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

MINISTERO DELLO SVILUPPO ECONOMICO

Ricerca di Sistema elettrico

Strumenti integrativi per lo strato semantico

Michela Milano, Federico Chesani

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

Report RdS/PAR2018/021

STRUMENTI INTEGRATIVI PER LO STRATO SEMANTICO

Michela Milano, Federico Chesani (Università degli Studi di Bologna)

Dicembre 2018

Report Ricerca di Sistema Elettrico

Accordo di Programma Ministero dello Sviluppo Economico - ENEA
Piano Annuale di Realizzazione 2018
Area: Efficienza energetica e risparmio di energia negli usi finali elettrici e interazione con altri vettori energetici
Progetto: D.7 Sviluppo di un modello integrato di smart district urbano
Obiettivo: Strumenti integrativi per lo strato semantico
Responsabile del Progetto: Claudia Meloni, ENEA

Il presente documento descrive le attività di ricerca svolte all'interno dell'Accordo di collaborazione "Estensione del set di tool per lo strato semantico"
Responsabile scientifico ENEA: Nicola Gessa
Responsabile scientifico Università: Michela Milano

Indice

SOMMARIO 4

1 INTRODUZIONE 5

2 NUOVE SPECIFICHE 7

2.1 GENERAZIONE AUTOMATICA DELLA STRUTTURA DELLE CLASSI 7
2.2 MESSAGGI DI ERRORE 8
2.3 MODIFICHE ULTERIORI 8

3 TEMPLATE DI MESSAGGIO DI UN URBAN DATASET 9

3.1 DESCRIZIONE DELLA NUOVA ARCHITETTURA 9
3.2 INTERFACCIA DI INVOCAZIONE 14

4 TRASFORMATORE DI FORMATO DEI TEMPLATE DI MESSAGGIO 15

4.1 DESCRIZIONE DELLA NUOVA ARCHITETTURA 15
4.2 INTERFACCIA DI INVOCAZIONE 19

5 SISTEMA DI VALIDAZIONE DEI MESSAGGI 20

5.1 CARATTERISTICHE FUNZIONALI 20
5.2 DESCRIZIONE DEI REQUISISTI 20
5.3 DESCRIZIONE DELL'ARCHITETTURA 20
5.4 INTERFACCIA DI INVOCAZIONE 21

6 NUOVA VERSIONE DELL'INTERFACCIA WEB 23

6.1 VALIDAZIONE 23
6.2 TRASFORMAZIONE 23
6.3 GENERAZIONE DELLA DOCUMENTAZIONE 24

7 CONCLUSIONI 25

8 RIFERIMENTI BIBLIOGRAFICI 26

Sommario

Il presente documento è uno dei risultati dell'estensione del terzo anno del progetto del MiSE per lo sviluppo di un modello integrato di Smart District Urbano nel Piano Trimestrale di Realizzazione ENEA 2018 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 anni precedenti 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 che in grado di fornire un facile accesso alle precedenti applicazioni.

Questo documento descrive il lavoro svolto per l'aggiunta di alcune nuove funzionalità agli strumenti sviluppati in precedenza. Ovvero, sono state riviste alcune applicazioni sviluppate in precedenza alla luce di alcune nuove specifiche, è stata realizzata un'applicazione per la validazione dei documenti contenenti le informazioni degli Urban Dataset e sono state integrate sul sito web le applicazioni di validazione e trasformazione in modo da facilitarne l'uso.

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 consiste, quindi, nel 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 la risposta.

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 separai e non pubblici. 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 ed elaborazione delle informazioni 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 di raccogliere 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, al 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 descritte brevemente le nuove specifiche per le applicazioni collegate all'ontologia. Nel capitolo 3 e 4 verranno mostrate le modifiche effettuate ad alcune delle applicazioni sviluppate in precedenza in modo da essere conformi alle nuove specifiche. Nel capitolo 5 verrà mostrata la nuova applicazione di validazione dei documenti contenenti le informazioni sugli Urban Dataset. Nel capitolo 6 verranno elencate le aggiunte apportate al server web. Nell'ultimo capitolo verranno tratte le conclusioni del lavoro svolto.

