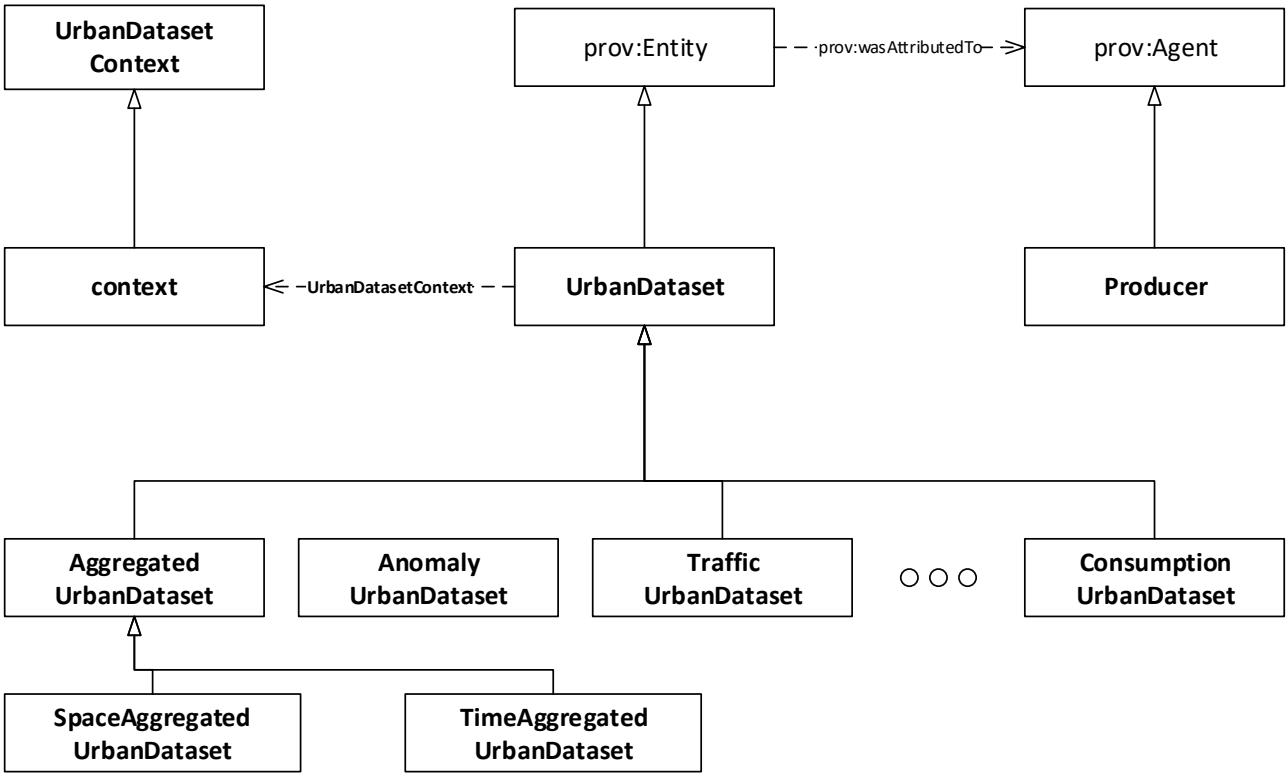

Figura 3. Schema dei principali concetti relativi agli Urban Dataset

Un aspetto importante di un set di dati è la sua provenienza, cioè chi è il fornitore dei dati. PROV Ontology [2], ontologia che fornisce un set di classi, proprietà e restrizioni usate per rappresentare e scambiare informazioni di provenienze generate in diversi sistemi e diversi contesti, in questo caso svolge questa

funzione. Nel caso specifico, *Producer* (Figura 3) descrive l'ente a cui è demandata l'attività di creazione del dato.

Un Urban Dataset (Figura 3), quando viene generato, viene associato ad altre informazioni statiche che lo descrivono, come la sua descrizione o il suo identificativo. Per questo motivo sono definite come data property informazioni specifiche come il nome, la descrizione, un URI di riferimento e la versione, ovvero la **specificationRef**. Ogni Urban Dataset necessita di altre informazioni di contesto che servono per descrivere meglio le informazioni e per caratterizzarle, ovvero informazioni come produttore, posizione geografica a cui fanno riferimento i dati, lingua usata per le descrizioni, tempo e fuso orario.

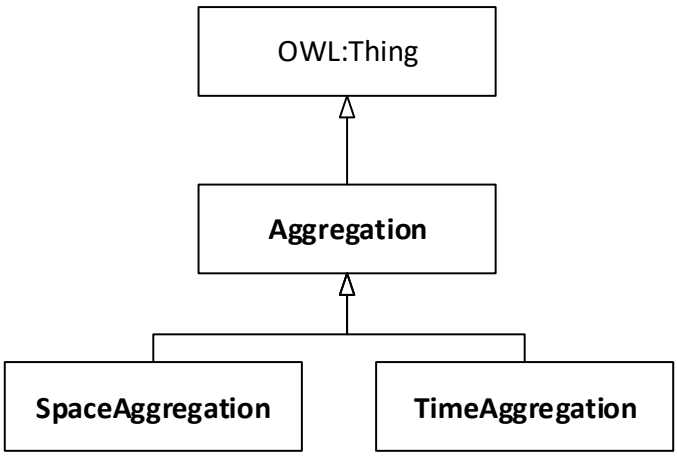

Figura 4. Schema dei concetti per definire il tipo di aggregazione

Uno stesso Urban Dataset può essere aggregato in spazio e tempo in modo diverso. È stata ipotizzata la possibilità di definire dei livelli di aggregazione predefiniti e creare delle istanze diverse di Urban Dataset in funzione delle modalità di aggregazione utilizzate. In particolare, sono state definite le classi *Space* e *TimeAggregatedUrbanDataset* (Figura 3), un Urban Dataset con diversa aggregazione è definito come istanza di queste sottoclassi e riferisce, attraverso una proprietà, l'aggregazione usata e definita come istanza della classe *Aggregation* (Figura 4). Ad esempio, se si volesse definire un Urban Dataset che fornisce un livello di aggregazione settimanale dei parcheggi liberi in una strada, si può definire un nuovo Urban Dataset che fa riferimento all'Urban Dataset base e al livello di aggregazione "average" definito come istanza di TimeAggregation come mostrato nell'esempio seguente:

```
<owl:NamedIndividual
rdf:about="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#AverageFreeParking">
        <rdf:type
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#TrafficUrbanDataset"/>
        <HasUrbanDataset
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#FreeParking"/>
        <context
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#context1"/>
        <hasAggregation
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#average"/>
        <hasApplicationContext
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#SmartMobility"/>
```

```
        <hasUrbanDatasetProperty
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#FreeParkNum"/>
        <prov:wasAttributedTo
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#SmartStreet"/>
        <UrbanDataset_ID
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">UrbanDataset
D5b</UrbanDataset_ID>
    </owl:NamedIndividual>
```

Qui sopra è mostrato un estratto di RDF dove si definisce una istanza *AverageFreeParking* in cui si fa riferimento all'aggregazione (proprietà *hasAggregation*) e al tipo (proprietà *type*) di Urban Dataset.

Gli Urban Dataset che sono frutto di aggregazione sono comunque definiti come istanze di Urban Dataset. Viene poi specificato che la classe *TimeAggregatedUrbanDatasetAnomaly*, sottoclasse di *TimeAggregatedUrbanDataset* ha una equivalenza con le istanze che hanno la proprietà *hasUrbanDataset* definita una sola volta con una istanza di *TrafficUrbanDataset* e possiede la proprietà *hasAggregation* definita una sola volta con una istanza di *TimeAggregation*. In questo modo è possibile aggiungere facilmente nuove possibilità di aggregazione che possono valere per tutti gli Urban Dataset già definiti o ancora da definire e avere già una gerarchia pronta per questo uso. Questa logica è stata anche usata per la definizione delle unità di misura con prefissi all'interno dell'ontologia OM che è stata integrata in questa ontologia.

2.1.2 Concetti di supporto

Le singole proprietà a cui sono associati i diversi Urban Dataset sono a loro volta delle istanze di sottoclassi di *Property* (Figura 5). Property ha due sottoclassi:

- *ContextProperty*: raggruppa tutte le istanze di proprietà usate per descrivere il contesto (es. coordinate, lingua, tempo in cui è stato raccolto e inviato il dato, ecc.);
- *UrbanDatasetProperty*: raggruppa tutte le istanze di proprietà usate per descrivere proprietà specifiche degli Urban Dataset (es. il numero di anomalie, il numero di parcheggi occupati, l'assorbimento energetico medio, ecc.).

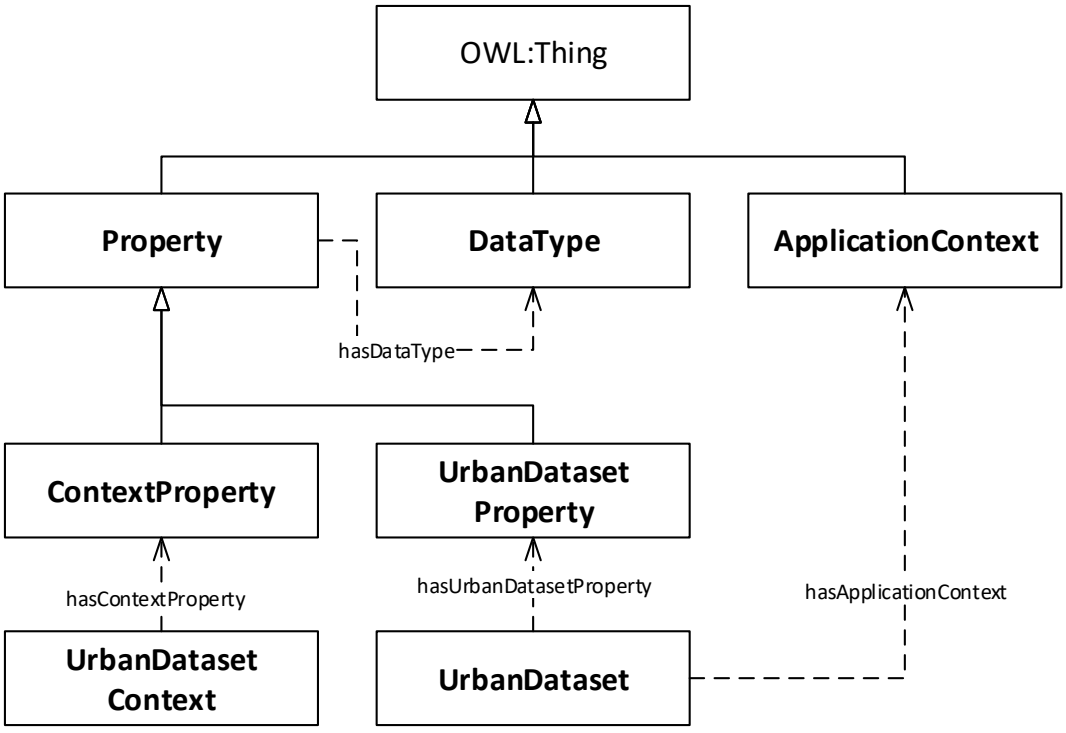

Figura 5. Concetti di supporto

È possibile anche creare proprietà composte, ovvero è possibile che una proprietà sia composta a sua volta da altre proprietà. Ciò è descritto semplicemente creando una istanza di proprietà che ha come uno o più riferimenti ad altre istanze di proprietà tramite le object property *ContextProperty* e *UrbanDatasetProperty*.

A ogni proprietà sono state associate il *DataType*, ovvero il formato del dato usato per rappresentare l'informazione, ovvero un numero intero, un numero reale, una string o un timestamp, tramite la object property *hasDataType*.

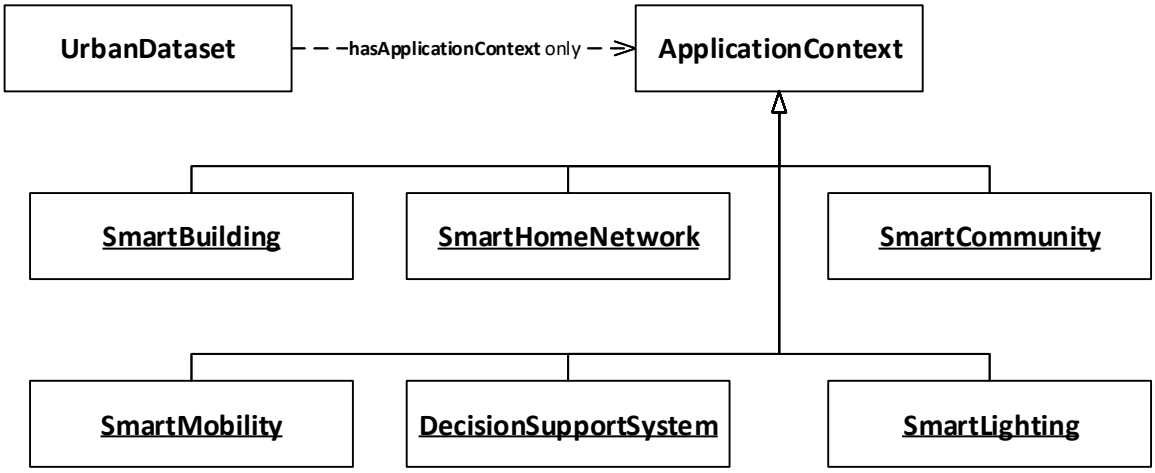

Figura 6. Associazione tra UrbanDataset e ApplicationContext

Un Urban Dataset contiene un'aggregazione di dati provenienti da sensori di tipi specifici e perciò tali informazioni sono ascrivibili a un particolare contesto applicativo. Per questo motivo, è stata prevista l'esistenza di una proprietà che metta in relazione una particolare istanza di *UrbanDataset* con un contesto applicativo. In Figura 6 sono definite le istanze (indicate con testo sottolineato) che un *ApplicationContext* può assumere attraverso la relazione *hasApplicationContext*. Inoltre, è specificato dall'ontologia che un *UrbanDataset* può essere associato esclusivamente a un solo *ApplicationContext* esplicitato tramite la dicitura *only.*

Di seguito è specificata l'istanza della proprietà FreeParkNum dell'esempio precedente dove si vedono l'unità di misura e il tipo di dato utilizzato:

```
<owl:NamedIndividual
rdf:about="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#FreeParkNum">
    <rdf:type
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#UrbanDatasetProperty"/>
    <om-2:hasUnit rdf:resource="http://www.ontology-of-units-of-
measure.org/resource/om-2/Number"/>
    <hasDataType
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#integer"/>
    <rdfs:comment>numero parcheggi liberi</rdfs:comment>
</owl:NamedIndividual>
```

Di seguito è mostrata una istanza di contesto utilizzata dagli Urban Dataset dove sono mostrate le proprietà di contesto usate:

```
<owl:NamedIndividual
rdf:about="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#context1">
        <rdf:type
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#UrbanDatasetContext"/>
        <hasContextProperty
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#coordinate"/>
        <hasContextProperty
rdf:resource="http://www.semanticweb.org/disi/ontologies/2017/8/smart-city-
ontology#language"/>
</owl:NamedIndividual>
```

2.2 *La libreria di interfacciamento*

La caratteristica principale che ha una libreria software è quella di nascondere delle complessità all'utilizzatore finale. Ovvero di semplificare l'accesso a delle capacità di elaborazione che altrimenti lo sviluppatore dovrebbe imparare autonomamente a richiedere attraverso l'apprendimento di interfacce di accesso o altri linguaggi. Il nostro caso fa riferimento proprio a quest'ultima situazione. Per interrogare un'ontologia sviluppata con le tecnologie del web semantico è necessario conoscere il linguaggio di interrogazione SPARQL. Dal momento che una stessa interrogazione è molto probabile che sia rivolta più di una volta, ha senso avere delle query predeterminate che possono essere richiamate con un comando più semplice preso da una lista definita a priori.

Questa modalità di funzionamento non obbliga chi vuole accedere alle informazioni contenute nell'ontologia a imparare un nuovo linguaggio di interrogazione e per questo risulta più facile integrare queste query all'interno di software più grandi e con uno scopo diverso ma che necessitino delle informazioni presenti nell'ontologia.

Un altro obiettivo è, quindi, quello di fornire una copertura completa per l'accesso alle informazioni presenti nell'ontologia. A partire dalla struttura dell'ontologia definita, si è partiti con l'ideazione di una serie di richieste che permettessero di accedere a tutte le informazioni presenti senza conoscere la struttura e da punti di partenza diversi (es. conoscendo il soggetto o l'oggetto di una relazione). La libreria, quindi, deve permettere di recuperare le informazioni conoscendo solo in parte ciò che vuol sapere e per questo risulta molto importante un buon grado di facilità di esplorazione a partire dalle interfacce fornite.

Un obiettivo necessario riguarda la possibilità di accedere alle informazioni dell'ontologia in remoto. I sistemi che accettano interrogazioni SPARQL operano su protocolli web per cui è necessario che la libreria sviluppata sia in grado di raggiungere su http il server che contiene l'ontologia.

Un altro vincolo importante riguarda l'utilizzo di Java come linguaggio di programmazione. Si prevede che questa libreria sia sviluppata per essere integrata all'interno di altre applicazioni. Java è uno dei linguaggi più diffusi e conosciuti. Per garantire una facilità di utilizzo della libreria si è optato per questa scelta.

2.2.1 Descrizione dell'architettura

Partendo dalle caratteristiche appena descritte si è arrivati alla definizione dell'architettura. Per poter recuperare le informazioni presenti nell'ontologia, essa deve essere inserita all'interno di un server che accetti richieste di interrogazione ed essendo tecnologie sviluppate dal W3C, le richieste vengono portate al server attraverso protocollo web http. Per affrontare tale vincolo e ottenere già un supporto alle astrazioni dei linguaggi SPARQL e RDF si è deciso di utilizzare una libreria già sviluppata per tali scopi. A questo proposito si è scelto di utilizzare Apache Jena [3]. Questa libreria fornisce una serie di strumenti utili per interagire con un server SPARQL e fornisce un supporto per memorizzare ed elaborare in locale dataset in formato RDF. Il software in questione è sviluppato da diversi anni ed è ancora supportato e in evoluzione. Questo garantisce che il prodotto abbia una buona maturità e affidabilità e una garanzia di evoluzione e supporto futuri sufficiente, oltre che necessaria per l'utilizzo in un sistema complesso e duraturo come quello oggetto del progetto del MiSE di cui questa attività è parte, da poter essere usata come base per la nostra libreria.

Un'ulteriore garanzia deriva dalla diffusione che ha Apache Jena, infatti risulta essere una delle più usate in ambiente Java per questo tipo di attività.

La libreria sviluppata, denominata SmartCityOntology Library, si interfaccia, quindi, con Apache Jena fornendo un accesso semplificato alle sue operazioni. Tale accesso è costruito in maniera specifica per l'ontologia definita in precedenza. Dalla Figura 7 si può notare, attraverso la rappresentazione grafica, che la libreria definita ha come compito quello di nascondere la complessità dell'accesso a un sistema SPARQL e per farlo sfrutta le operazioni messe a disposizione da Apache Jena.

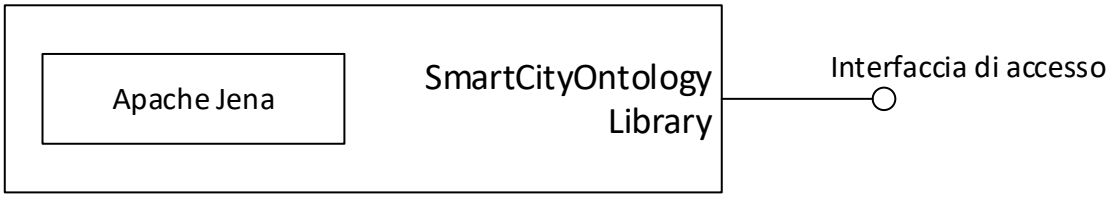

Figura 7. Architettura della libreria

La libreria deve svolgere una funzione piuttosto semplice, ovvero prevedere un set di query pronte da usare per interrogare la base di dati. Un approccio in cui si preparano solamente delle query dentro dei metodi potrebbe essere veloce all'inizio ma difficile da estendere. Per cui si è pensato di organizzare la sua architettura su più livelli in modo da poter riusare le funzionalità dei livelli sottostanti per creare più facilmente interrogazioni e funzionalità più complesse.