2 Nuove specifiche

Procedendo con lo sviluppo e il testing delle applicazioni presentate nel corso del precedente periodo di lavoro, sono nate alcune domande e sono emerse nuove problematiche che avrebbero potuto creare dei problemi nel corso della vita delle applicazioni. Di seguito sono presentate le principali aggiunte e modifiche effettuate durante il periodo di estensione del progetto

2.1 Generazione automatica della struttura delle classi

Durante la prima fase di uso delle applicazioni di generazione dei template e trasformazione dei messaggi di errore si è ipotizzato un possibile cambiamento degli schemi dei file XML e JSON e si è convenuto che cambiamenti di questo tipo avrebbero potuto portare a riscrivere parte del codice fin qui sviluppato.

Sebbene la modifica agli schemi si preveda sia un evento raro, questo non è da escludere, soprattutto durante il periodo iniziale di uso, in quanto ci si può accorgere di alcuni errori o mancanze nella modellazione di questi schemi quando si comincia ad usarli in maniera più intensa.

Per far fronte a questa ipotesi e cercare di limitare il lavoro in seguito ad una modifica di questi schemi si è pensato ad almeno due strade. La prima è stata quella di realizzare un sistema di traduzione configurabile dal formato attuale ad un futuro formato degli schemi. Questa soluzione non incontrava i nostri favori dal momento che bisognava prevedere a priori i possibili cambiamenti agli schemi e aveva necessità di definire un linguaggio di mapping delle trasformazioni. Secondo noi avrebbe portato via molto tempo di progetto per avere una cosa molto complicata e non necessariamente resistente alle modifiche future.

Una soluzione più semplice, anche se comunque legata a modifiche del codice è quella che si avvale di strumenti automatici di analisi dei file di schema per generare una struttura delle classi che permetta la compilazione degli schemi. In questo modo è possibile utilizzare librerie già esistenti e testate e richiedere uno sforzo minore di progetto e realizzazione. Allo stesso tempo limitare le modifiche future da apportare al codice in caso di modifica degli schemi.

Per la generazione di file template in XML ed in JSON, essendo stati definiti degli schemi precisi per i formati, si è potuto utilizzare un sistema automatico per la generazione di una struttura di classi che rispecchia la struttura del formato di dati. Questi sistemi permettono una più semplice serializzazione e de-serializzazione dei dati attraverso una struttura di classi che effettua una corrispondenza uno ad uno tra tag XML e campi di una classe. Stesso discorso vale per i sistemi per il parsing di file JSON. Si tratta, in ogni caso, di sistemi molto usati, testati e sperimentati in queste situazioni ed offrono anche il vantaggio di far risaltare eventuali modifiche ai formati di dati tramite la rigenerazione delle classi. Infatti, un IDE evoluto evidenzierebbe subito degli errori di incompatibilità tra il codice sviluppato per un formato precedente degli schemi (sia XML che JSON) e le classi generate tenendo conto del nuovo formato. In questo modo ci si troverebbe subito indicate le incompatibilità e le modifiche da effettuare.

Dal momento che la libreria di accesso all’ontologia, sviluppata in precedenza, è stata sviluppata in JAVA, per questo scopo è stato scelto per la generazione di classi a partire dallo schema XML la libreria JAXB [1] che è inclusa nella distribuzione di base di JAVA e per la generazione delle classi che rispecchiano la struttura del corrispondente formato, è stato usato il compilatore xjc. Tale software tramite un semplice comando prende in ingresso il file XSD contenente lo schema e genera la struttura delle classi che rispecchia la struttura e fornisce anche tutti i metodi necessari per accedere e modificare tali valori della struttura, ovvero i singoli campi delle diverse classi. Questa libreria fornisce anche strumenti automatici per ricostruire la struttura delle classi partendo da un file XML letto (de-serializzazione) o compilare su file XML la struttura di classi creata via codice.

Stesso discorso è stato fatto per la rappresentazione JSON dei dati. In particolare, è stata scelta la libreria JSONSCHEMA2POJO [2]. Questa libreria offre alcuni vantaggi dovuti alla configurabilità. In particolare, è possibile definire a priori alcune caratteristiche e comportamenti che deve avere il (de)serializzatore, ovvero la libreria che si occuperò di fare il mapping tra classi e dati su file. È possibile configurare le classi generate per essere usate con un particolare (de)serivalizzatore. Questa libreria, infatti, permette la generazione di classi compatibili con due dei più usati (de)serialzzatori per JSON ovvero Jackson [3] e Gson. In seguito ad

alcune prove è stato scelto di usare a questo scopo la libreria Jackson2 perché permette una più semplice configurazione di alcune caratteristiche del risultato della serializzazione, ovvero della forma del file JSON di output.

2.2 Messaggi di errore

Un altro problema riscontrato durante la fase di valutazione del risultato delle precedenti versioni è relativo ai messaggi che i software forniscono all'utente. I feedback dati dagli utenti hanno portato ad una riscrittura delle parti che generano messaggi di errore in modo da fornire informazioni comprensibili anche a chi non ha a disposizione il codice da analizzare o non ha chiaro come leggere i messaggi di errore che vengono generati dalla macchina virtuale Java (JVM).

I vari software sviluppati precedentemente stampavano a video un messaggio di errore non particolarmente informativo. Durante l'uso che se ne è iniziato a fare sono stati individuati i principali casi di errore e sono stati riscritti i messaggi. Adesso non mostrano i messaggi della JVM ma dei messaggi più esplicativi pe l'utente comune da cui è possibile capire più facilmente cosa è andato storto e come correggere. Un'altra particolarità aggiunta è che questi messaggi non sono stampati a video dalle classi da cui scaturiscono. Quello che si è fatto e creare una serie di eccezioni a seconda dell'errore. Tali eccezioni vengono inoltrate al chiamante che decide come visualizzarle.

Questa scelta è stata dettata dalla necessità di integrare all'interno del sito web questi servizi. Il sito web non ha uno schermo su cui inviare l'errore. Il sito web che offre i servizi ha un sistema di logging su file di testo e non sarebbe stato compatibile con la stampa a video dei messaggi di errore. La ricezione dei messaggi attraverso eccezioni permette la configurabilità della gestione di questi messaggi. In questo modo è possibile salvare sul file di logging questi messaggi e decidere anche una politica più fine di salvataggio (ovvero mostrare solo parte dei messaggi a seconda della tipologia degli stessi). Nel caso delle applicazioni usate singolarmente da riga di comando si può mantenere la stampa a video di tali messaggi semplicemente gestendo le eccezioni all'interno della funzione main che chiama le varie parti del programma per portare a termine il lavoro richiesto.

2.3 Modifiche ulteriori

Oltre a quanto presentato finora, è stata richiesta una nuova applicazione che validasse i file di messaggio XML e JSON sia con XSD e JSONSchema rispettivamente e verso i file Schematron. La validazione doveva funzionare sia singolarmente sia all'interno del sistema di traduzione da XML a JSON e viceversa. Tale validazione è stata quindi integrata anche all'interno del trasformatore.

Tutte queste applicazioni sono state integrate anche nel sito web. Dalla sua interfaccia è adesso possibile accedere a questi servizi tramite pochi click del mouse. Adesso, infatti, sono presenti i collegamenti che permettono di invocare sia la validazione di file di Urban Dataset che la trasformazione tra XML e JSON.