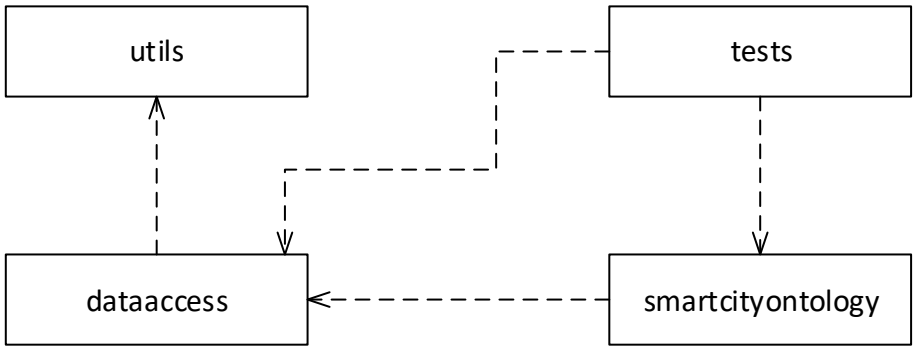

Figura 8. Diagramma dei package

In Figura 8 è mostrato il diagramma dei package. Nel package *utils* sono state inserite le informazioni per la configurazione della libreria stessa e sono stati creati degli scheletri di query per costruire dinamicamente l'interrogazione. Nel package *dataaccess* sono state inserite le classi per effettuare la connessione al sistema remoto e per costruire interrogazioni che, appoggiandosi sulle funzionalità del package utils, recupera le informazioni di una generica ontologia a partire da parametri specifici. In particolare, in questo package sono presenti metodi per poter ricostruire la struttura dell'ontologia e per questo potrebbe essere utilizzata anche per altre ontologie diverse dalla nostra. Naturalmente questo è ottenuto nascondendo della complessità ma allo stesso tempo limitando i gradi di libertà. Il package *smartcityontology*, infine, è quello che contiene le operazioni per recuperare le informazioni specifiche per l'ontologia definita in precedenza. Sono stati implementati anche una serie di test automatici per verificare con il comportamento della libreria sia corretto.

Andando nel dettaglio: è prevista una classe che si occupa del collegamento al database remoto attraverso il pattern Singleton. Al momento dell'uso della libreria viene creata in automatico la connessione verso il server definito all'interno di un file di configurazione. Naturalmente è possibile modificare il server remoto anche durante l'utilizzo, ma è possibile interrogare solo un server alla volta. Questa scelta è stata fatta per evitare di dover creare una diversa istanza di connessione ogni volta che si accede a un server diverso e per evitare di dover passare come riferimento la classe di connessione a ogni richiesta. Questa scelta si è basata sul fatto che, essendo pensata per una specifica ontologia, si presuppone che venga usata per interrogare l'ontologia presente su un particolare server che fornisce la funzione di indice per i dataset.

Un'altra scelta è stata quella di avere un sistema che costruisse le query utilizzando più livelli per rendere più facile l'esenzione nel caso l'ontologia dovesse subire delle modifiche come delle aggiunte di nuovi concetti o una variazione della struttura.

Sono state previste anche delle query che permettono di limitare il numero di risultati recuperati dal server. Potrebbe essere semplice effettuare un filtro, attraverso la libreria o da parte dell'utilizzatore, dei dati recuperati dal server in caso di non necessità della lista completa. Questo, però, potrebbe causare dei problemi di sovraccarico del server e rallentare il recupero delle informazioni. Infatti, nel caso in cui la query dovesse ottenere come risposta una lista molto lunga di dati, causerebbe un rallentamento nel recupero e nell'invio degli stessi, per cui si è riportato anche nella documentazione la preferenza a richiedere che sia il server stesso a limitare l'output, invece che il software che integra la libreria. In questo modo si può ottenere un carico minore sul server e riuscire a gestire più richieste contemporanee.

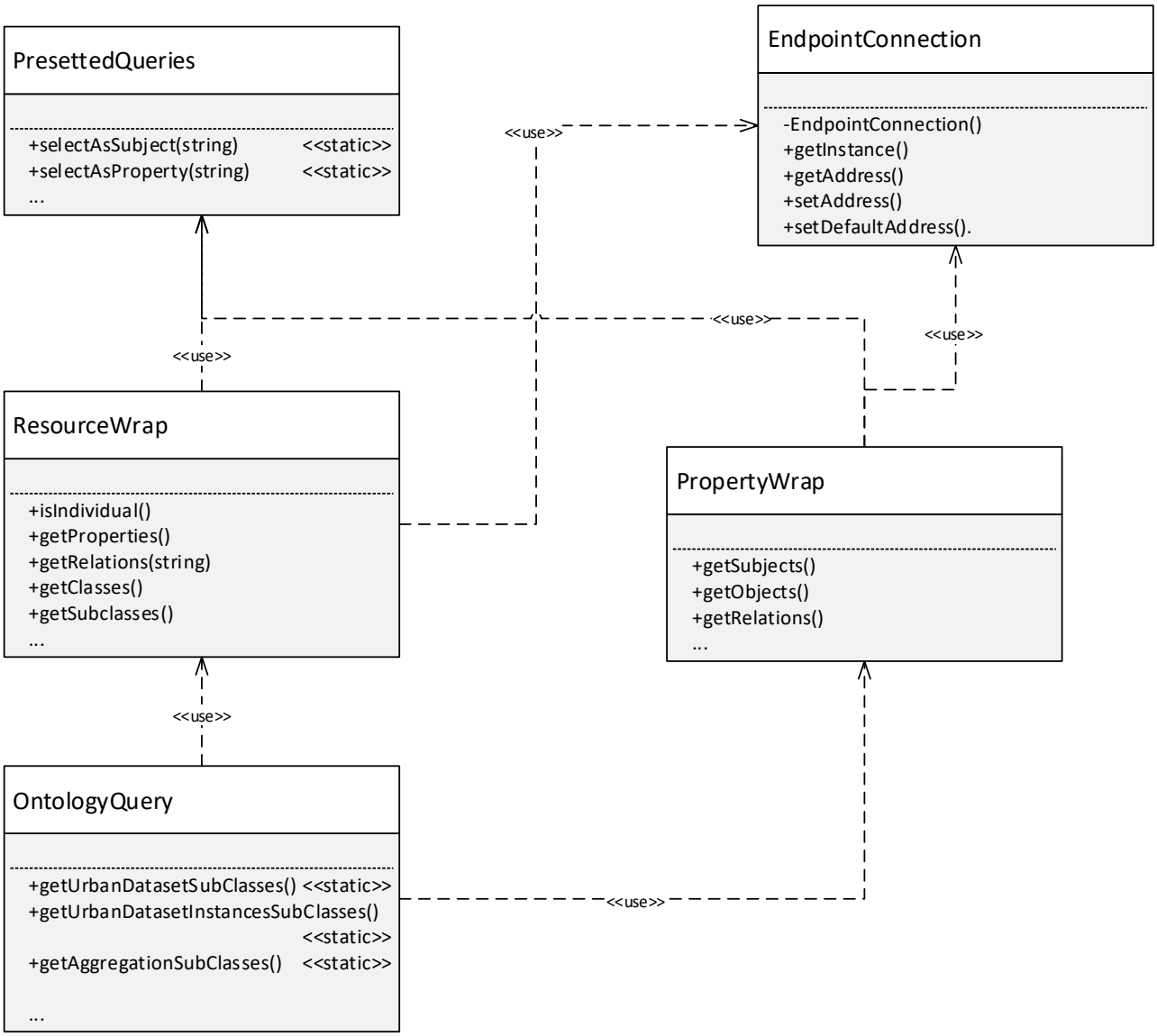

Figura 9. Diagramma delle classi

In Figura 9 è mostrato un diagramma delle classi parziale della libreria. Sono mostrati solamente alcuni dei metodi presenti. Si può notare che le query principali specifiche per l'ontologia sono definite dentro la classe *OntologyQuery*. Le classi *ResourceWrap* e *PropertyWrap* sono delle classi pensate per fungere da wrapper rispettivamente per le classi *Resource* e *Property* di Apache Jena. Aggiungono metodi per verificare se una risorsa è una istanza o meno, recuperare le sue sottoclassi o le sue istanze e le relazioni in cui sono coinvolte

all'interno dell'ontologia. Queste due classi mantengono il riferimento al server remoto nascondendolo all'esterno. Naturalmente il riferimento al server può essere recuperato dall'esterno per modificare l'indirizzo del server durante l'esecuzione. Come già detto, la classe *EndpointConnection* gestisce la connessione alla prima esecuzione prendendo l'indirizzo da un file di configurazione che può quindi essere modificato dall'utente prima dell'esecuzione ed evita di cablare nel codice l'indirizzo del server che gestisce le richieste garantendo flessibilità.

Infine, è stata creata una documentazione che descrive le diverse operazioni, per facilitarne l'uso e l'integrazione all'interno di altri software.

3 Le nuove applicazioni

Partendo dall'ontologia, sono stati definite alcune applicazioni che utilizzano le informazioni in essa contenute per facilitare lo scambio di informazioni sulla piattaforma.

In particolare, sono state realizzate le applicazioni:

- per la generazione di un **template di messaggio XML** contenente i dati di un Urban;
- per la generazione di un **file Schematron** con lo scopo di validare la correttezza semantica di un messaggio XML contenente le informazioni di un Urban Dataset;
- per la **conversione di un file XML** contenente un Urban Dataset, e relativi dati raccolti sul campo, in formato JSON, e viceversa da JSON a XML. Naturalmente i formati di tutti questi file sono stati definiti precedentemente e la generazione rispetta tali specifiche.

Inoltre, è stata realizzata un'applicazione web per la visualizzazione dei dati contenuti nell'ontologia e per richiedere attraverso una interfaccia comoda all'utente la generazione dei template realizzati dalle applicazioni descritte in precedenza. Ovvero, una interfaccia che sfrutta le applicazioni sviluppate in questo anno di progetto per generare automaticamente i template che servono per lo scambio di informazioni sulla piattaforma.

In questo capitolo sono descritti in dettaglio gli obiettivi generali delle applicazioni. Nei capitoli successivi verranno descritte nel dettaglio le applicazioni e il loro comportamento.

3.1 Caratteristiche generali

Come già detto in precedenza, lo scopo della piattaforma è lo scambio di informazioni tra i diversi attori della Smart City. Naturalmente, affinché questa comunicazione sia efficacie, è necessario che la comunicazione avvenga tenendo conto di specifiche di formati ben definiti.

A questo scopo sono stati identificati due formati di scambio dati di riferimento, XML e JSON, e definite le opportune specifiche che determinano la struttura e la sintassi dei messaggi di scambio dati. Nello specifico, per i messaggi in formato XML, la specifica è costituita da un XML Schema; per i messaggi in formato JSON, la specifica è costituita da un JSON Schema.

Alla struttura definita dalle specifiche del formato, però, vanno associate delle regole sul contenuto ammesso per alcune particolari sezioni dei messaggi e dipendente dallo specifico UrbanDataset. Lo scopo delle applicazioni di generazione dei template è proprio quello di creare degli artefatti che includono sia informazioni sulla struttura dei messaggi, ma anche informazioni sul loro contenuto (definito nell'ontologia). In questo modo il software sviluppato funge da anello di collegamento tra i due mondi (ontologia e formato dei dati) dal momento che allo stato attuale non esistono strumenti e regole standard che permettono di definire questo collegamento.

Nei prossimi capitoli sono descritti in dettaglio i formati e le relazioni con i dati contenuti nell'ontologia e come sono state implementate le applicazioni software e l'applicazione web realizzate durante questo anno di lavoro sul progetto.

3.2 Tecnologie utilizzate

Per la realizzazione di tali software sono state usate librerie esterne sia per facilitare lo sviluppo e sia per utilizzare librerie e architetture già testate e validate. Tale metodologia permette di sfruttare il lavoro di progettazione e validazione su parti del codice e di affidare una parte del componente software che si va a sviluppare al lavoro di validazione e verifica fatta dalla comunità di sviluppatori che ha deciso di utilizzare tale libreria. A questo scopo, le librerie esterne che sono state identificate sono diverse per affrontare diversi problemi. In particolare, i problemi individuati che possono essere demandati ad altre librerie riguardano prima di tutto la gestione della struttura dei file degli Urban Dataset da generare automaticamente.

La libreria di query per l'ontologia sviluppata lo scorso anno di progetto è stata fondamentale per recuperare le informazioni e costruire i file per l'invio di dati sulla piattaforma. Questa libreria si è rivelata molto utile e comoda grazie anche alle astrazioni per proprietà ed entità costruite per l'interrogazione di una generica

ontologia. A partire da tali astrazioni è stato possibile costruire astrazioni specifiche per il concetto di Urban Dataset e altri ad esso collegati che ha permesso un più rapido sviluppo dei software presentati qui.

Inoltre, dal momento che le applicazioni dovevano essere eseguibili da linea di comando, si è deciso di sfruttare la libreria Apache Commons-CLI [4] che fornisce le primitive per configurare, accettare e controllare i parametri che possono essere passati da riga di comando all'applicazione in modo da decidere un particolare flusso di esecuzione per ottenere il risultato richiesto.

Per la parte web, si è deciso di sfruttare uno dei principali framework di sviluppo web per JAVA. La scelta è ricaduta su Spring Framework 5 [5] perché, oltre ad essere uno di più usati, permette una integrazione semplice di pagine web statiche e generate dinamicamente con applicativi JAVA di base permettendo di costruire semplicemente un collegamento per l'invocazione delle altre applicazioni sviluppate tramite pagina web. In questo modo si è riusciti a costruire un sito web che riesce a recuperare le informazioni sull'ontologia tramite la libreria di accesso ed invocare le applicazioni di generazione di template tramite semplici link web e chiamate REST.

4 Template di messaggio di un Urban Dataset

In questo capitolo è descritto il lavoro svolto per la realizzazione del software responsabile della generazione di template XML a partire dei dati dell'ontologia conformemente al XSD che definisce il formato.

4.1 Caratteristiche funzionali

La prima caratteristica richiesta dalle specifiche per questo software è la possibilità di generare automaticamente un template di messaggio codificato in XML secondo le caratteristiche di uno specifico Urban Dataset così come è definito all'interno dell'ontologia.

La caratteristica di un generatore di template è quella di rendere semplice, idealmente con un semplice comando o gesto, recuperare uno schema da una sorgente di informazioni. Da questa descrizione, ipoteticamente, si potrebbero generare staticamente tanti template quanti sono i possibili casi e ritornare all'utente quello specificato al momento della richiesta. Naturalmente una tale soluzione mancherebbe di flessibilità. Ovvero, nel momento in cui si dovesse aggiungere un nuovo Urban Dataset, come nel nostro caso, bisognerebbe modificare il codice e generare il nuovo template staticamente. Per questo motivo sono state sviluppate una serie di classi che modellano i concetti principali identificati all'interno del documento XSD per rendere flessibile la generazione dei documenti.

Un'altra caratteristica è la necessità di poter essere invocata in due modi diversi. Ovvero, deve poter essere chiamata sia singolarmente, cioè da riga di comando, che integrata nel servizio web. Quindi deve prevedere una struttura adattabile a questa doppia natura.

4.2 Descrizione dei requisisti

Il formato XML del file di messaggio è composto da tre parti principali:

```
<UrbanDataset>
    <specification/>
    <context/>
    <values/>
</UrbanDataset>
```

ovvero da **Specification**, **Context** e **Values**.

La sezione **Specification** descrive la struttura generale dell'Urban Dataset. In essa sono elencate tutte le proprietà che lo compongono e le eventuali sotto-proprietà di ciascuna proprietà. Oltre a ciò sono presenti anche l'identificativo, il nome e l'URL (che lo identifica univocamente l'UrbanDataset all'interno dell'ontologia), come mostrato nel seguente frammento:

```
    <specification version="1.0">
        <id> </id>
        <name> </name>
        <uri> </uri>
        <properties>
            <propertyDefinition> </propertyDefinition>
        </properties>
    </specification>
```

Nel frammento sono evidenti le parti appena citate. Le diverse proprietà sono descritte tramite una serie di campi all'interno del tag *propertyDefinition*.

All'interno del tag *propertyDefinition* sono presenti dei campi specifici per la definizione delle singole proprietà:

```
<propertyDefinition>
    <propertyName> </propertyName>
    <propertyDescription ></propertyDescription>
    <dataType> </dataType>
    <codeList> </codeList>
    <unitOfMeasure> </unitOfMeasure>
</propertyDefinition>
```

Ciascuno di questi campi ha un corrispettivo all'interno dell'ontologia. Per cui è possibile fare un mapping uno ad uno a meno di casi particolari. Tra i casi particolari c'è da annoverare il caso delle proprietà con sotto-proprietà. Queste non possiedono *dataType*, *codeList* e *unitOfMeasure*, ma al loro posto un campo *subProperties* con i nomi delle proprietà che sono poi definite nella stessa lista delle proprietà. Di seguito un esempio della struttura:

```
<properties>
    <propertyDefinition>
        <propertyName>PropA</propertyName>
        <propertyDescription> </propertyDescription>
        <subProperties>
            <propertyName>subPropA</propertyName>
            <propertyName>subPropB</propertyName>
        </subProperties>
    </propertyDefinition>
    <propertyDefinition>
        <propertyName>subPropA</propertyName>
        <propertyDescription> </propertyDescription>
        <dataType> </dataType>
        <unitOfMeasure> </unitOfMeasure>
    </propertyDefinition>
</properties>
```

Come si vede dall'esempio, La proprietà **PropA** ha due sotto-proprietà e di seguito nella stessa lista **properties** sono definite le sotto-proprietà (nello schema ne è mostrata solo una per motivi di sintesi).

Tra le proprietà con sotto-proprietà c'è il caso particolare delle proprietà speciali *period* e *coordinates*. Queste hanno delle proprietà che devono essere presentate in un ordine particolare. Il problema è che nei grafi RDF con cui è disegnata l'ontologia non c'è una relazione di ordinamento, per cui l'ordine con cui vengono ritornate le sotto-proprietà è casuale per semantica delle operazioni degli store RDF. Per questo è necessario prevedere un sistema per ordinare tali proprietà.

La sezione Context è così definita:

```
<context>
    <producer> </producer>
```

```
    <timeZone> </timeZone>
    <timestamp> </timestamp>
    <coordinates> </coordinates>
    <language> </language>
</context>
```

Da come si può osservare, deve essere composto da cinque parti: *producer*, *timeZone*, *timestamp*, *coordinates* e *language*. Si tratta di una struttura statica che non è influenzata dall'ontologia ed è parte del formato. Quando il template deve essere generato, ciascun campo deve contenere delle informazioni generiche che devono essere poi settate dal gestore dei dati che vuole inviare un messaggio, relativo all'Urban Dataset descritto, sulla piattaforma.

La sezione *values* è composta da una o più *line* ognuna delle quali contiene un set di coppie proprietà/valore rilevato (campi *property/@name* e *val*); ad ogni *line* può essere associato un numero d'ordine (campo *id*), come mostrato nel seguente frammento:

```
<values>
    <line id="1">
        <description> </description>
        <timestamp> </timestamp>
        <coordinates> </coordinates>
        <period>
            <start_ts> </start_ts>
            <end_ts> </end_ts>
        </period>
        <property name="PropA">
            <property name="subPropA">
                <val> </val>
            </property>
            <property name="subPropB">
                <val> </val>
           </property>
        </property>
        <property name="PropB">
            <val> </val>
        </property>
    </line>
</values>
```

All'interno di ogni line sono, inoltre, definiti alcuni campi opzionali: *description*, *timestamp*, *coordinates* e *period.* La loro presenza dipende dalla definizione dell'Urban Dataset. In particolare, se all'interno della definizione nell'ontologia dell'Urban Dataset sono presenti le relazioni con queste proprietà speciali, allora in questa sezione sono mostrati secondo la struttura appena mostrata.

Tutte le altre proprietà specifiche sono mostrate di seguito a queste proprietà speciali e contengono il tag *val* che contiene il valore rilevato dal sistema e oggetto della trasmissione dati. Le proprietà che contengono

sotto-proprietà, a differenza del caso della sezione *Specification*, contengono direttamente un livello delle sotto-proprietà con il valore da trasmettere.

Un altro requisito importante è la possibilità di avere una doppia modalità di esecuzione di queste applicazioni, sia da riga di comando che attraverso interfaccia web. Per poter ottenere questo risultato, è stata realizzata una interfaccia di invocazione per la generazione del template. Questa interfaccia è la porta di accesso che sarà usata dalle due diverse modalità di invocazione per eseguire l'operazione di generazione del template di un Urban Dataset.

4.3 Descrizione dell'architettura

Stando ai requisiti appena presentati, è evidente che l'applicazione dovrà interagire con due sistemi software: un servizio web e una interfaccia di invocazione da CLI (Command Line Interface). Potrebbe essere possibile considerare questi due software che interagiscono con l'applicazione in oggetto come degli attori che interagiscono col software. Per questo motivo, in Figura 10 sono presentati come attori sia l'interfaccia da riga di comando che il servizio web che fanno in modo di eseguire l'applicazione per la generazione dei template.

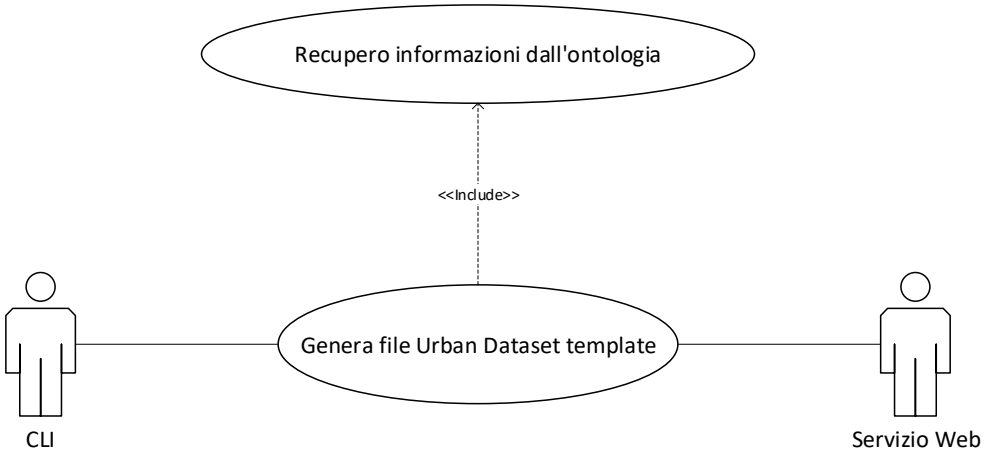

Figura 10. Diagramma dei casi d'uso per la generazione di template XML

Come si può osservare dall'immagine sempre in Figura 10, il caso d'uso principale è, ovviamente, la generazione del template secondo le specifiche del file XSD. Naturalmente non è possibile generare il template senza avere a disposizione le informazioni da inserire. Per questo è necessario recuperare le informazioni dall'ontologia attraverso l'interrogazione del server SPARQL dove essa risiede. In Figura 10 tale caso d'uso è indicato come incluso nel caso d'uso della generazione del template perché è una condizione necessaria per la compilazione del template e deve quindi essere sempre presente.