Dal sito web è possibile accedere anche alla documentazione dell'ontologia. Per ogni voce di Urban Dataset è possibile generare una pagina con la relativa documentazione oppure generare la documentazione completa per tutte le voci. Il risultato viene mostrato in una uova pagina del browser.

3 Template di messaggio di un Urban Dataset

In questo capitolo è descritto il lavoro svolto per la modifica del software responsabile della generazione del template XML a partire dei dati dell'ontologia secondo le specifiche definite nel XSD che definisce il formato.

3.1 Descrizione della nuova architettura

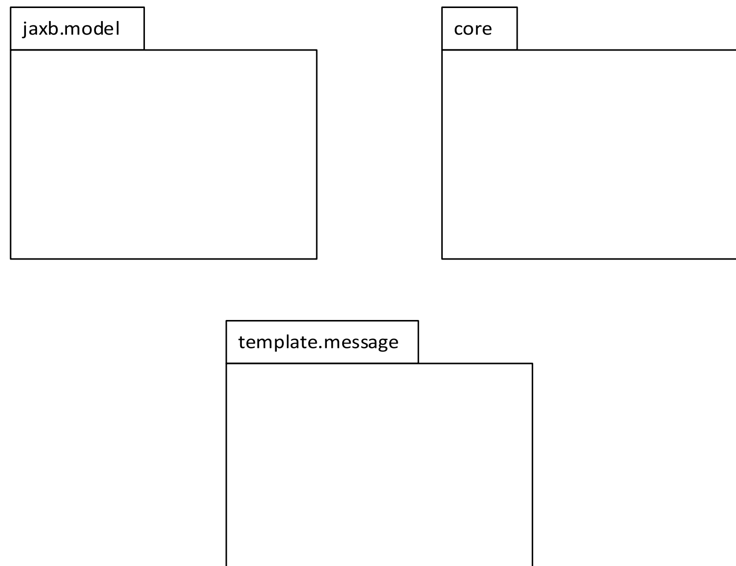

Figura 1. Organizzazione dei package

Il software è stato diviso in tre package, come mostrato in Figura 1. Il package *jaxb.model* contiene le classi genate automaticamente da xjc, come indicato nel capitolo precedente. 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.

Partendo dalle classi generate da xjc della libreria JAXB di JAVA, queste sono state generate senza impostare particolari parametri. Xjc consente di configurare alcune cose nella generazione delle classi che modellano la struttura del fil XML; nel nostro caso si è scelto di impostare solamente il package da usare, ovvero *jaxb.model*.

In Figura 2 è mostrato parte dello schema delle classi generate. In particolare, è stata definita una classe *UrbanDatasetType* che modella la radice dello schema del file XML. Questa classe contiene riferimenti alle tre sotto parti dello schema ovvero alle sezioni *specification*, *context* e *values*. Ciò è fatto modellando tre classi che le mappano ovvero *SpecificationType*, *ContextType* e *ValuesType*. Ognuna di queste ha a sua volta riferimenti a classi costruite in base alle sotto parti che la compongono che in Figura 2 non sono state riportate per questioni di sintesi. In ogni caso, dallo schema qui presentato, si può intuire che nel caso si tratta di informazioni non articolate, il contenuto di un tag XML è stato modellato con una semplice stringa. In casi più complessi è stata costruita una nuova classe contenente le vari sotto parti del tag.

La classe *ObjectFacory* è, invece, usata per creare un template XML di tipo *JAXBElement* a partire dall'oggetto *UrbanDatasetType* passato come parametro. L'oggetto *JAXBElement* risultante viene usato per generare il template XML sottoforma di stringa da salvare su file di testo. Tutte queste scelte sono state operate autonomamente dalla libreria usata per la generazione delle classi.

Nel package *core*, sono presenti due classi (Figura 3). La prima è l'interfaccia *IBuilderTemplate* che fornisce quattro 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 3, 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.

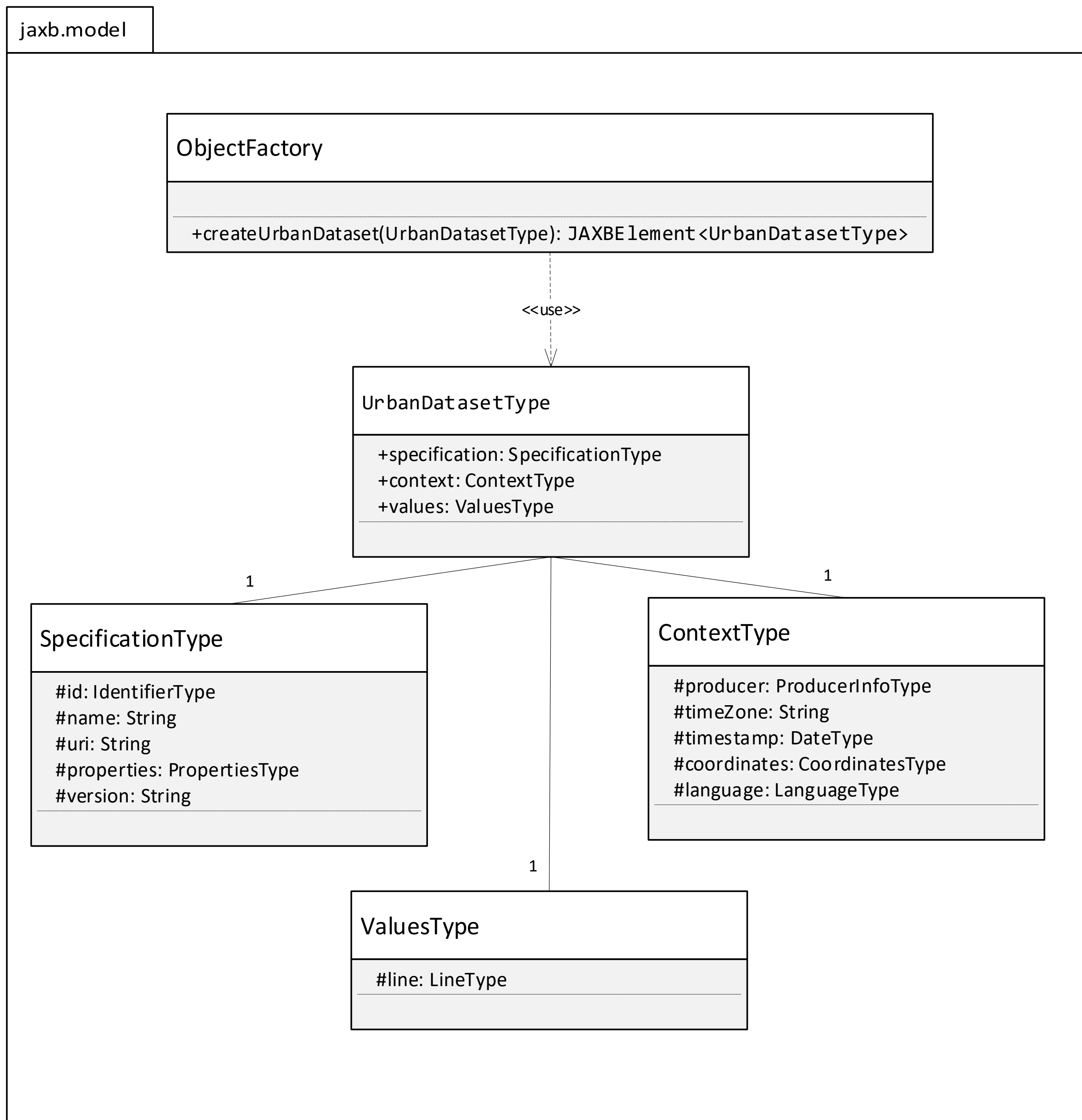

Figura 2. Diagramma delle classi generate da JAXB

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 sottoproprietà delle proprietà *hasUrbanDatasetProperty* e *hasContextProperty* sottoforma di list. Accetta come parametro un campo booleano per indicare se la ricerca deve essere ricorsiva alle sottoproprietà o no. Il metodo

getFirstLevelPropName() ritorna la lista di proprietà escluse le proprietà speciali *coordinate*, *period* e *DataDescription*.