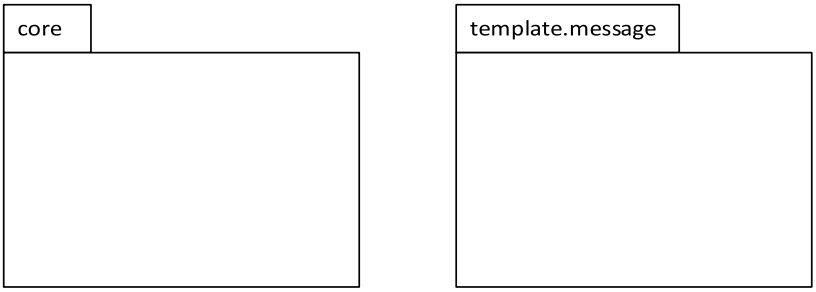

Figura 11. Organizzazione dei package

Il software è stato diviso in due package, come mostrato in Figura 11. Il package *template.message* contiene le classi che svolgono le funzioni necessarie per la creazione del template XML dell'Urban Dataset. Mentre il package *core* contiene le classi che identificano dei concetti principali che saranno usati anche nelle applicazioni presentate in seguito.

Per prima cosa sono state definite le astrazioni che riguardano la generazione del documento XML finale. Dal momento che il documento XML è definito da tre macro-sezioni, sono state individuate tre classi per rappresentare tali astrazioni, ovvero *Specification*, *Context* e *Values* come mostrati in Figura 12. Dal momento che queste classi si occupano di generare sezioni di documento, sono accomunati da un obbiettivo comune e pertanto si è provveduto a definire una interfaccia comune tra queste classi e che quindi la implementano. L'interfaccia definita si chiama *IMessageTemplate* e definisce il metodo *text()*. L'obiettivo di tale metodo è di generare la sezione di XML che compete alla classe partendo da un riferimento ad un documento XML su cui si sta operando. Le informazioni utili per il completamento delle diverse sezioni sono passate alle varie istanze di classe al momento della rispettiva creazione degli oggetti.

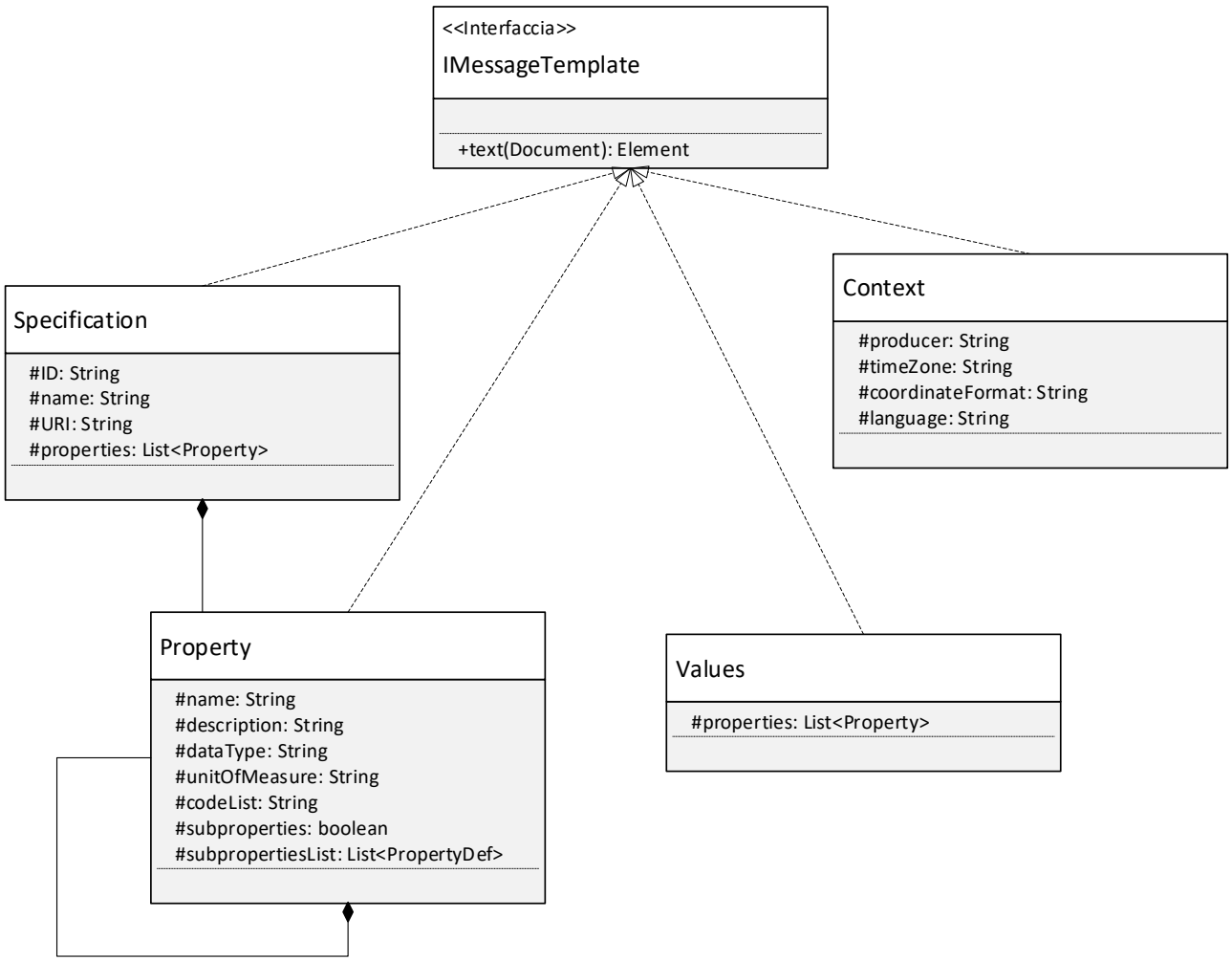

Figura 12. Diagramma delle classi relativo alla generazione delle sezioni XML

Oltre alle tre sezioni principali, è stata individuata anche un'altra sezione del documento XML come candidata ad essere mappata come una nuova entità e quindi un oggetto specifico, ovvero le diverse sezioni *property* all'interno della sezione *specification*. Anche questa classe implementa l'interfaccia *IMessageTemplate* e si occupa lei stessa di generare la parte di XML che le compete. La particolarità è che, essendo possibile avere sotto-proprietà, ogni proprietà ha un riferimento alle sue sotto-proprietà. Di conseguenza la richiesta di generazione di una delle sezioni di proprietà scatenerà una richiesta a tutte le sotto-proprietà di generare la parte di XML che compete a ciascuna di esse. Stesso discorso vale per l'istanza di *Specification* che si occupa della sezione *specification* del documento XML. Questa istanza ha un riferimento alla lista di proprietà che la compongono. Quando verrà invocato il metodo *text()*, che genera la sua sezione di XML, dovrà provvedere a richiedere alle sue proprietà di generare la relativa parte di XML che verrà poi composta del documento finale.

Nel package *core*, sono presenti due classi (Figura 13). La prima è l'interfaccia *IBuilderTemplate* che fornisce tre metodi per costruire un generico template. Tali metodi salvano su file di testo (in un caso passando come parametro il path) o su variabile stringa il risultato della creazione del template per l'Urban Dataset. La classe *UrbanDataset* rappresenta il concetto di Urban Dataset e fornisce i metodi per accedere alle informazioni presenti nell'ontologia. Il tutto è fatto attraverso la libreria di accesso sviluppata nel precedente anno di progetto. Come si può notare dalla Figura 13, nella classe è presente il riferimento all'oggetto *ResourceWrap* che è una classe appartenente alla libreria sviluppata nel precedente anno e che permette di recuperare informazioni sulle proprietà di una risorsa di un'ontologia.

Grazie al riferimento a quest'oggetto è possibile eseguire le operazioni definite per la classe *UrbanDataset*. In particolare, *getProperties()* ritorna tutte coppie proprietà-oggetto di cui l'Urban Dataset è soggetto. Mentre *getSpecificProperties()* ritorna tutte le coppie per cui la proprietà è specifica e definita all'interno dell'ontologia sviluppata per il progetto, ovvero *hasUrbanDatasetProperty* e *hasContextProperty*. In realtà la struttura di ritorno è una Map dove ad ogni proprietà corrisponde una lista di proprietà in quanto in ambito RDF è possibile associare più volte una proprietà anche con diversi valori ad una stessa risorsa.

Il metodo *getSpecificSubproperties()* ritorna, invece, la lista di sotto-proprietà delle proprietà *hasUrbanDatasetProperty* e *hasContextProperty* sottoforma di list. Accetta come parametro un campo booleano per indicare se la ricerca deve essere ricorsiva alle sotto-proprietà o no. Il metodo *getFirstLevelPropName()* ritorna la lista di proprietà escluse le proprietà speciali *coordinate*, *period* e *DataDescription*.

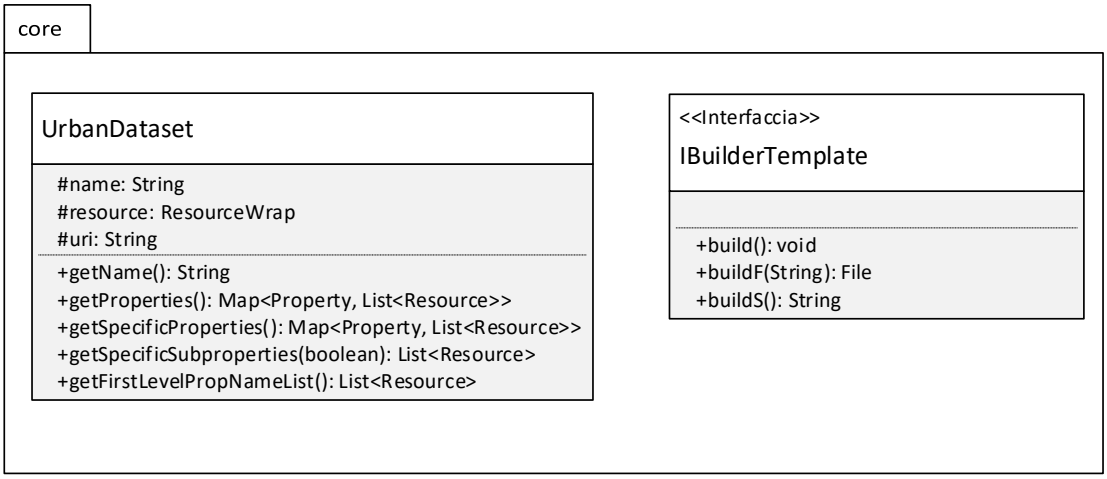

Figura 13. Diagramma delle classi del package *core*

In Figura 14 è presente il diagramma delle classi completo per il package *template.message*. Da tale grafico si può notare che è presente la classe *MessageBuilder* e che questa svolge un ruolo centrale. Essa implementa l'interfaccia *IBuilderTemplate* ed è responsabile del recupero delle informazioni dall'ontologia. Per far questo, istanzia un oggetto *UrbanDataset* a partire dal nome dell'Urban Dataset di cui si vuole costruire il template XML. Tramite i metodi privati *templateProperties()* e *templateIDProperties()* si occupa di recuperare rispettivamente la struttura delle proprietà, sottoforma di lista di oggetti *PropertyDef*, di cui l'Urban Dataset è soggetto e le altre informazioni necessarie a costruire lo schema, come l'ID, l'URI e la versione. Questi dati vengono poi usati dal metodo privato *buildGeneral()*, invocato in origine da una delle *build()*, risultato delle implementazioni dell'interfaccia, per costruire un oggetto *UDTemplate*. Questo oggetto non è altro che un contenitore di tutte le informazioni necessarie per generare il file o la stringa oggetto della trasformazione. Infatti, esso contiene il riferimento all'oggetto che mappa la radice del documento XML, ovvero *DOMSource*, ed un oggetto *Trasnformer*, proprio della libreria base di JAVA che ha lo scopo di serializzare un oggetto di tipo DOMSource in una stringa di testo. Tale oggetto *Transformer* contiene già una configurazione dell'output da generare ma che può essere modificata a seconda di chi la deve usare. In questo caso verranno usate dai metodi *build()* per serializzare il risultato in un file di testo o in una variabile di tipo stringa.

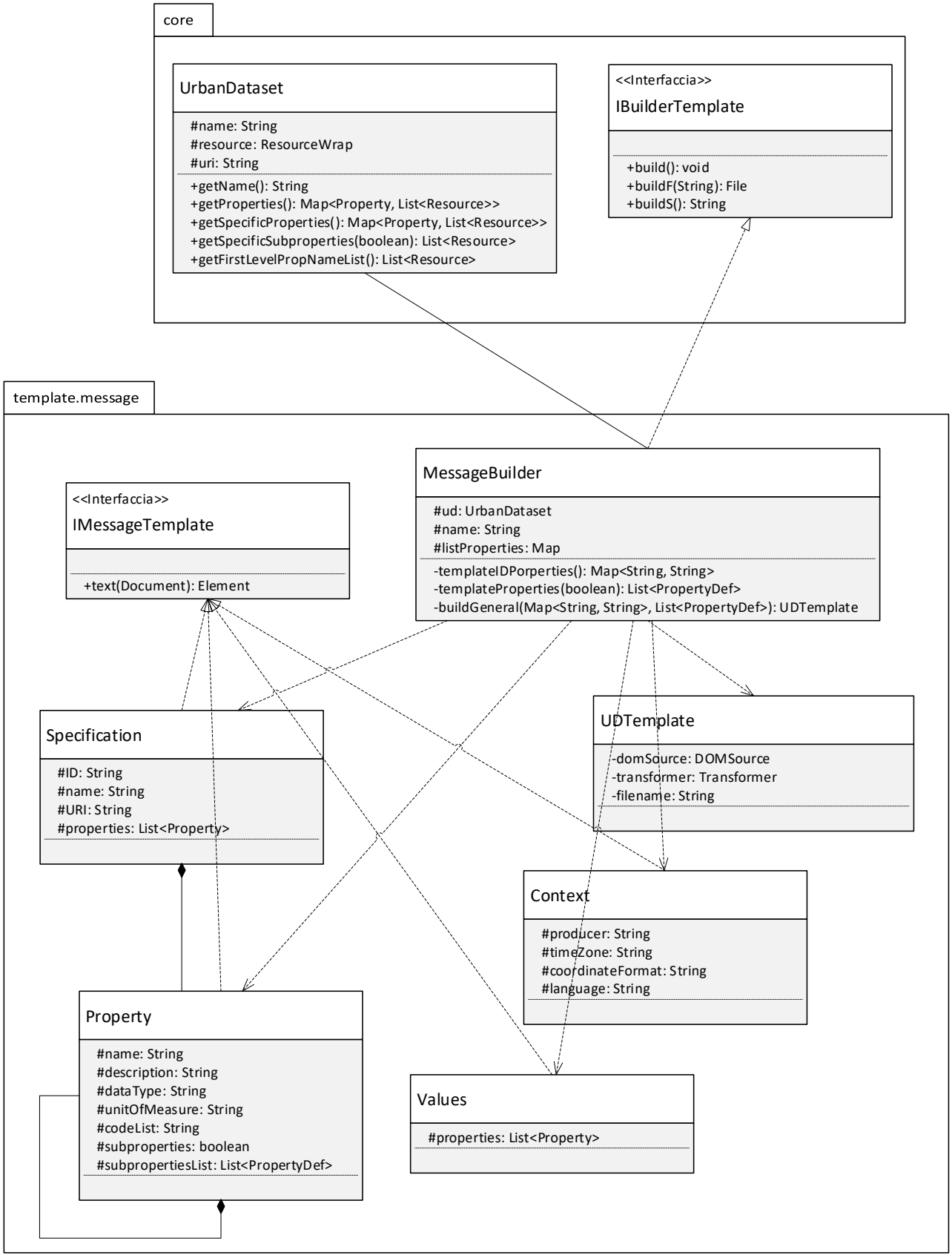

Figura 14. Diagramma delle classi del package *template.message*

L'oggetto istanza di *MessageBuilder*, prima di effettuare la serializzazione dovrà, ovviamente, provvedere a costruire la struttura di oggetti che fa capo ad *IMessageTemplate*. Per fare ciò, l'oggetto istanza di

MessageBuilder si occupa di costruire gli oggetti istanze di *Specification*, *Context*, *Values* e *Property*. Ognuno di essi provvederà a costruire la sezione del messaggio che gli compete prendendo come parametri di costruzione i dati necessari che contengono le informazioni per la corretta costruzione del documento XML. Tra queste informazioni c'è la struttura delle proprietà specifiche dell'Urban Dataset di cui necessita *Specification*. Per questo viene costruita una lista che contiene queste informazioni, istanze della classe *Property*, e viene passata all'istanza di *Specification* o all'istanza di *Property* di cui è sotto-proprietà per completare l'operazione.

Una volta costruite le singole parti del template XML, l'istanza di *Transformer* provvede ad estrarre il contenuto dell'oggetto istanza di DOMSource in cui si è costruito il documento e a generare il file formattato contenente il testo del documento XML con la struttura attesa.

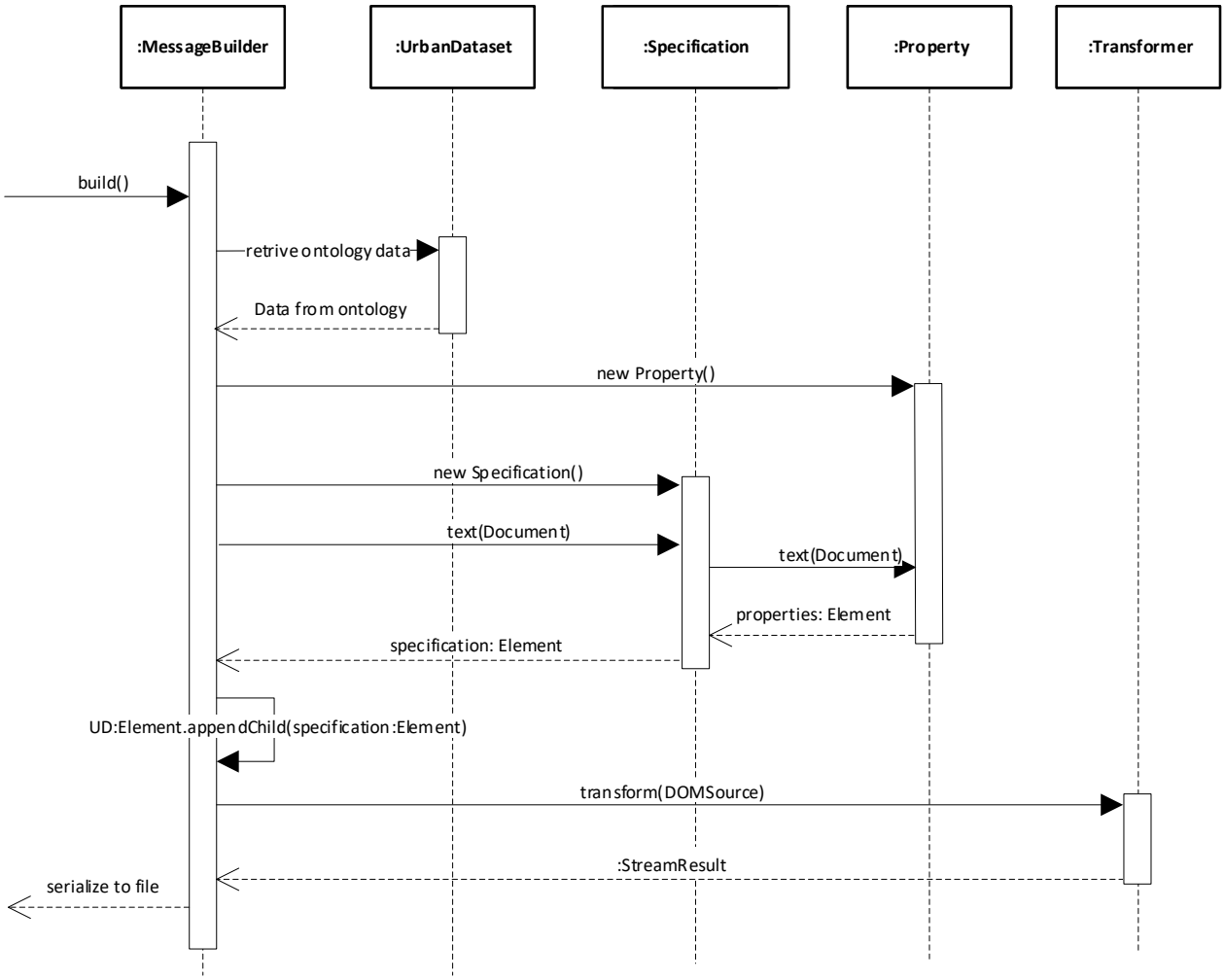

Figura 15. Diagramma di sequenza della creazione di un template XML

In Figura 15 è mostrato il diagramma sequenza delle principali azioni che vengono svolte durante la creazione di un template XML. La richiesta di *build()* di un template XML è causa prima di tutto del recupero delle informazioni dall'ontologia ed in seguito della creazione di diversi oggetti *Property* quante sono le proprietà dell'Urban Dataset. Tali informazioni vengono reperite attraverso l'invocazione dei metodi *getProperties()* e *getSpecificSubproperties()* dell'oggetto istanza di *UrbanDatset*.