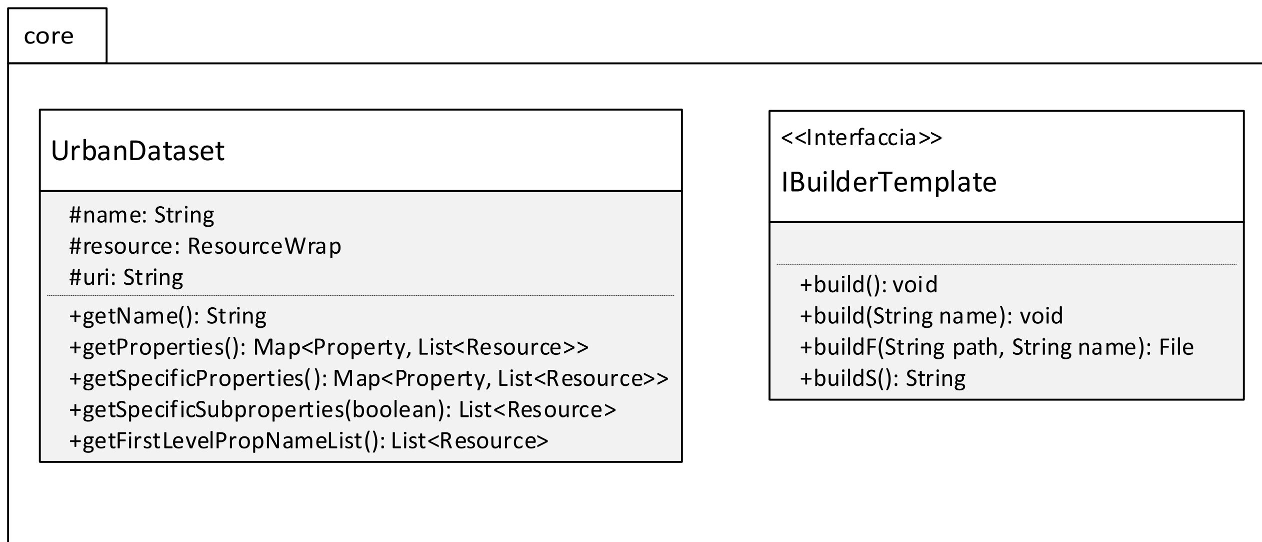

Figura 3. Diagramma delle classi del package *core*

In Figura 4 è presente il diagramma delle classi del package *template.message*. Da tale grafico si può notare che la classe *MessageBuilder* 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 *tempalteIDProperties()* 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 file XML, ovvero *UrbanDatasetType*, ed un oggetto *Marshaller*, proprio di JAXB, che 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.

L'oggetto istanza di *MessageBuilder*, prima di effettuare la serializzazione dovrà, ovviamente, provvedere a costruire la struttura di oggetti che fa capo ad *UrbanDatasetType*. Per fare ciò, l'oggetto istanza di *MessageBuilder* si occupa di costruire gli oggetti *SpecificationSection*, *ContextSection* e *ValueSection*. Ognuno di essi provvederà a costruire la sezione del messaggio che gli compete prendendo come parametri di costruzione le strutture dati che contengono le informazioni necessarie. Tra queste informazioni c'è la struttura delle proprietà specifiche dell'Urban Dataset di cui necessitano sia *SpecificationSection* che *ValuesSection*. Per questo viene costruita una struttura che contiene questa informazione, istanza della classe *PropertyDef*, e viene passata alle istanze per completare l'operazione.