Dopo di che vengono create le sottosezioni del file XML ovvero Specification, Context e Value a partire dalle classi *SpecificationSection*, *ContextSection* e *ValueSection* (in Figura 15 è mostrato solo il caso di *Specification* che a sua volta delega la sezione delle proprietà al relativo metodo di ciascun *Property*). Una volta riunite le

tre sottosezioni nell'oggetto radice istanza di DOMSource, l'esecuzione dell'applicazione continua col richiedere all'istanza di *Transformer* della libreria standard di JAVA di recuperare la stringa di testo corrispondente al DOM costruito.

Come già detto, per poter eseguire l'applicazione è necessario un sistema di interfacciamento con la CLI. Per questo, è stata creata una interfaccia di invocazione utente che accetta alcuni parametri. In particolare, dal momento che si è ipotizzato l'uso di indicare più di un Urban Dataset di cui generare i template, è stata prevista la possibilità sia di indicare un Urban Dataset singolarmente oppure una lista da file di testo contenente un Urban Dataset per riga. Nel primo caso, l'esecuzione deve essere richiesta eseguendo il programma con il parametro ***-U*** (oppure ***–Urbandataset***) seguito dal nome dell'Urban Dataset. Al contrario, per indicare una lista di Urban Dataset da elaborare è possibile aggiungere il parametro ***-F*** (oppure ***–file***) seguito dal nome del file di testo contenente la lista degli Urban Dataset.

L'output dell'applicazione viene generato all'interno della cartella specificata dall'utente preceduta dal parametro ***-P*** (oppure ***--path***) ed ogni file avrà il nome dell'Urban Dataset di cui contiene il template.

5 Schemi di validazione di Urban Dataset

Schematron è un linguaggio di validazione a regole per identificare la presenza o l'assenza di pattern all'interno di documenti XML. Si tratta di schemi espressi a loro volta in XML e, nella sua tipica implementazione, il processo di validazione consiste nell'effettuare una serie di trasformazioni XSLT. Il risultato di questa operazione è a sua volta un file XML che conterrà le indicazioni sugli eventuali errori presenti nel file XML iniziale di cui si cercava la validazione.

Quindi, mentre la validazione con XSD verifica la correttezza strutturale e sintattica del documento, la validazione attraverso Schematron verifica la presenza di pattern nella struttura e quindi può essere usato per effettuare una validazione di tipo semantico. Esempi di controlli effettuati attraverso Schematron sono: verificare la presenza di particolari strutture in caso siano presenti particolari tag opzionali all'interno del file XML, verifica sul contenuto di alcuni elementi oppure verificare la presenza di alcuni attributi di un elemento in particolari situazioni.In questo capitolo è descritto il lavoro svolto per la realizzazione del software responsabile della generazione del file di validazione Schematron per verificare che il file XML, contenente i dati dell'UrbanDataset da trasferire sulla piattaforma di comunicazione, rispetti alcune regole semantiche sulla sua struttura.

5.1 Caratteristiche funzionali

Come nel caso del template XML mostrato nel capitolo precedente, le caratteristiche richieste a questo software sono relative alla capacità di generare automaticamente un file Schematron per ciascun Urban Dataset, presente e futuro, che implementi le specifiche di rappresentazione delle informazioni definite all'interno dell'ontologia.

Anche in questo caso, è richiesta la capacità di integrazione dell'applicazione all'interno di un servizio web (oltre all'uso da riga di comando) per far sì che sia possibile generare un nuovo Schematron relativo ad un Urban Dataset con pochi semplici click del mouse.

Nella prossima sezione è mostrato in dettaglio il formato del file e le operazioni richieste per sua la compilazione.

5.2 Descrizione dei requisiti

L'obiettivo di questo software è la generazione di un file Schematron. Un file di questo tipo è un insieme di regole sulla struttura di un documento XML che sono decise a priori. Per cui tali regole sono comuni ad una serie di file generati. Nel nostro caso sono state definite diverse regole che valgono per ogni Urban Dataset. Per rendere la generazione più immediata, all'interno dello Schematron sono state definite una serie di variabili che sono usate nelle regole definite anch'esse nello Schematron. Di conseguenza l'unica differenza tra i diversi Urban Dataset è relativa alla composizione e valorizzazione delle variabili.

Le variabili definite all'interno del file possono essere classificate fondamentalmente in due gruppi: il primo con valori singoli che devono essere associati a particolari campi all'interno del documento XML e il secondo da liste di valori che indicano i diversi valori che devono essere in particolari strutture del documento XML per poter terminare con successo la validazione.

Dal momento che l'unica cosa che cambia tra gli Schematron per i diversi Urban Dataset sono queste variabili si è deciso di definire staticamente il file Schematron da generare; l'unica cosa che cambia è il valore che deve essere associato alle variabili e per fare questo si è deciso di identificare le posizioni per le sostituzioni attraverso parole chiave di cui fare la sostituzione sulla base delle specifiche dell'Urban Dataset definite nell'ontologia. Di seguito sono presenti alcuni estratti dallo Schematron base.

Per prima cosa sono presenti i campi che devono contenere valori singoli come il nome, l'URI o il numero di proprietà presenti.

```
<pattern abstract="false" id="$$udID">
    <!-- sostituire la stringa $$udName con il nome dell'UD -->
    <let name="UDname" value="'$$udName'"/>
```

```
<!-- sostituire la stringa $$udURI  con l'URI dell'UD -->
<let name="UDuri" value="'$$udURI'"/>
```

In questo primo gruppo mostrato sono state evidenziate le stringhe de sostituire con i diversi parametri. Si nota subito che le stringhe per la sostituzione hanno la caratteristica di essere precedute dai simboli *$$* per una più facile identificazione. Dal momento che si tratta di una implementazione interna, si è scelto di identificare in questo modo per pura comodità.

Le parti importanti del documento sono inframezzate da commenti che in output non dovranno esserci. La rimozione dei commenti, usati solamente in fase di sviluppo, diventa un fattore importante per l'output.

```
<!-- sostituire la stringa $$udPropN con il numero di proprieta', anche ripetute (ad es. la proprieta' C che e' sottoproprieta' di A e B deve essere conteggiata due volte), che compongono l'UD (sono INCLUSE nel conteggio eventuali sottoproprieta'; se eventulmente presenti, sono ESCLUSE dal conteggio period, coordinates e le loro sottoproprieta') -->
<let name="propertyNum" value="$$udPropN"/>

<!-- sostituire la stringa $$udPropDefN con il numero di proprieta' distinte che compongono l'UD (sono INCLUSE nel conteggio eventuali sottoproprieta' e period, coordinates e le loro sottoproprieta', se presenti. Eventuali sottoproprieta' ripetute devono essere conteggiate una sola volta) -->
<let name="propertyDefNum" value="$$udPropDefN"/>
```

Continuando con il documento, vediamo che ci sono da compilare due campi che contengono il numero di proprietà. Nel primo caso le sotto-proprietà possono essere ripetute ma non devono esserci proprietà speciali (come indicato anche nel commento). Nel secondo caso, il numero deve considerare proprietà distinte, quindi senza ripetizioni, ma contando anche le proprietà speciali se presenti.

```
<!-- sostituire la stringa $$udPropNameList con la lista dei nomi delle proprieta' che compongono l'UD, separati da spazio. Eventuali proprieta' ripetute vanno inserite una sola volta. Se la proprieta' e' COORDINATES, aggiungo di default alla lista le proprieta' format, longitude, latitude, height. Se la proprieta' e' PERIOD, aggiungo di default alla lista le proprieta' start_ts, end_ts.-->
<let name="propNameList" value="'$$udPropNameList'"/>

<!-- sostituire la stringa $$udFirstPropNameList con la lista dei nomi delle proprieta' di primo livello che compongono l'UD, separati da spazio. Sono escluse dalla lista coordinates e period -->
<let name="firstLevelPropNameList" value="'$$udFirstPropNameList'"/>
```

Queste altre due variabili devono contenere sequenze di proprietà. La prima corrisponde alla lista di ***$$udPropN*** ovvero alla seconda variabile osservata nel blocco precedente, la seconda mostra le proprietà di primo livello, quindi senza sotto-proprietà, ad esclusione delle proprietà speciali.

Nelle seguenti variabili vanno inseriti degli array di proprietà, ognuno con le sue peculiarità. Siccome questi array contengono il nome della proprietà e le informazioni ad esse associate (ognuna in array distinti, quindi organizzati per tipologia di informazione), gli array devono contenere le informazioni nel giusto ordine perché l'indice dell'elemento nell'array consente il collegamento con l'elemento corrispondente in un altro array.

```
    <!-- sostituire la stringa $$udPropNameSet con la lista dei nomi delle proprieta' e
sottoproprieta' che compongono l'UD, separati da virgola spazio. Eventuali sottoproprieta'
ripetute devono comparire una sola volta -->
    <let name="propNameSet" value="tokenize('$$udPropNameSet', '[,\s]+')"/>

    <!-- sostituire la stringa $$udDataTypeSet con la lista dei tipi di dato relativi alle
proprieta' dell'UD, separati da virgola spazio. Se la proprieta' e' composta da
sottoproprieta', assegnare al dataTypeSet il valore 'complex-numeroSottoprieta', dove
numeroSottoproprieta e' un intero pari al numero delle sottoproprieta' della
    proprieta' complessa -->
    <let name="dataTypeSet" value="tokenize('$$udDataTypeSet', '[,\s]+')"/>

    <!-- sostituire la stringa $$udUmSet con la lista delle unita' di misura relative alle
proprieta' dell'UD, separati da virgola spazio. Se la proprieta' e' composta da
sottoproprieta', assegnare a udUmSet il valore 'adimensionale' -->
    <let name="umSet" value="tokenize('$$udUmSet', '[,\s]+')"/>

    <!-- sostituire la stringa $$codeList con la lista degli eventuali riferimenti a
codelist associati alle proprieta' dell'UD, separati da virgola spazio. Se alla proprieta'
non e' associata una codelist, assegnare in valore 'null' -->
    <let name="codeListSet" value="tokenize('$$codeList', '[,\s]+')"/>

    <!-- sostituire la stringa $$udSubPropNameSet con una lista che avra' tanti valori
quante sono le coppie "proprieta' composta - sua sottoproprieta'". Ogni valore e' la
concatenazione di  "nome della proprieta' composta" e "nome della sua sottoproprieta'
(tutto attaccato, senza caratteri separatori) -->
    <let name="subPropNameSet" value="tokenize('$$udSubPropNameSet', '[,\s]+')"/>
```

Come si può vedere anche dai commenti alle varie sezioni, gli array devono essere definiti con la virgola e spazio come separatore e non devono esserci ripetizioni di sotto-proprietà. In caso di proprietà con sotto-proprietà, il tipo di dato deve essere indicato come *complex* e avere come unità di misura *adimensionale*. Nel caso dell'ultima variabile, sono contenute tutte le sotto-proprietà usate e inserite come concatenazione di proprietà padre e proprietà figlia.

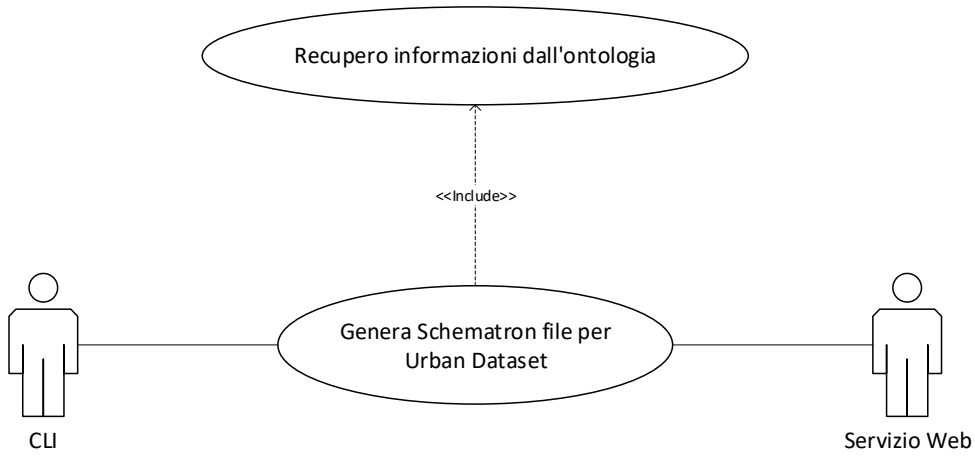

Figura 16. Diagramma dei casi d'uso per la generazione di file Schematron

5.3 Descrizione dell'architettura

Così come il generatore di template visto in precedenza, l'applicazione dovrà interagire con due sistemi software: un servizio web e una interfaccia di invocazione da CLI. Per questo lo schema dei casi d'uso risultante è molto simile al precedente (Figura 16).

Anche in questo caso si tratta di recuperare le informazioni dall'ontologia sull'Urban Dataset richiesto per organizzare queste informazioni nel formato richiesto.

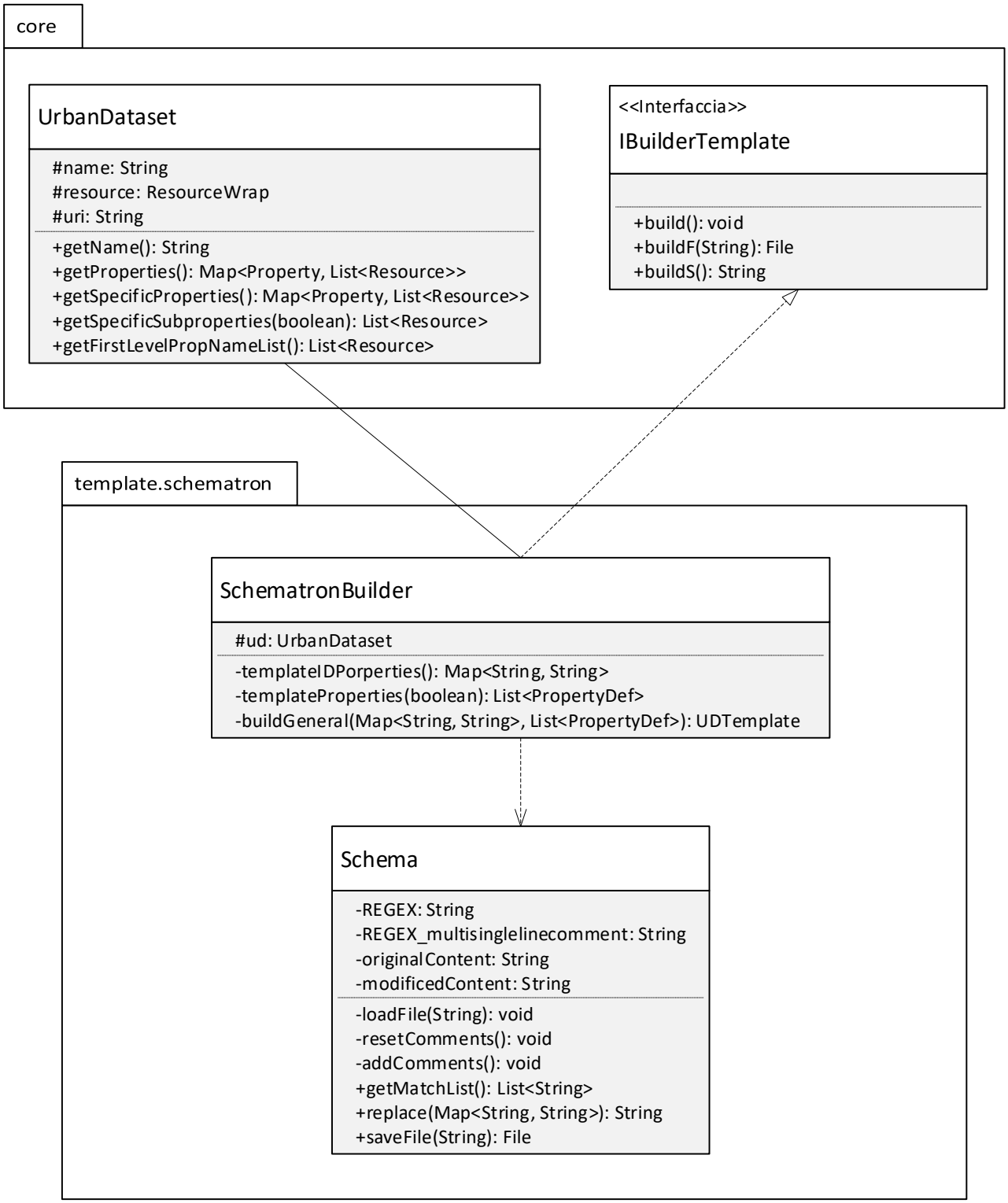

Figura 17. Diagramma delle classi del package *template.schematron*

Per recuperare le informazioni dall'ontologia, è stata usata la stessa classe sviluppata in precedenza per tale scopo, ovvero *UrbanDataset*. Tale classe fornisce un'astrazione del concetto di Urban Dataset e quindi un facile accesso ai dati contenuti nell'ontologia. A partire da un'istanza di *UrbanDataset*, vengono invocati i metodi *getName()*, *getProperties()*, *getSpecificSubProperties()* e *getFirstLevelPorpNameList()* (vedi Figura 17).

Il metodo *getProperties()* provvede a recuperare la lista delle relazioni nell'ontologia dove l'Urban Dataset è soggetto. La lista è ritornata sotto forma di Map dove ad una property è collegata una lista di resource dal momento che per ogni proprietà può corrispondere più di un valore.

Il metodo *getSpecificSubProperties()* si occupa di recuperare tutte le risorse che sono oggetto delle relazioni definite con: *hasUrbanDatasetProperty* e *hasContextProperty*. Si tratta quindi delle proprietà ed eventuali relative sotto-proprietà che sono definite specificatamente per quest'ontologia per indicare le informazioni che devono essere trasmesse all'interno del formato di comunicazione. Il risultato dell'operazione è una lista di risorse con possibili ripetizioni nel caso una proprietà sia sotto-proprietà di più di un'altra.

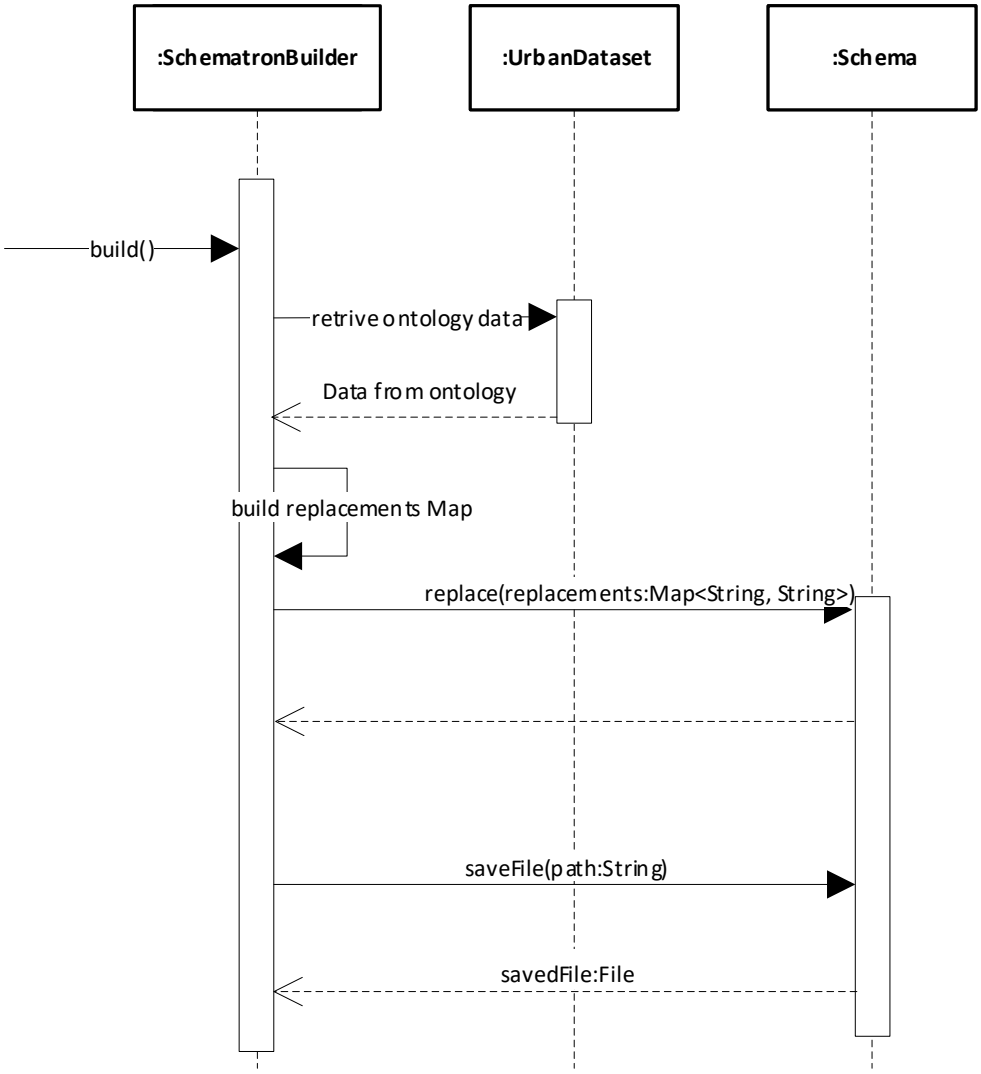

Figura 18. Diagramma di sequenza della creazione di un file di validazione Schematron

Il metodo getFirstLevelPorpNameList(), invece, recupera tutte le proprietà specifiche dell'UrbanDataset di primo livello, quindi senza le sotto-proprietà, ad esclusione delle proprietà speciali quali periodo, coordinate e descrizione. Anche in questo caso, il risultato dell'operazione è una lista di oggetti che modellano una proprietà.

Tutte le liste fin qui costruite serviranno per compilare tutti i campi dello Schematron. Infatti, a partire da queste informazioni è possibile ricavare tutte le liste e le dimensioni necessarie per i diversi campi dello Schematron visti in precedenza.

Una volta recuperate tutte le informazioni come mostrato in Figura 18, l'oggetto istanza di *SchematronBuilder* si occuperà di organizzare le sostituzioni sul file di base per gli Schematron. A questo scopo organizza tutte le informazioni su una struttura Map, denominata *replacements*, per inviarli ad una istanza dell'oggetto Schema. Sarà quest'oggetto ad occuparsi della sostituzione vera e propria. Questo oggetto ha informazioni su qual è il file base per effettuare le sostituzioni e memorizzerà il contenuto nella variabile locale *originalContent*. Eliminerà i commenti presenti nel testo del file individuandoli con una espressione regolare. Una delle specifiche è di inserire nell'intestazione del file anche un commento che indica la data di generazione ed altre indicazioni fisse. Anche in questo caso si è scelto di utilizzare una espressione regolare per individuare la porzione di testo dove andare a posizionare il commento. I metodi che effettuano queste operazioni sono rispettivamente *resetComments()* e *addComments()* indicati in Figura 17.

I metodi *getMatchList()* e *replace()*, invece, operano sulle etichette da sostituire all'interno dello scheletro di file, ovvero sostituisce le stringhe che iniziano con i caratteri ***$$***. Il primo ritorna tutta la lista di stringhe disponibili nel file scheletro che sono indicate come da sostituire. Il secondo, data una struttura Map che collega alla stringa da sostituire la stringa che la deve sostituire, effettua la sostituzione vera e propria. Anche in questo caso per l'individuazione delle parole chiave da sostituire è stata usata una espressione regolare. L'operazione ritorna il contenuto del file modificato secondo le liste definite dall'istanza di *SchematronBuilder*.

Come si può notare, in diversi casi si è scelto di operare attraverso espressione regolare per la selezione di particolari stringhe all'interno del file scheletro di base. La motivazione di tale scelta è dovuta alla relativa semplicità con cui è stato possibile completare l'operazione di sostituzione usando questa tecnica. La libreria standard di JAVA fornisce uno strumento per la gestione di espressioni regolari. In particolare, la classe Pattern permette, data una espressione regolare e un testo, di procedere iterativamente lungo il testo per scoprire eventuali corrispondenze tra testo ed espressione regolare. In caso di individuazione di un match, effettua la sostituzione sulla base delle caratteristiche del gruppo di caratteri individuato (cioè che parola particolare ha ottenuto una corrispondenza con l'espressione regolare). Quello che viene effettuato esattamente è appendere il testo dalla corrispondenza precedente (o dall'inizio del testo) in una nuova stringa su cui si sta operando, appendere la sostituzione alla corrispondenza trovata e procedere fino al prossimo match (o alla fine del testo). Un altro vantaggio è la possibilità di rispondere velocemente al cambiamento del formato delle stringhe da individuare o se si dovesse aggiungere una nuova variabile da sostituire all'interno del testo, a patto che abbia sempre lo stesso formato delle altre, ovvero i due caratteri iniziali particolari. In questo caso, infatti, la classe Schema non cambierebbe, l'unica cosa che cambierebbe è la lista delle corrispondenze da sostituire e relativi sostituti che si dovrebbe passare per effettuare la sostituzione. Quindi recuperando questa informazione l'istanza dallo *SchematronBuilder* non dovrebbe essere fatta nessun'altra modifica.

Discorso simile vale per la rimozione e inserimento di commenti. Si tratta di uno strumento facile da usare per individuare del testo con particolari proprietà su cui operare. I commenti in linguaggi di programmazione hanno la caratteristica di essere delimitati con regole chiare per far sì che i compilatori o altri strumenti automatici possano isolarli facilmente per ignorali usando proprio specifiche espressioni regolari.

L'istanza di Schema contiene anche il risultato delle operazioni, per questo è possibile chiamare un metodo per il salvataggio su file del contenuto modificato passando semplicemente il nome del file su cui salvare. In questo caso viene ritornato un riferimento all'oggetto File ottenuto dall'operazione.

Dal momento che si è deciso di far implementare ad entrambe le applicazioni fin qui mostrate l'interfaccia IBuildTemplate, è stato anche deciso di creare un'unica interfaccia di interrogazione tramite CLI. Come in precedenza sono presenti i parametri ***-U*** oppure ***--Urbandataset*** per indicare quale Urban Dataset selezionare per la generazione del file Schematron relativo e dal parametro ***-F*** oppure ***--file*** per indicare una lista di Urban Dataset da elaborare elencati uno per riga in un semplice file di testo. Per differenziare la volontà di generare un template XML o uno Schematron si è deciso di aggiungere i parametri ***-t*** (in alternativa

--XMLmessage) e ***-s*** (o in alternativa ***--schematron***) per generare rispettivamente un template XML o un file di validazione Schematron.

6 Trasformatore di formato dei template di messaggio

In questo capitolo è descritto il lavoro svolto per la realizzazione del software responsabile della trasformazione di documenti XML contenenti i dati di un Urban Dataset in formato JSON e viceversa secondo le specifiche definite all'interno di un JSON Schema.

6.1 Caratteristiche funzionali

Una delle caratteristiche richieste dalle specifiche per questo software è la possibilità di trasformare un UrbanDataset in formato XML conforme alle specifiche XSD viste in precedenza, in un UrbanDataset in formato JSON conforme alla specifica JSON Schema utilizzata dalla piattaforma.

La seconda caratteristica è la possibilità di eseguire l'operazione inversa. Ovvero, è necessario elaborare la richiesta di trasformazione di un file JSON contenente un Urban Dataset in un documento XML secondo il formato visto in precedenza.

Anche in questo caso è necessario poter invocare in due modi diversi questa applicazione. Ovvero, deve poter essere chiamata sia singolarmente, cioè da riga di comando, che integrata nel servizio web. Quindi deve prevedere una struttura adattabile a questa doppia natura.

6.2 Descrizione dei requisiti

Il formato del JSON rispecchia abbastanza fedelmente la struttura del documento XML visto in precedenza:

```
{
    "UrbanDataset": {
        "specification": {
        },
        "context": {
        },
        "values": {
        }
    }
}
```

Come si vede dallo scheletro di struttura qui sopra, si vede che anche in questo caso il documento è diviso in tre parti che mappano fedelmente la struttura XML, ovvero da **Specification**, **Context** e **Values**. Anche qui **Specification** descrive la struttura generale dell'Urban Dataset ed in esso sono elencate tutte le proprietà che lo compongono quali *version*, *id*, *name*, *uri* e le proprietà di ciascun Urban Dataset come mostrato qui di seguito.

```
    "specification": {
        "version": "1.0",
        "id": {
            value: ""
        },
        "name": "",
        "uri": "",
        "properties": {
            "propertyDefinition": [
            ]
        }
    }
```

Qui è evidente il tag *propertyDefinition* dove, come nel caso dei documenti XML, sono inseriti i dettagli sulle proprietà dello specifico Urban Dataset.

All'interno del tag *propertyDefinition* sono presenti dei campi specifici per la definizione delle singole proprietà:

```
"propertyDefinition": [
    {
        "propertyName": "",
        "propertyDescription": "",
        "dataType": "",
        "unitOfMeasure": ""
    }]
```

Ciascuno di questi campi ha un corrispettivo all'interno dei documenti XML. Per cui è possibile fare un mapping uno ad uno. Anche in questo caso, le proprietà con sotto-proprietà, non possiedono *dataType*, *codeList* e *unitOfMeasure*, ma al loro posto un campo *subProperties* con i nomi delle proprietà che sono poi definite nella stessa lista delle proprietà anziché all'interno della proprietà entro cui compaiono per la prima volta. Di seguito un esempio della struttura:

```
"properties": {
    "propertyDefinition": [{
        "propertyName": "PropA",
        "propertyDescription": "",
        "subProperties": {
            "propertyName": [
                " subPropA",
                " subPropB",
            ]
        }
    },
    {
        propertyName": " subPropA ",
        "propertyDescription": "",
        "dataType": "",
        "unitOfMeasure": ""
    }
    ]
}
```

Come si vede dall'esempio, La proprietà **PropA** ha due sotto-proprietà e di seguito nella stessa lista **properties** sono definite le sotto-proprietà (nello schema ne è mostrata solo una per motivi di sintesi) seguendo la stessa struttura del caso XML.

Anche in questo caso, tra le proprietà con sotto-proprietà c'è il caso particolare delle proprietà speciali *period* e *coordinates*. Queste hanno delle proprietà che devono essere presentate in un ordine particolare. Il problema è che nei grafi RDF con cui è disegnata l'ontologia non c'è una relazione di ordinamento, per cui l'ordine con cui vengono ritornate le sotto-proprietà è casuale per semantica delle operazioni degli store RDF. Per questo è necessario prevedere un sistema per ordinare tali proprietà.

La sezione Context è così definita:

```
"context": {
    "producer": {
        "id": "",
        "schemeID": ""
```

```
        },
        "timeZone": "",
        "timestamp": "",
        "coordinates": {
            "format": "",
            "longitude": 0,
            "latitude": 0
        },
        "language": ""
    },
```

Anche i n questo caso deve essere composto da cinque parti: *producer*, *timeZone*, *timestamp*, *coordinates* e *language*. Si tratta di una struttura statica che non è influenzata dall'ontologia ed è parte del formato.

La sezione values è così definita:

```
    "values": {
        "line": [{
            "id": 1,
            "description": {
            },
            "timestamp": {
            },
            "coordinates": {
            },
            "period": {
            },
            "property": [{
                "name": "PropA",
                "property": {
                    "name": " subPropA ",
                    "val": "",
                "property": {
                    "name": " subPropB ",
                    "val": ""
                },
                {
                "name": "PropB",
                "val": ""
                }
            }]
        }
    ]}
```

Come si può notare, questa sezione contiene la chiave *line* che contiene una lista di coppie proprietà/valore rilevato (campi *property/name* e *property/val*). Ad ogni *line* può essere associato un numero d'ordine (campo *id*).

All'interno di ogni line sono definiti alcuni campi opzionali: *description*, *timestamp*, *coordinates* e *period.* La loro presenza dipende dalla definizione dell'Urban Dataset. In particolare, se all'interno della definizione nel

documento XML sono presenti le relazioni con queste proprietà speciali, allora in questa sezione sono mostrati secondo la regola appena mostrata.

Tutte le altre proprietà specifiche sono mostrate di seguito a queste proprietà speciali e contengono il tag *val* che contiene il valore rilevato dal sistema e oggetto della trasmissione dati. Le proprietà che contengono sotto-proprietà, a differenza del caso della sezione *specification*, contengono direttamente un livello delle sotto-proprietà con il valore da trasmettere.

6.3 Descrizione dell'architettura

Anche in questo caso, l'uso previsto per quest'applicazione è di comunicare sia con una interfaccia da riga di comando che con una applicazione web. Il principio di funzionamento resta, quindi, lo stesso delle applicazioni mostrate in precedenza: un sistema con una interfaccia interrogabile da due attori diversi.

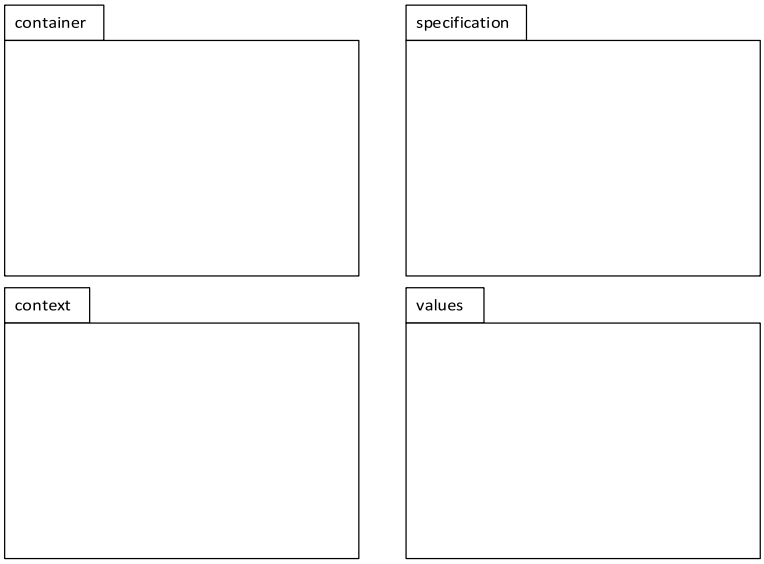

Figura 19. Diagramma dei package per il trasformatore XML-JSON

Il sistema è diviso in quattro package (Figura 19). Il package *container* contiene le classi principali che si occupano di gestire la fase di traduzione XML JSON e viceversa. Gli altri package contengono le classi che si occupano di effettuare la traduzione delle diverse sezioni del documento.

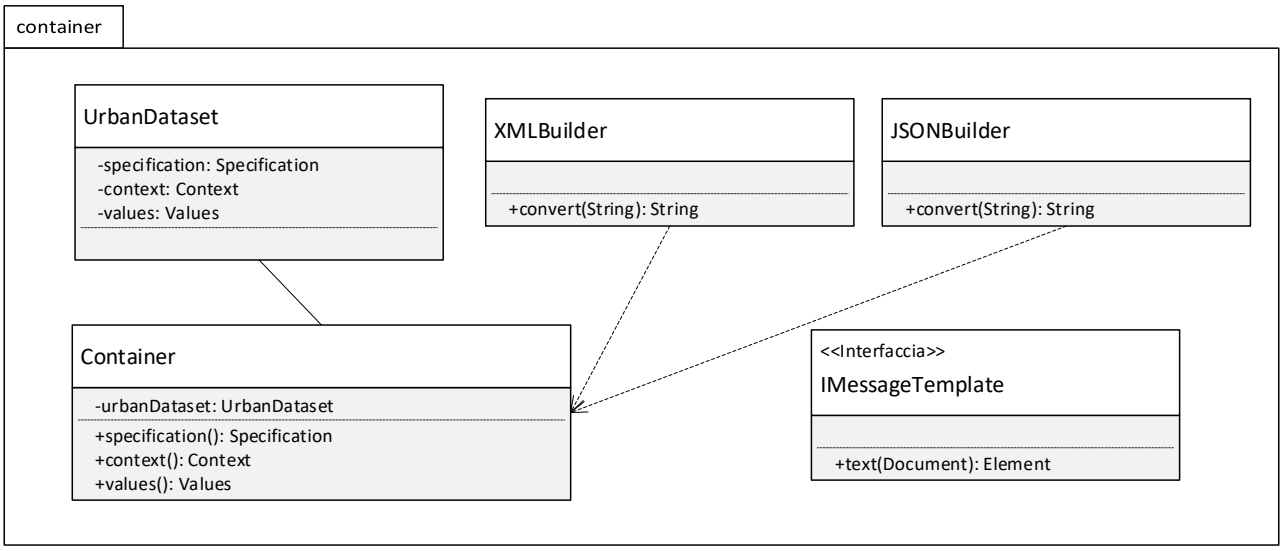

Figura 20. Diagramma delle classi del package *container*

Il package container contiene una classe denominata *Container* il cui scopo è fornire i metodi di trasformazione XML-JSON e provvede a creare l'oggetto di tipo *UrbanDataset*. Questo oggetto, che

rappresenta il risultato della trasformazione, sarà compilato nelle varie sezioni durante l'esecuzione del metodo *convert()* presente nelle classi *XMLBuilder* e *JSONBuilder*. Queste due classi sono il punto di accesso alle operazioni di trasformazione e a cui viene passato il testo del file da convertire. I metodi specification(), context e values() di container hanno lo scopo, invece, di completare la trasformazione dei file costruendo la struttura dell'UrbanDataset rappresentata nel file stesso. Tale rappresentazione è effettuata secondo la struttura definita dalle classi negli altri package.

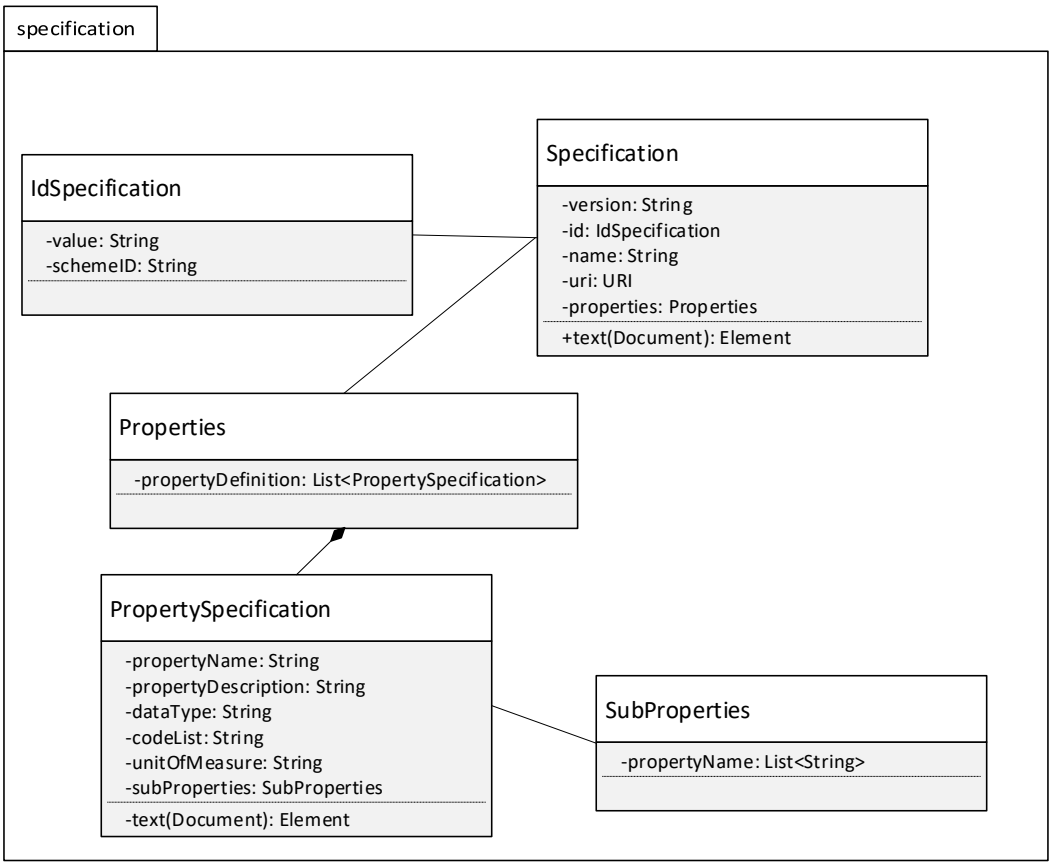

Figura 21. Diagramma delle classi del package *specification*

Il package *specification* (Figura 21) mostra la struttura delle classi usata per rappresentare le informazioni della parte relativa alla sezione Specification all'interno dei file XML e JSON utilizzati per lo scambio sulla Smart City Platform. La classe Specification costituisce la parte principale ed il punto di accesso alla struttura dati. In essa sono rappresentati tutti i campi che sono utilizzati all'interno delle due tipologie di file.

La rappresentazione JSON delle informazioni prevede che attraverso le parentesi graffe vengano delimitati degli oggetti Javascript (JSON sta infatti per JavaScript Object Notation). In java viene usata la stessa tecnica di rappresentazione. Per cui il campo properties è rappresentato come una nuova classe che contiene a sua volta una lista di altri oggetti, in questo caso come singola proprietà. Rappresentando la struttura del file di output come una struttura di classi permette di serializzare automaticamente il contenuto degli oggetti come l'output desiderato. Questa è la modalità comune di funzionamento di varie librerie di serializzazione per JSON.

In sostanza, quello che viene fatto è ricreare la stessa gerarchia di dati attesa come output testuale, come struttura di classi. I nomi dei campi su file saranno gli stessi nomi dati alle proprietà dell'oggetto JAVA. L'unica necessità è che tutte le variabili (fields) dell'oggetto siano dotati di getter e setter (metodi di accesso e modifica ai vari campi definiti nel software ma non mostrati dei documenti UML). In questo modo il serializzatore potrà operare sia per generare il file JSON sia per leggere il file JSON e compilare la struttura di classi in memoria, in modo da rappresentare fedelmente la struttura del file.

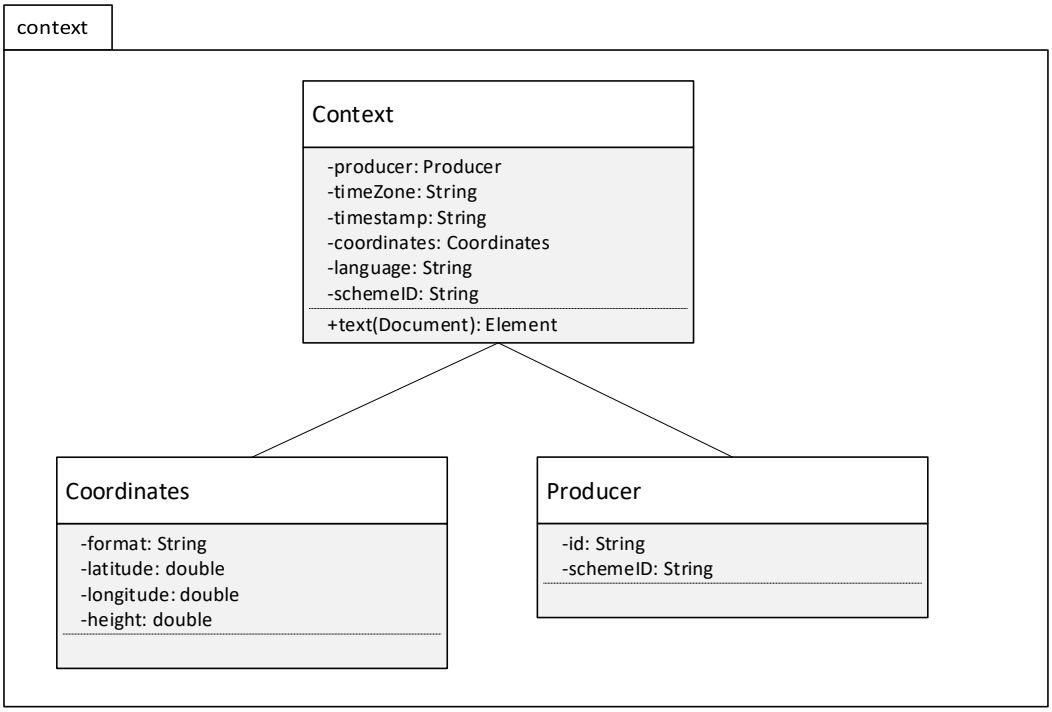

Figura 22. Diagramma delle classi del package *context*

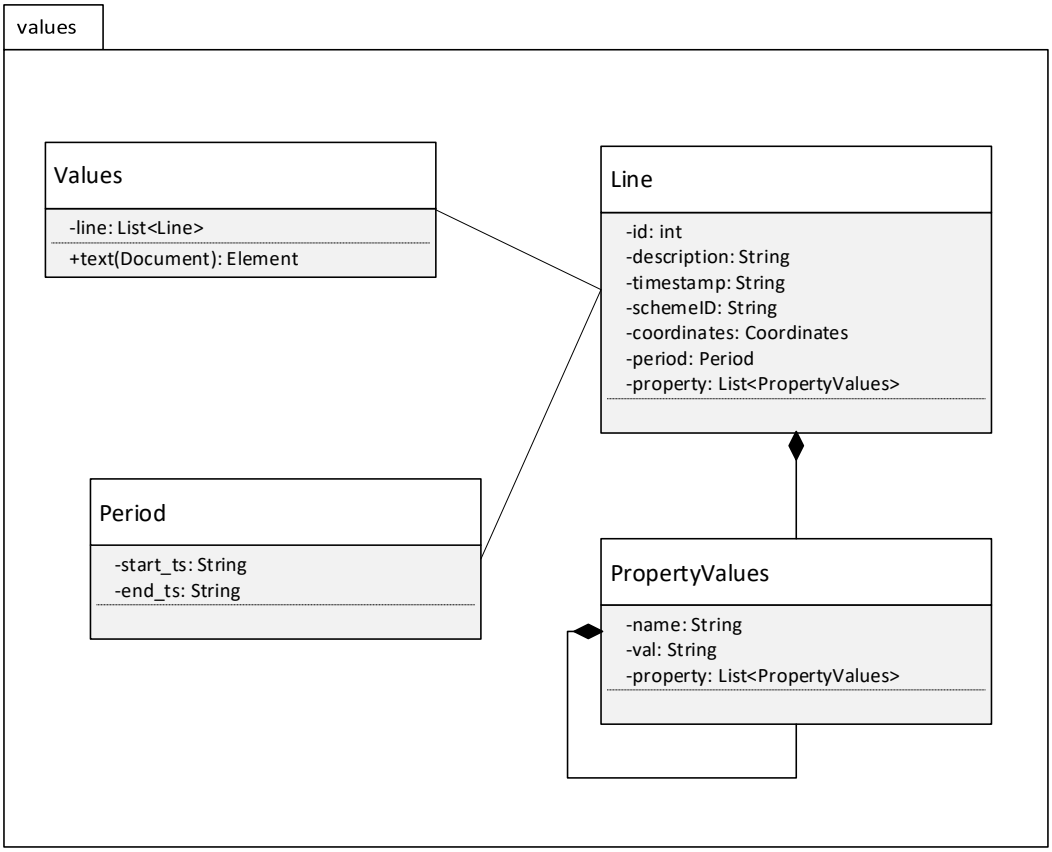

Figura 23. Diagramma delle classi del package *values*

In Figura 22 e Figura 23 sono mostrate le classi presenti nei package relativi alle sezioni context e values dei documenti XML e JSON. La struttura di *context* è più semplice dato che contiene meno proprietà annidate in sottoclassi. La struttura di *values* è più articolata perché contiene più proprietà per ogni linea ed ogni

proprietà può avere delle sotto-proprietà, come si può vedere dallo schema UML in Figura 23, dove c'è una relazione di composizione che sta ad indicare esattamente questo comportamento.

Per la trasformazioni dei documenti JSON in XML vengono usate le stesse classi solo che, come nel caso mostrato nel capitolo 4, la generazione avviene attraverso i metodi *text()* delle diverse sezioni. Tali metodi ricostruiscono la struttura del documento XML e i valori associati ai diversi tag sulla base dei contenuti delle variabili definite nelle rispettive classi. Anche in questo caso, come per il generatore di template, le classi *Specification*, *PorpertySpecification*, *Context* e *Values* implementano l'interfaccia *IMessageTemplate* con la ridefinizione del metodo *text()* con cui viene generata la parte di documento XML che gli compete (Figura 24).

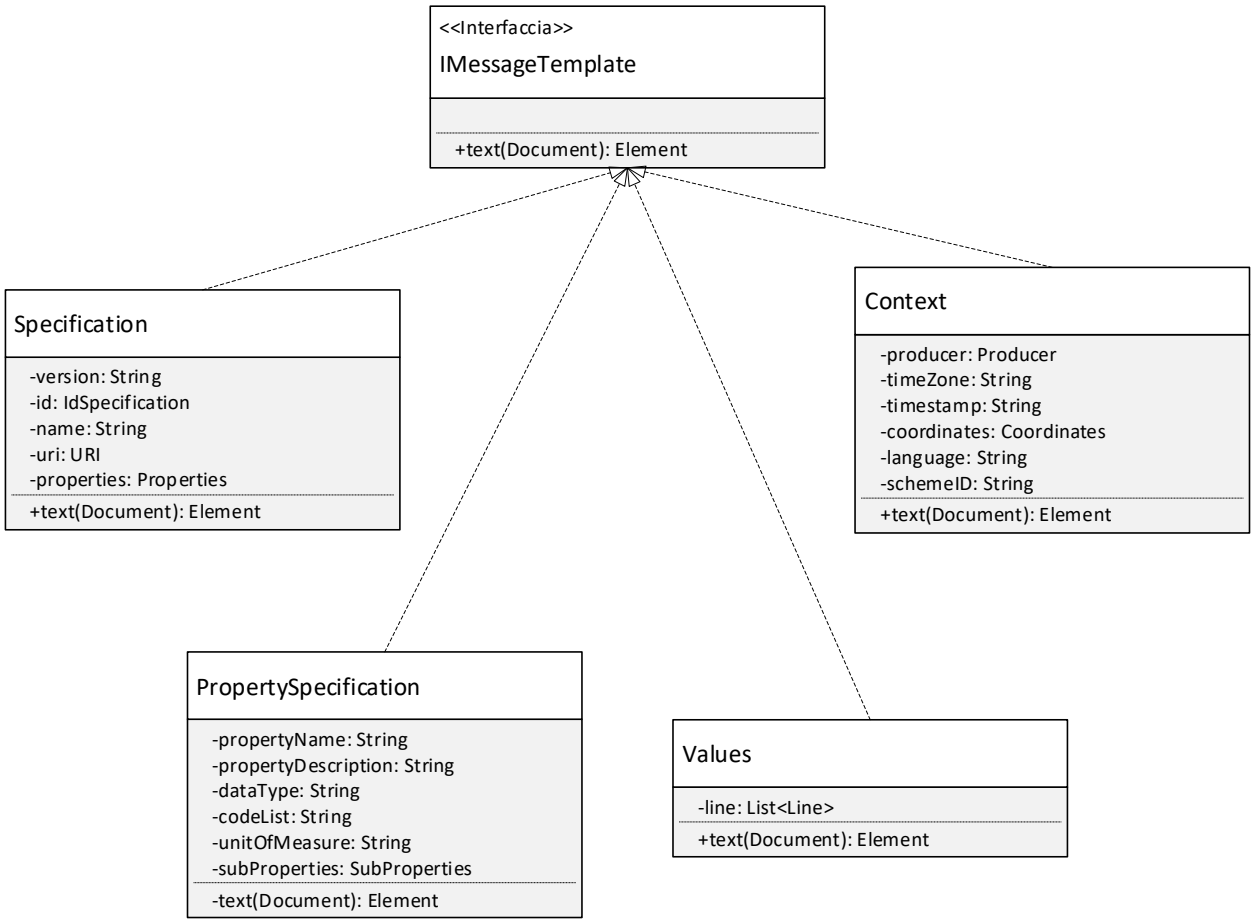

Figura 24. Implementazione dell'interfaccia IMessageTemplate

In Figura 25 è mostrato il diagramma di sequenza per la trasformazione di un documento XML contenente un Urban Dataset in formato JSON. La conversione comincia con la lettura del file XML e il parsing effettuato dalla libreria di elaborazione XML già inclusa nella distribuzione di JAVA. L'oggetto di tipo Document ottenuto è usato per estrarre dai vari tag i valori da usare per settare la gerarchia di classi usata per rappresentare i concetti e le informazioni legate all'Urban Dataset. Per questo vengono invocati i metodi *specification()*, *context()* e *values()* dell'oggetto istanza di *Container*. In Figura 25 è mostrato solo il caso della compilazione della sezione Specification, inoltre non è mostrato tutto il processo sulle varie classi che compongono la struttura ma viene mostrato solo il caso del settaggio della struttura *propertyDefinition* all'intero della classe *Property*.

Una volta ottenuta la struttura completa delle classi, può essere serializzata in JSON utilizzando una libreria di serializzazione per JSON come appena descritto. Nel nostro caso è stata usata la libreria Jackson che fornisce diversi parametri per configurare l'output. In particolare, è possibili configurare come gestire la formattazione (come effettuare ritorni a capo e tabulazioni per gli oggetti innestati) in modo che sia più facilmente leggibile dalle persone.

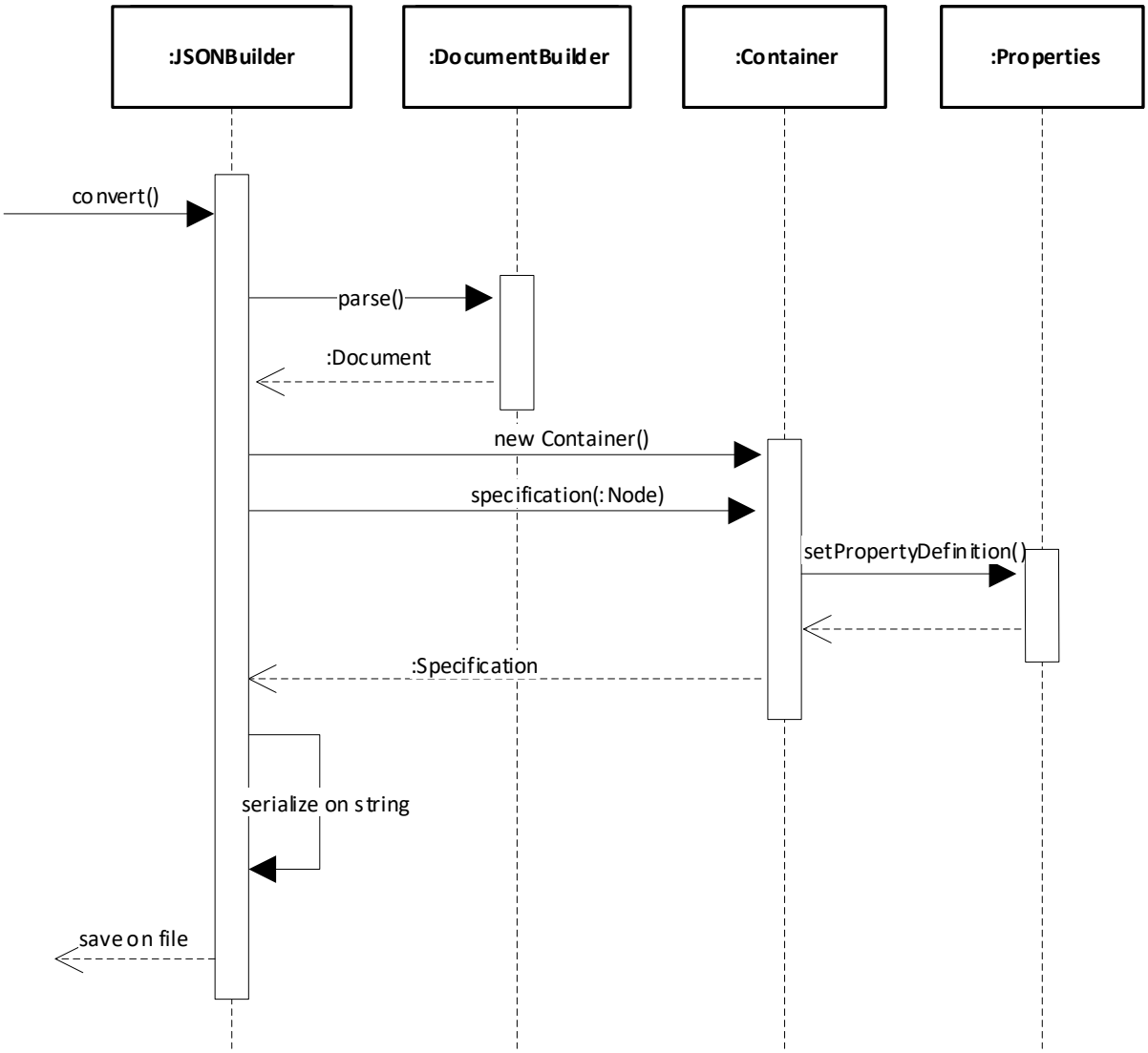

Figura 25. Diagramma di sequenza per la trasformazione di XML in JSON

7 Interfaccia web per l'accesso alle funzionalità sviluppate

In questo capitolo è descritto il lavoro svolto per la realizzazione del software responsabile della presentazione del contenuto dell'ontologia attraverso interfaccia web. Oltre a ciò, fornisce anche un semplice punto di accesso alle altre applicazioni sviluppate permettendo di generare template in XML o JSON e file di validazione Schematron per i diversi Urban Dataset definiti nell'ontologia.

7.1 Caratteristiche funzionali

La progettazione di una interfaccia grafica è sempre un aspetto cruciale in un'applicazione destinata ad un pubblico di non esperti del dominio. Tale è l'obiettivo di questa interfaccia web, ovvero fornire accesso ad informazioni molto specifiche come il contenuto e la struttura dell'ontologia definita e a strumenti di generazione di template per la comunicazione sulla SmartCity Platform.

Quindi, il requisito fondamentale risulta essere quello di fornire un comodo accesso all'esplorazione e il recupero di concetti presenti nell'ontologia in modo da far comprendere la struttura e i collegamenti fra le varie parti. Inoltre, deve fornire un comodo accesso alla generazione di template specifici per un particolare Urban Dataset, sia XML che JSON. Deve essere garantita anche la possibilità di generare file di validazione Schematron relativi ad un Urban Dataset.

Un altro requisito importante riguarda la localizzazione. Dal momento che questa piattaforma nasce per essere interoperabile, deve prevedere la possibilità di essere localizzata anche in lingue diverse perché lo scopo è di favorire l'integrazione di queste informazioni anche al di fuori dei confini nazionali. Una predisposizione alla possibilità di avere pagine multilingua diventa un requisito molto importante.

Altra richiesta importante è la necessità di tenere traccia delle richieste che vengono effettuate sul sito, ovvero è necessario creare dei log per tracciare l'uso del sito ed eventuali problemi devono essere segnalati lì sopra.

7.2 Descrizione dei requisiti

In una interfaccia web destinata a persone non esperte del dominio, risulta fondamentale strutturare un sistema di navigazione tra le pagine in modo che sia comprensibile e facilmente accessibile senza che sia necessario fornire una guida per l'uso. Pe facilitare l'accesso si è deciso che la struttura della pagina fosse divisa in quattro parti separate.

Si è ritenuto necessaria una sezione con un menù di navigazione che evidenziasse subito quali fossero i contenuti disponibili e navigabili nella interfaccia web. I concetti da evidenziare per la navigazione sono stati identificati in: Urban Dataset, Property, Application Context e Code List. In seguito alla selezione di uno dei concetti sarebbe dovuta apparire una lista con le specifiche istanze all'interno dell'ontologia che facessero riferimento a quel concetto. In particolare, per Urban Dataset e Property, dal momento che è prevista una classificazione delle diverse istanze, tale classificazione doveva essere evidente dal modo come avrebbero dovuto essere presentati a schermo. In questo caso, una rappresentazione ad albero navigabile avrebbe dato subito l'idea della struttura e sarebbe stata molto utile per facilitare la navigazione. Le foglie dell'albero, poi, avrebbero dovuto essere selezionabili per poter visualizzare le informazioni specifiche di un Urban Dataset, di una Property o degli altri concetti mostrati.

All'interno del dettaglio di un concetto ci sarebbero dovuti essere, oltre alle specifiche proprietà e valori corrispondenti, anche un collegamento alle pagine degli altri concetti che sono in qualche modo collegati al concetto attualmente mostrato. Ad esempio, visualizzando il dettaglio di un Urban Dataset sono elencate anche le proprietà usate al suo interno. Di conseguenza, questo elenco dovrebbe essere una lista di collegamenti alle pagine contenenti le descrizioni della proprietà. Al contrario, all'interno di una pagina di dettaglio di una proprietà dovrebbe essere presente anche una lista contenente tutti gli Urban Dataset in cui quella particolare proprietà è usata. Tale lista dovrebbe essere fatta da link alle corrispondenti pagine degli Urban Dataset. In questo modo è possibile navigare tutta la struttura seguendo facilmente i collegamenti incrociati tra i diversi concetti.

Naturalmente, all'interno delle diverse pagine degli Urban Dataset devono esserci i link che avviano la generazione dei template XML e JSON assieme al link per la generazione del file di validazione Schematron. Ovviamente, questi link generano i documenti relativi all'Urban Dataset che si sta visualizzando.

Un altro modo di navigare richiesto riguarda la possibilità di cercare l'istanza che si sta cercando attraverso un campo di ricerca. In alto nella pagina è richiesto di inserire un campo di ricerca con delle checkbox che indicano in quale categoria di concetti effettuare la ricerca, ovvero Urban Dataset, Property e Application Context. Una volta eseguita la ricerca, i risultati devono comparire al di sotto nella pagina. Naturalmente lo scopo della ricerca è visualizzare una lista di contenuti che soddisfano i parametri di ricerca, per cui è necessario che la lista dei risultati sia formata da link che portino alle specifiche pagine delle risorse selezionate.

Come descritto prima, la capacità di essere multilingua è importante. La selezione della lingua deve essere impostata dall'utente e deve essere facilmente integrata all'interno dell'applicativo web al fine di aggiungere in seguito nuove lingue per le traduzioni. Per selezionare la lingua da mostrare esistono diversi approcci. I due più diffusi sono: visualizzare un selettore per la scelta della lingua tra le disponibili, oppure selezionare la lingua confrontandola con le impostazioni del browser. All'interno dei browser è sempre presente la possibilità di definire staticamente una preferenza tra le lingue da visualizzare. All'interno delle richieste http queste preferenze vengono inviate al server web che deve fornire la pagina. Il parametro che indica la referenza sulla lingua può, quindi, essere usato per scegliere la traduzione da visualizzare. In questo caso si è scelto di visualizzare la lingua in funzione delle impostazioni del browser. Non è quindi presente un menù di scelta all'interno della pagina web. La scelta della lingua è definita nelle impostazioni del browser. Nel caso la lingua scelta non sia tra le disponibili, le pagine vengono presentate in lingua inglese.

Un'ultima funzionalità che deve essere presente è la possibilità di attivare dei log delle richieste. Si vuole tenere traccia dell'uso che si fa del servizio web ed individuare eventuali problemi che possono nascere durante l'uso a causa di bug. Per questo motivo è stato richiesto un sistema che tenga traccia delle chiamate effettuate al web server memorizzando gli specifici percorsi visualizzati, l'istante della richiesta e l'IP del richiedente. Naturalmente, nel caso di problemi ed errori, le eccezioni scatenate devono essere salvate anche esse nel file di log.

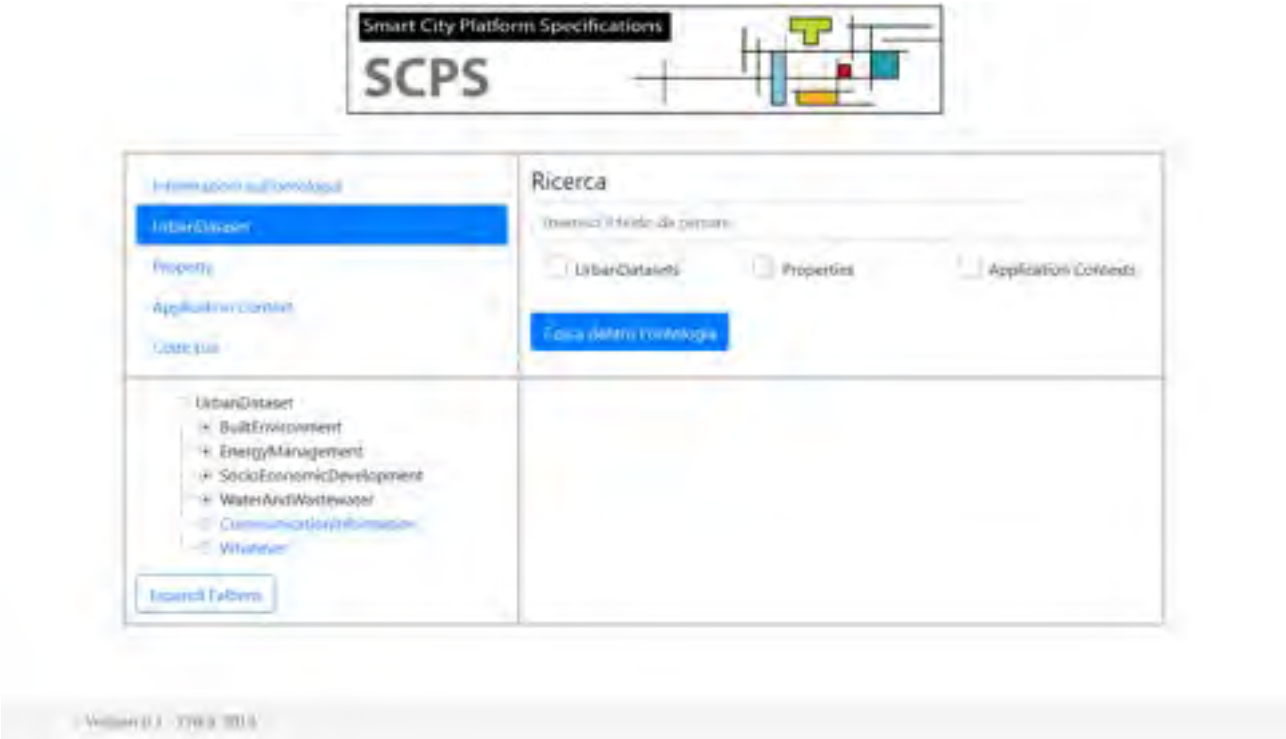

Figura 26. Schermata di esempio dell'applicazione

In Figura 26 è mostrata l'interfaccia sviluppata secondo le specifiche indicate finora. In alto a sinistra è presente il menù di navigazione tra i vari concetti dell'ontologia da visualizzare, in alto a destra c'è l'area di ricerca. A sinistra in basso c'è l'albero di navigazione tra la gerarchia dei concetti presenti, le foglie sono selezionabili e mostrano nel riquadro in basso a destra il dettaglio di ciò che si è selezionato.

7.3 Descrizione dell'architettura

Per la realizzazione del servizio web è stato scelto di usare Spring 5 Framework come framework di sviluppo. Oltre ad essere un framework molto famoso e supportato che sta avendo, da alcuni anni, una diffusione sempre maggiore in ambito di sviluppo JAVA Enterprise e web, offre diversi vantaggi a chi deve sviluppare da zero un servizio web come nel nostro caso. Il vantaggio principale è la disponibilità di una infrastruttura modulare e abbastanza semplice da usare, con componenti già pronti per gestire tutti i principali aspetti e la complessità dello sviluppo di un'applicazione di livello enterprise che abbia a che fare con il web e non solo. Questo framework opensource ha diversi anni di sviluppo alle spalle ed una comunità molto vasta. Questo ha portato ad avere a diversi componenti che possono concorrere a sviluppare con facilità un'applicazione gestendo i diversi aspetti che compongono le diverse tipologie di applicazione e possono essere combinati sulla base dei problemi che questi componenti gestiscono.

Un esempio classico di ciò è lo sviluppo web. All'interno di un'applicazione web ci sono diversi aspetti che tendono ad essere coinvolti. Tra questi c'è la gestione dei dati da visualizzare su una pagina web dinamica, ovvero la visualizzazione di dati raccolti dalle basi di dati all'interno di una struttura di pagina coerente e definita a priori. Oppure la sostituzione di parole chiave con testi.

Quest'ultimo meccanismo viene usato per visualizzare la stessa pagina in lingue diverse. Si evita di realizzare due pagine diverse per due lingue diverse, quello che viene fatto è definire un file per lingua in cui ogni parola chiave è associata ad un testo nella lingua del file. Al momento della trasformazione viene scelto il file di traduzione sulla base della lingua scelta dell'utente, in questo modo la pagina servita viene completata con i testi nella lingua giusta. Per aggiungere una lingua è sufficiente creare un nuovo file di traduzione che abbia tutte le parole chiave tradotte e posizionarlo all'interno del percorso del web server. Ancora, la gestione dei log di sistema è già disponibile tramite l'integrazione con una libreria di logging per applicazioni JAVA. In questo modo si ha subito a disposizione un intero ecosistema completo per sviluppare in modo rapido applicazioni di livello enterprise con strumenti già pronti e testatati da una comunità ampia sia di sviluppatori e contributori che di semplici utenti.

Nel nostro caso specifico, Spring può essere usato in modo molto semplice per compilare delle pagine JSP in cui sono presenti dei contenuti dinamici raccolti in seguito alla chiamata di una particolare API REST. Una volta definita la struttura della pagina sottoforma di JSP, è possibile indicare delle variabili e delle condizioni all'interno della pagina attraverso un particolare tag. In questo modo la pagina può essere costruita in base al "modello" dei dati passati alla pagina per la visualizzazione. Infatti quello che fa Spring, prima di creare la pagina, è di costruire un oggetto di classe *Model* che contiene tutti i dati usati all'interno della pagina sottoforma di coppie chiave-valore dove la chiave corrisponde al nome di variabile presente nella pagina JSP da visualizzare. Per la costruzione del *model*, naturalmente, l'applicativo provvederà a recuperare i dati secondo le indicazioni dello sviluppatore invocando gli opportuni metodi. Il *model* pronto viene passato automaticamente alla pagina, ovvero alla vista, che viene indicata alla fine di un metodo come stringa di ritorno. Le viste sono semplicemente un altro nome per chiamare le pagine da visualizzare e il loro nome viene identificato tra i file presenti grazie ad un oggetto di tipo *ViewResolver*, opportunamente configurato, che permette di risalire alla posizione in cui sono memorizzate. In questo modo si mantengono facilmente separati modello dei dati ed il modo con cui sono visualizzati. Infatti, sapendo che una pagina deve visualizzare un certo tipo di informazioni, queste sono impacchettate dentro un unico *model*. L'aspetto della pagina diventa indipendente dai dati, per cui la pagina può posizionare le informazioni nel modo che preferisce e può anche decidere di visualizzare solo parte dei dati del modello o inserire delle condizioni di visualizzazione sulla base del contenuto delle variabili presenti nel modello. Questo è, in breve, il funzionamento del meccanismo *ModelAndView* di Spring.

I punti di accesso a questo meccanismo sono gli indirizzi effettivamente chiamati attraverso browser web. Quando si chiede di visualizzare un particolare indirizzo, nel caso si un sito web con pagine statiche, a questo corrisponde una particolare pagina html che viene servita al richiedente senza particolari elaborazioni. Nel caso di un servizio web con pagine dinamiche, queste devono essere compilate prima di essere servite e la compilazione può dare risultati diversi a seconda di parametri nella richiesta e dal contenuto in quel momento della base di dati presente sul server. Per gestire questo meccanismo di compilazione delle pagine, ad ogni percorso del server web disponibile corrisponde un particolare metodo JAVA che avvia il recupero delle informazioni e ritorna i risultati ad esempio in una nuova pagina web. Dal momento che questa corrispondenza percorso-metodo è molto comoda e potente, è un comportamento standard e la creazione della corrispondenza è facilitata. Infatti, il sistema a tempo di esecuzione cerca automaticamente tutte le corrispondenze definite all'interno del codice e rende disponibile il collegamento richiedendo pochissimo sforzo allo sviluppatore. I parametri delle richieste http sono automaticamente passati ai metodi corrispondenti basandosi sulla corrispondenza dei nomi dei parametri tra i mondi http e Java operando anche una trasformazione automatica dei tipi se possibile e necessaria.

Affinché il mapping sia attuato, i metodi con i percorsi web corrispondenti devono essere definiti all'interno di oggetti di tipo *Controller*. Spring definisce una serie di stereotipi per i tag di Java con cui è possibile identificare classi e metodi che ci si aspetta abbiano un certo comportamento. Uno di questi è proprio *Controller*. Il significato è proprio quello di gestire i percorsi ed effettuare le operazioni corrispondenti. Dal momento che l'applicativo web sviluppato ha principalmente la funzionalità di fornire pagine in seguito a richieste abbastanza semplici, i *Controller* risultano quindi una parte fondamentale dell'applicazione. Perciò sono state definite alcune classi *Controller* che gestiscono le diverse richieste e servono le pagine corrispondenti.

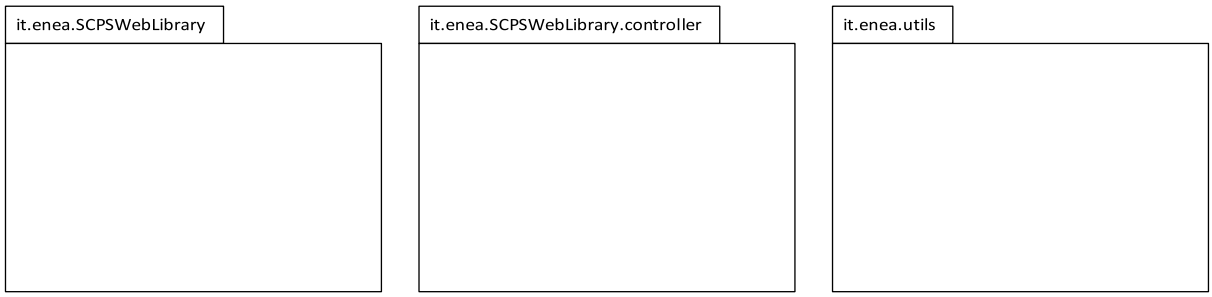

Figura 27. Diagramma dei package

In Figura 27 è mostrato il diagramma dei package per l'applicazione dove si può notare che un intero package è usato per contenere i *Controller* usati. In *utils* sono inserite funzioni statiche di utilità per recuperare i dati e costruire dinamicamente l'albero di navigazione laterale. Nel package *it.enea.SCPCWebLibrary* sono presenti classi di configurazione del servizio.

Il package principale è quello che contiene i *Controller* ed in Figura 28 ne è mostrato il diagramma delle classi. Come si può vedere, non c'è una connessione tra i diversi controller perché i loro comportamenti sono indipendenti dagli altri, ci pensa il framework a connetterle con le classi del framework stesso per gestirne le chiamate e il ciclo di vita. L'unica attività richiesta al programmatore è che siano associate le annotazioni corrette di Spring a classi e metodi di classe. Trattandosi di operazioni di recupero informazioni e visualizzazione, i loro compiti sono spesso limitati ad un unico metodo. Tale metodo, naturalmente, per recuperare le informazioni deve fare riferimento alle applicazioni scritte in precedenza ed in particolare alla libreria di accesso alle informazioni dell'ontologia sviluppata durante il precedente anno di progetto.

Nel dettaglio, i controller hanno tutti in metodo pubblico denominato *index()*. Tale metodo è stato legato alla chiamata generica della specifica pagina senza parametri e mostra la pagina in modo simile a quella in Figura 26 dove la parte centrale della schermata è bianca dato che non è stato selezionato nessun elemento dall'albero di navigazione a sinistra. Viene compilato un oggetto Model con i dati da visualizzare, nel caso specifico si tratta della struttura ad albero a sinistra che deve essere ricreata ogni volta dato che deve riflettere eventuali cambiamenti nell'ontologia. La creazione dell'albero è demandata al metodo *initPage()*

che costruisce l'albero e lo aggiunge all'oggetto istanza di Model. Viene selezionata la vista, ovvero la pagina JSP attraverso la stringa di ritorno, e a questa il framework in automatico associa il Model appena creato e costruisce la pagina che viene spedita al richiedente.

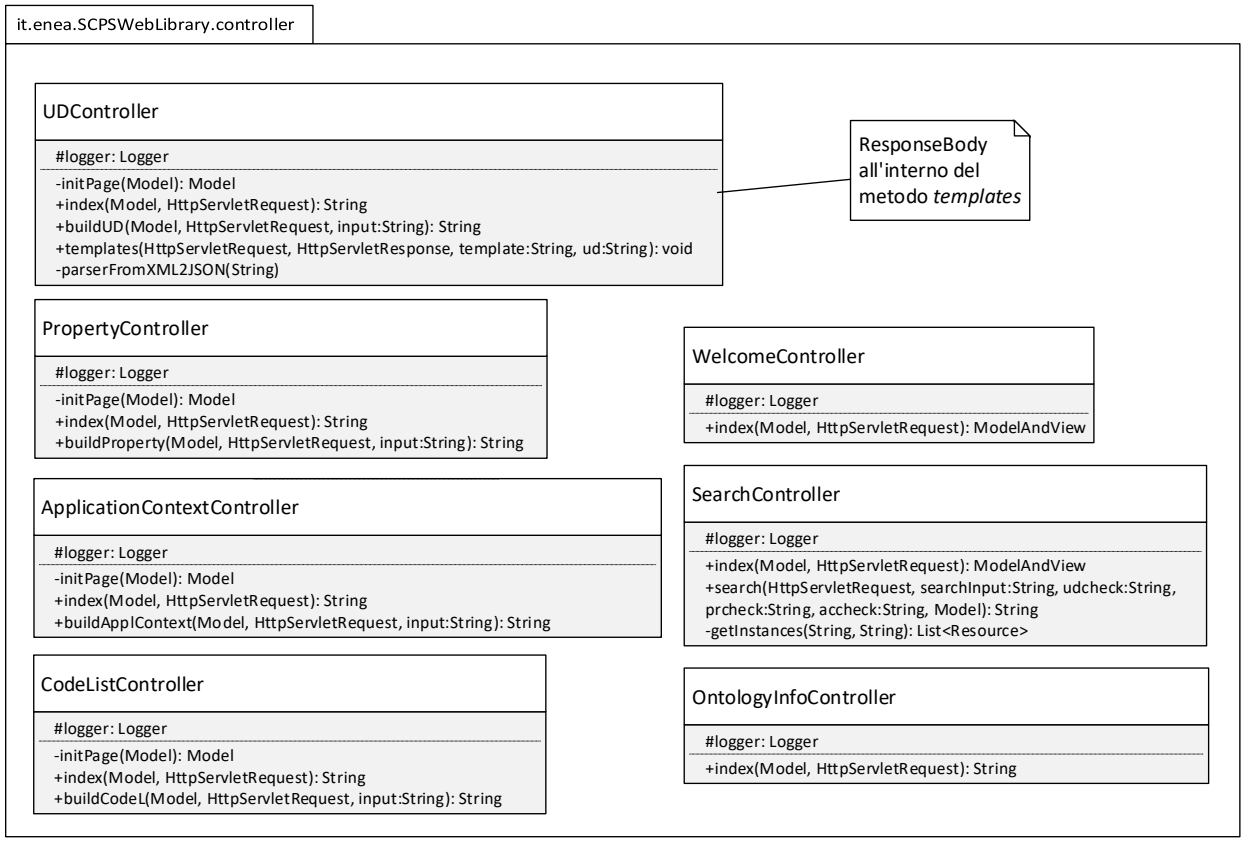

Figura 28. Diagramma delle classi nel package *it.enea.SCPSWebLibrary.controller*

In caso venisse passato il parametro relativo alla selezione dall'albero di sinistra, l'oggetto della selezione è passato come parametro di input e viene chiamato un metodo diverso perché è stato associato un metodo diverso quando è stata decisa la corrispondenza tra chiamate REST e metodi JAVA. In questo caso l'oggetto istanza di Model cambia e contiene altre informazioni. Per questo motivo l'aspetto della pagina cambia e vengono mostrate le informazioni della parte in basso a destra in Figura 26, sebbene la pagina base servita sia la stessa.

Quanto descritto qui vale per le classi *UDController*, *PropertyController*, *ApplicationContextController* e *CodeListController*. *WelcomeController* nel suo unico metodo mappa il percorso radice del servizio web ("/") e fa un redirect alla pagina fornita da *OntologyInfo*. Questa pagina fornisce semplicemente il namespace dell'ontologia ed il percorso URL dove trovare l'ontologia da scaricare.

La classe *UDController* ha un altro metodo chiamato *templates()*. Questo metodo ha come scopo di generare i template di XML e JSON, nonché il file di validazione Schematron attraverso le applicazioni presentate in precedenza. All'interno della pagina di uno specifico Urban Dataset sono presenti del link per effettuare la chiamata a questo metodo. La chiamata è fatta usando la POST http e specificando come parametro anche quale link è selezionato. In questo modo il metodo prepara il file e lo invia attraverso un http *ResponseBody* di tipo testo.

Altro caso speciale è la ricerca all'interno dell'ontologia. La ricerca si focalizza solo sulla ricerca di istanze dei concetti Urban Dataset, Property e Application Context. La ricerca vera e propria è eseguita dal metodo privato *getInstances()*, il risultato viene presentato nella pagina dei risultati con il model costruito dal metodo *search()*.

In tutte le classi è presente una variabile di tipo Logger. Il suo scopo è di recuperare l'oggetto Singleton che si occupa di effettuare il logging del web server e su cui vengono inviati i messaggi generati all'interno del codice sviluppato e salvati eventuali errori. La configurazione del sistema di logging è definita all'interno di un file XML all'interno della cartella "/resource" del web server.

All'interno del package *it.enea.SCPSWebLibrary* sono presenti classi di configurazione e di settaggio per il framework. Tutti i settaggi sono standard secondo le specifiche del framework Spring. L'unica cosa che è stata aggiunta è la configurazione multilingua. In particolare, sono stati definiti, all'interno della classe *PropertiesContextConfiguration*, dei file delle risorse multilingua nella cartella "/resources/i18n". I file vengono individuati automaticamente esplorando la cartella all'avvio e nel nome è indicata la lingua con il codice identificativo internazionale di lingua. È stato inserito un risolutore diverso per la lingua perché il comportamento di default nel caso multilingua è che, in assenza di una traduzione per la lingua indicata nel browser dell'utente, non viene effettuata nessuna assegnazione di lingua per cui vengono mostrate le parole chiave usate per posizionare il testo che necessita di essere localizzato senza effettuare un'associazione di lingua e rendendo il sito non compresibile all'utente. L'unico modo è di creare una nuova classe *LocaleResolver* che sostituisce la classe di default di Spring che si occupa di individuare la localizzazione del browser dell'utente. Tale classe si occupa di scegliere un comportamento di default nel caso non sia presente una lingua adatta al browser richiedente. A questo scopo è stata scelta come lingua di default l'inglese.

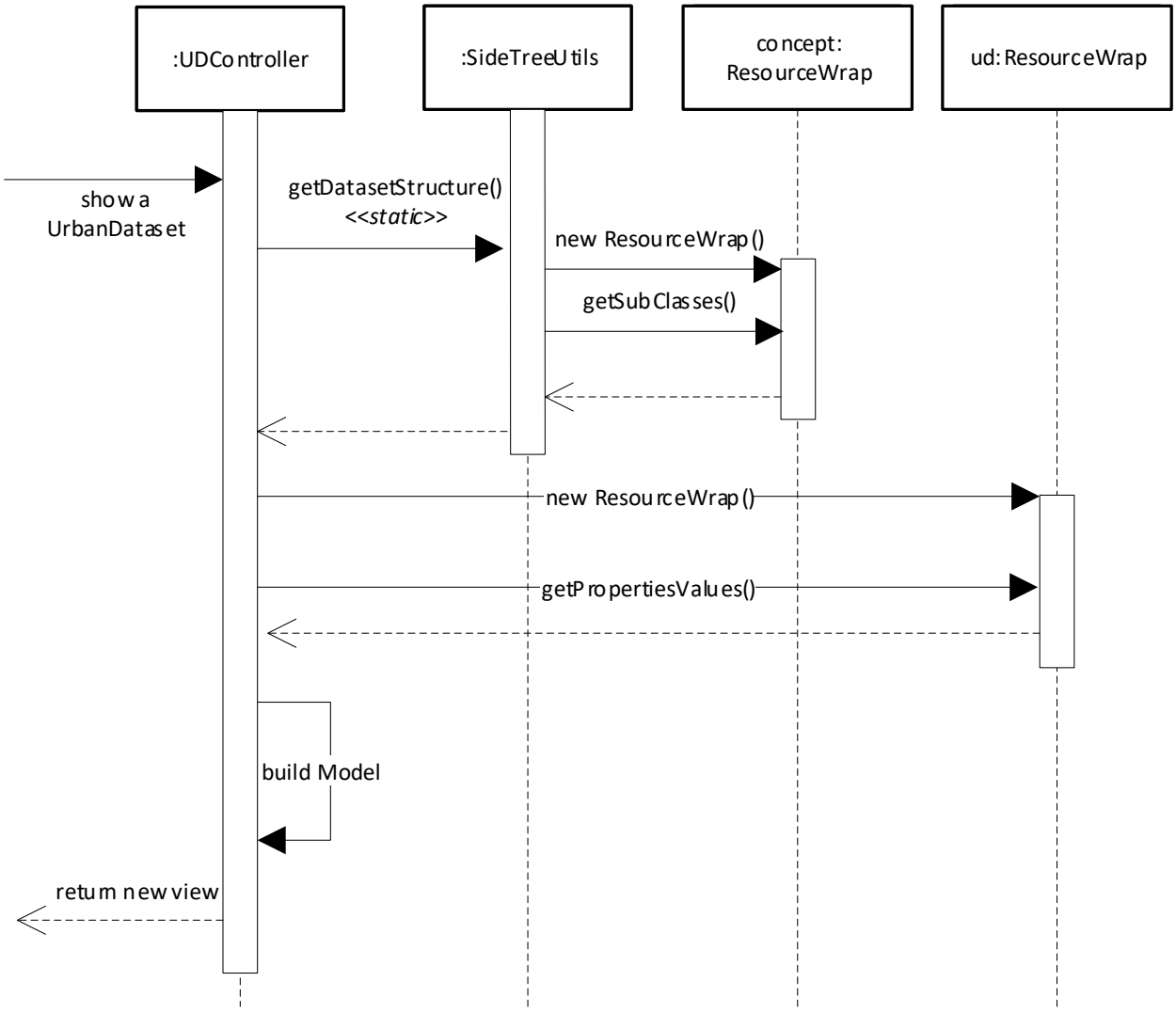

Figura 29. Diagramma di sequenza della visualizzazione del dettaglio di un Urban Dataset

Per quanto riguarda la visualizzazione dei contenuti, sono state scelte alcune librerie per mostrare i contenuti con uno stile uniforme tra le pagine. A questo scopo si è scelto di usare Bootstrap [6] perché fornisce un ambiente comodo per posizionare i componenti all'interno di una pagina html e fornisce in automatico un sistema di riposizionamento in caso gli utenti abbiano schermi di vari formati. L'albero di navigazione laterale ha la caratterisitca di essere collassabile. Tale funzionalità è data dalla libreria Javascript CollapsibleList [7] che associa automaticamente ad alcuni tag un comportamento grafico in modo da ottenere il risultato desiderato.

In Figura 29 è mostrata la sequenza delle operazioni necessarie per la visualizzazione del dettaglio di un Urban Dataset. In seguito alla richiesta, viene generata una richiesta per la costruzione della struttura dell'albero di navigazione per i concetti di Urban Dataset. Tali informazioni vengono recuperate dall'ontologia tramite la classe ResourceWrap presente nella libreria sviluppata nel precedente anno. In seguito, viene richiesto il dettaglio relativo all'Urban Dataset selezionato. Viene richiesta la lista delle proprietà all'ontologia e i dati ottenuti vengono inseriti nell'oggetto Molde per costruire la vista da ritornare al richiedente.

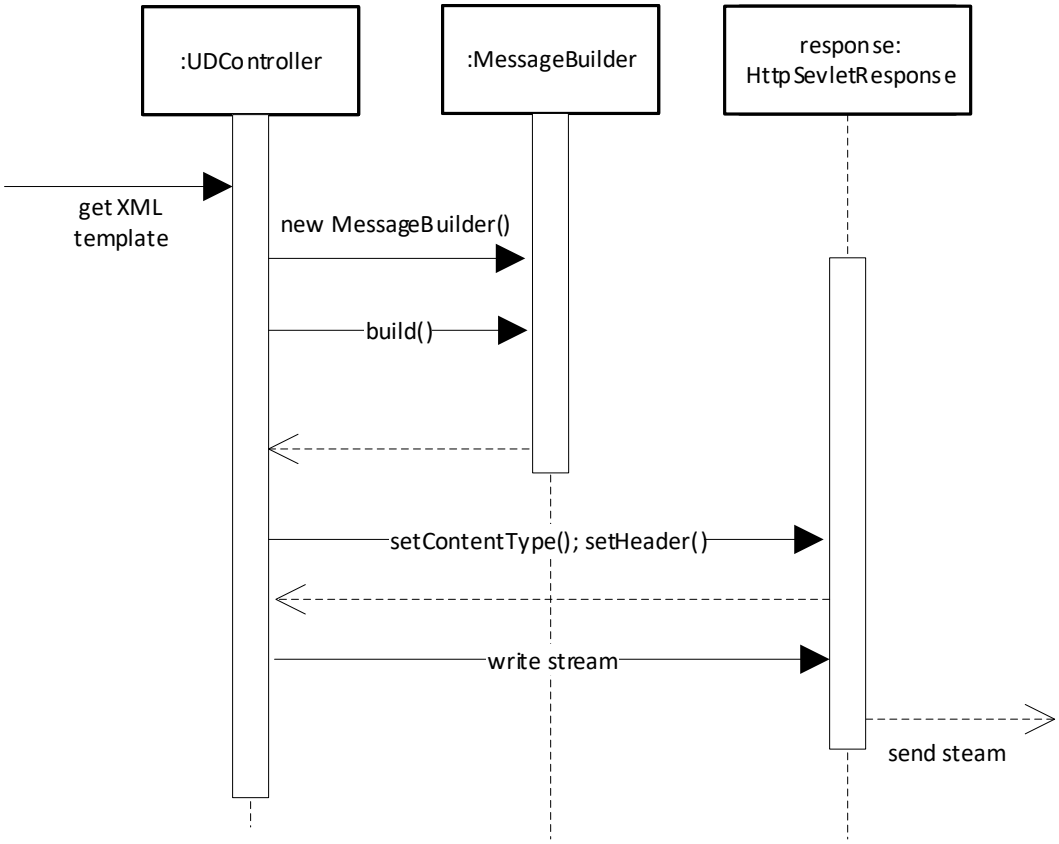

Figura 30. Diagramma di sequenza della generazione di un template file

In Figura 30 è mostrato il diagramma di sequenza della generazione di un template XML. In seguito alla generazione del file effettuata dall'istanza dell'oggetto *MessageBuilder* (vedere Figura 15 per il dettaglio della sequenza di generazione), questo file viene inserito all'interno di una risposta http per poter essere inviato al richiedente. Comportamento simile si ha nel caso di generazione ed invio di template JSON o file di validazione Schematron. La generazione viene effettuata con le applicazioni mostrate nei capitoli precedenti, l'invio viene effettuato inserendo i file generati all'interno di un oggetto istanza della classe *HttpServeltResponse* che Spring provvede ad inviare al richiedente al termine delle operazioni di composizione del messaggio.

8 Conclusioni

In questo documento si è descritto il lavoro svolto per la definizione e realizzazione di diverse applicazioni con l'obiettivo di sfruttare l'ontologia costruita nei precedenti anni di progetto. Lo scopo di queste applicazioni era di migliorare la gestione dell'ontologia e delle specifiche dei documenti di scambio delle informazioni in formato XML e JSON all'interno di una piattaforma ICT per una Smart City. Lo scopo di tale piattaforma era la condivisione di informazioni tra i vari ambiti applicativi di una città. La raccolta e la condivisione di informazioni tra i diversi ambiti può essere utile a creare sinergie tali da favorire nuovi e più importanti risultati nell'ambito del risparmio energetico e dell'efficienza di una città

Il lavoro svolto quest'anno, sfruttando quanto sviluppato precedentemente, è continuato partendo dalla costruzione di un'applicazione in grado di usare le definizioni degli Urban Dataset contenute nell'ontologia per generare automaticamente template XML conformi al formato di scambio di dati definito dalle SCPS nell'Information level. Le informazioni sono reperite usando la libreria di accesso all'ontologia scritta in precedenza; i template XML sono conformi all'XML Schema che definisce la sintassi e la struttura degli Urban Dataset scambiati in ambito SCPS.
Successivamente è stata progettata e sviluppata l'applicazione per la generazione automatica di file schematron (SCH) specifici per ogni Urban Dataset. I file SCH consentono di validare semanticamente i file XML, ovvero di verificare che vi siano contenute tutte ed esclusivamente le proprietà definite per lo specifico Urban Dataset a cui il file XML fa riferimento. In questo modo è possibile controllare non solo che il formato di scambio dati sia conforme alla sintassi definita al livello Information (compito del file XSD), ma anche che il contenuto sia conforme e completo rispetto al livello Semantic.
Dal momento che il formato di scambio dati definito per gli Urban Dataset supporta anche la sintassi JSON oltre che XML, è stata sviluppata anche un'applicazione per la trasformazione automatica dei file XML in file JSON e viceversa. Prendendo come specifiche di riferimento il JSON Schema e l'XML Schema definiti nel livello Information, si è provveduto a definire un sistema di trasformazione automatica da XML a JSON e da JSON a XML. Questo trasformatore avrà tre principali destinazioni d'uso:

- creazione automatica dei template JSON degli Urban Dataset a partire dai template XML;
- successiva implementazione della validazione semantica dei file JSON: poiché i file Schematron possono essere applicati solo a file XML, sarà implementata una procedura automatica che, automaticamente e a cascata, trasforma un file JSON in XML e lo valida con lo Schematron opportuno, ottenendo così un esito di validazione valido anche per il JSON iniziale;
- supporto alla SCP negli scambi di informazioni tra Solution che non usano la stessa sintassi.

Inoltre, il trasformatore sarà reso disponibile e potrà anche essere utilizzato direttamente dalle Solution che usano il formato JSON per inviare/ricevere Urban Dataset in formato XML e viceversa.
Infine, si è realizzato una interfaccia web per accedere ai dati contenuti dell'ontologia. Tramite una serie di pagine web è possibile accedere ai dati contenuti nell'ontologia sfruttando la libreria sviluppata in precedenza e visualizzare in maniera organizzata i vari concetti presenti. I diversi concetti sono collegati tra loro ed è possibile navigarli osservando la struttura degli Urban Dataset. Le pagine sono generate dinamicamente interrogando l'endpoint SPARQL su cui risiede l'ontologia e da queste pagine è possibile generare i file di scambio delle informazioni XML e JSON, ed i file SCH per la validazione. Inoltre, è possibile effettuare una ricerca per nome tra le diverse entità. La scelta della generazione dinamica delle pagine è stata fatta per assicurare la costante coerenza tra il contenuto dell'ontologia e le informazioni presentate all'utente.
Grazie a queste applicazioni sarà possibile sfruttare a pieno il potenziale dell'ontologia. Infatti, anche chi non è in grado di creare query SPARQL o di navigare i concetti di un'ontologia, avrà a disposizione degli strumenti che ne facilitano di molto sia l'accesso sia il recupero di template relativi agli Urban Dataset. Si tratta comunque di strumenti utili per non solo per i non esperti, ma utili anche per rendere più veloce l'accesso agli utenti esperti.

9 Riferimenti bibliografici

1. Schematron. Disponibile all'indirizzo: http://schematron.com/
2. Provenance Working Group, "The PROV Namespace". Disponibile a: https://www.w3.org/ns/prov.
3. Apache Jena. Disponibile all'indirizzo: https://jena.apache.org/.
4. Apache Commons-CLI. Disponibile all'indirizzo: https://commons.apache.org/proper/commons-cli/
5. Spring Framework. Disponibile all'indirizzo: https://spring.io/
6. Bootstrap. Disponibile all'indirizzo: https://getbootstrap.com/
7. CollapsibleLists. Disponibile all'indirizzo: http://code.iamkate.com/javascript/collapsible-lists/

Curriculum Vitae Michela Milano

Laureata in Ingegneria Elettronica presso l'Università degli Studi di Bologna riportando la votazione di 100/100 e lode, il 16 Marzo 1994.
Ha ottenuto il titolo di **Dottore di Ricerca** in Ingegneria Elettronica e Informatica il 30 Giugno 1998.
Dal 1 Luglio 1999 al 30 Giugno 2000 ha usufruito di una **borsa di studio** per lo svolgimento dell'attività di ricerca **post-dottorato** presso il Dipartimento di Ingegneria dell'Università degli Studi di Ferrara.
Dal 1 Luglio 2000 ha ricoperto il ruolo di **Ricercatore Universitario** presso la Facoltà di Ingegneria di Bologna afferendo al Dipartimento di Elettronica, Informatica e Sistemistica (DEIS).
Dal 1 Novembre 2001 ha ricoperto il ruolo di **Professore Associato nel settore concorsuale 9/H1 (ING-INF/05) – Sistemi di Elaborazione delle Informazioni** presso la Facoltà di Ingegneria di Bologna afferendo prima al Dipartimento di Elettronica, Informatica e Sistemistica (DEIS) poi al Dipartimento di Informatica – Scienza e Ingegneria DISI.

Posizione Attuale
Dal 1 Aprile 2016 ricopre il ruolo di **Professore Ordinario nel settore concorsuale 9/H1 (ING-INF/05) – Sistemi di Elaborazione delle Informazioni** presso la Facoltà di Ingegneria di Bologna afferendo al Dipartimento di Informatica – Scienza e Ingegneria presso cui svolge attività didattica e di ricerca, partecipando attivamente a convenzioni e progetti di ricerca.

Attività di Ricerca
L'attività di ricerca di Michela Milano riguarda i sistemi di supporto alle decisioni basati sulla Programmazione a Vincoli e la sua integrazione con tecniche di Programmazione Intera: in particolare, sono stati investigati sia aspetti metodologici sia aspetti applicativi con riferimento a numerose applicazioni quali scheduling, allocazione, cutting e packing, routing, aste combinatorie e recentemente per problemi decisionale e di ottimizzazioni legati allo sviluppo sostenibile e al processo di policy making.
In questo settore Michela Milano ha raggiunto visibilità internazionale e ha collaborazioni con diversi gruppi di ricerca, universitari e industriali.
È membro dei comitati di programma delle maggiori conferenze e workshop del settore e guest editor di diversi numeri speciali di riviste internazionali.
È **Editor in Chief** della rivista Constraints, è **Area Editor** di Constraint Programming Letters e **Area Editor** di INFORMS Journal on Computing.
È editor di cinque libri sull'ottimizzazione ibrida e autrice di più di 130 lavori su riviste e conferenze internazionali. È stata **program chair** di CPAIOR 2005 e CPAIOR 2010, di CP2012 e di CompSust2012. Su tali argomenti Michela Milano ha tenuto numerosi tutorial nelle maggiori conferenze italiane e internazionali quali: AI*IA99, PACLP2000, CP2000, IJCAI2001. Ha tenuto relazioni invitate a CP2013, ICAPS2004, INFORMS2002, INFORMS99, IFORS99 e numerosi seminari e relazioni invitate in centri di ricerca e industrie.
È membro dell'EurAI **Board** (European Association for Artificial Intelligence) e membro del **AAAI Council** (American Association of Artificial Intelligence). È membro dello **Steering Committee di CPAIOR**. È stata membro del Executive Committee della ACP Association of Constraint Programming, ed è membro del Consiglio Direttivo dell'Associazione Italiana per l'Intelligenza Artificiale AI*IA.
Ha partecipato a numerosi progetti di ricerca italiani ed Europei. È stata **coordinatrice** del progetto Europeo FP7 **ePolicy**, Engineering the Policy Making Life Cycle, 2011-2014 e partner del progetto Europeo FP7 **COLOMBO**, Cooperative Self-Organizing System for low Carbon Mobility at low Penetration Rates, 2012-2015, del progetto EU-FP7 **DAREED**: Decision Advisor for Energy Efficient Districts, 2013-2016 e del progetto EU-H2020 **OPRECOMP**: Open Transprecision Computing, 2017-2020. È stata principal investigator del **Google Focused Grant** on Mathematical Optimization and Combinatorial Optimization in Europe nel 2012. Inoltre le è stato assegnato il **Google Faculty Research Award** nel 2016 su integrazione di reti neurali profonde in modelli combinatori.

Curriculum Vitae Federico Chesani

Laureato in Ingegneria Informatica presso l'Università degli Studi di Bologna riportando la votazione di 98/100, il 17 Luglio 2002.
Ha ottenuto il titolo di **Dottore di Ricerca** in Ingegneria Elettronica, Informatica e delle Telecomunicazioni il 12 Aprile 2007.
Dal 1 Gennaio 2007 al 30 Marzo 2012 ha usufruito di alcune **borse di studio**, per lo svolgimento di attività di ricerca post-dottorato, bandite dal CINI (Consorzio Interuniversitario Nazionale per l'Informatica) e dall'Università di Bologna (Dipartimento DEIS).
Il giorno 1 Aprile 2012 ha preso servizio nel ruolo di **Ricercatore Universitario nel settore concorsuale 9/H1 (ING-INF/05) – Sistemi di Elaborazione delle Informazioni** presso la Facoltà di Ingegneria di Bologna, afferendo al Dipartimento di Elettronica, Informatica e Sistemistica (DEIS).
Nel Dicembre 2013 ha ricevuto **l'abilitazione** al ruolo di Professore di Seconda Fascia ("associato") nel settore concorsuale 9/H1(ING-INF/05), e nel Gennaio 2014 ha ottenuto l'abilitazione, sempre per il ruolo di Professore di Seconda Fascia, per il settore concorsuale 01/B1 (INF/01).

Posizione Attuale
Dal 1 Aprile 2012 svolge attività didattica e di ricerca presso la Scuola di Ingegneria e Architettura dell'Università di Bologna, e afferisce attualmente al Dipartimento di Informatica – Scienza e Ingegneria DISI, nel ruolo di Ricercatore Universitario a tempo indeterminato, partecipando attivamente sia a progetti di ricerca, che a progetti di trasferimento tecnologico.

Attività di Ricerca
Federico Chesani ha svolto la sua attività di ricerca prevalentemente nell'ambito dei sistemi esperti e di supporto alle decisioni basati su approcci a regole: in particolare, si è occupato sia di aspetti teorici legati ai sistemi a regole in logiche abduttive per gestire l'assenza di conoscenza e l'integrazione di conoscenza ontologica, sia ad aspetti maggiormente pratici legati all'applicazione di sistemi in presenza di conoscenza incerta e/o probabilistica. In particolare, nell'ambito dei numerosi progetti a cui ha contribuito, ha applicato sistemi a regole per il supporto alle decisioni in ambito sanitario (sia a livello italiano che europeo), "policy making", e manifatturiero/industriale. Nell'ambito di tale attività di ricerca, è co-autore di oltre 60 pubblicazioni, ed è stato invitato a tenere seminari e tutorial nell'ambito di conferenze internazionali.
Federico Chesani collabora attivamente con gruppi di ricerca nazionali e internazionali, e svolge attività di coordinamento e diffusione a livello nazionale e internazionale. E' membro di diversi comitati di programma di conferenze e workshop, e svolge con continuità attività di revisore sia per progetti nazionali ed europei, che per pubblicazioni su riviste scientifiche. Dal 2013 è membro del consiglio direttivo del Gruppo Ricercatori e Utenti Logic Programming (GULP).
Ha partecipato a numerosi progetti di ricerca italiani ed Europei, tra cui il progetto Europeo FP7 **ePolicy (**"Engineering the Policy Making Life Cycle", 2011-2014), il progetto Europeo FP7 **FARSEEING** (2012-2015), il progetto europeo FP5 **SOCS** (2002-2005), i progetti Nazionali PRIN/COFIN/FIRB **MASSIVE**, **SVP**, e **tocai.it**. Attualmente sta collaborando nel progetto Europeo H2020 **PreventIT** (2016-2018).