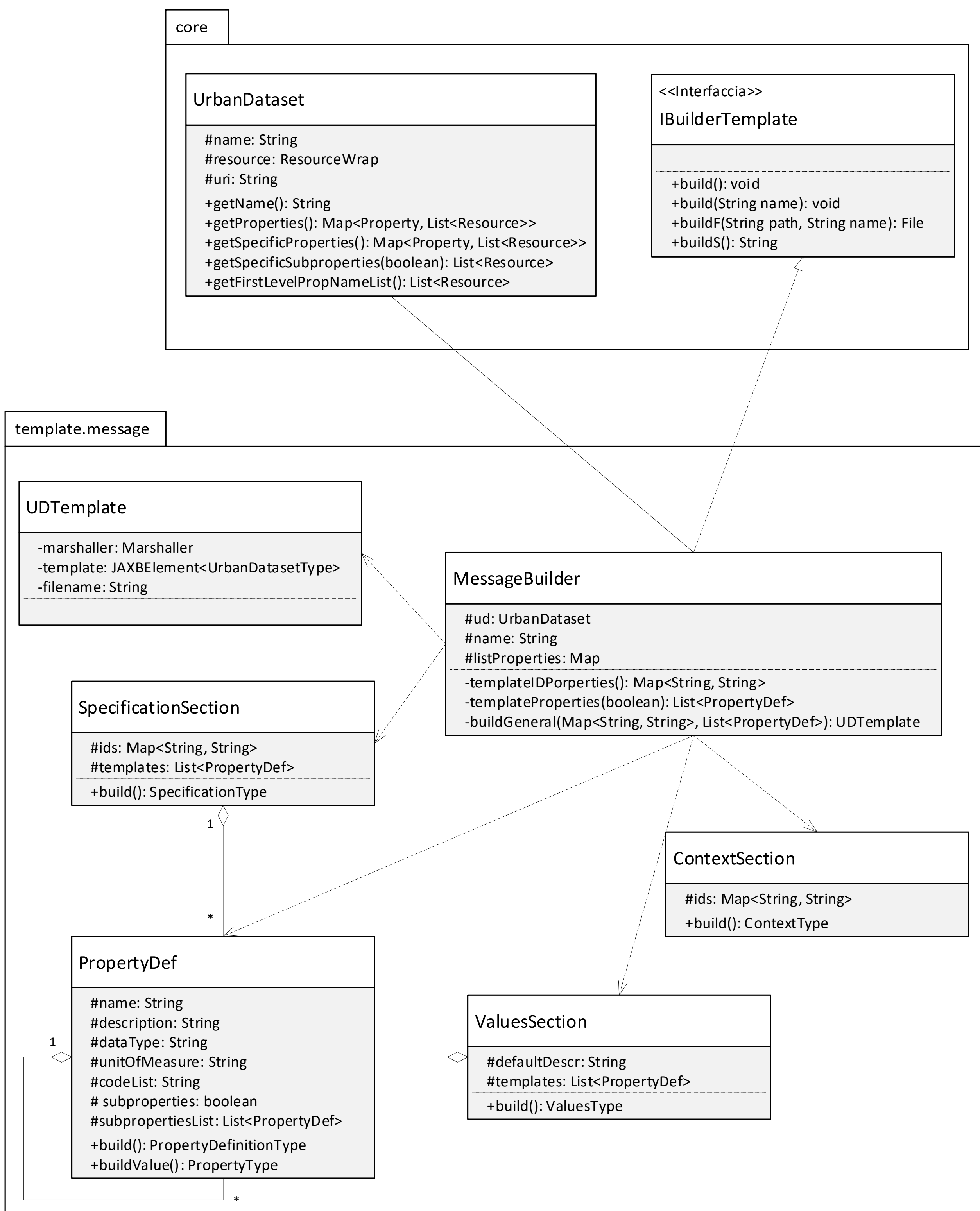

Figura 4. Diagramma delle classi del package *template.message*

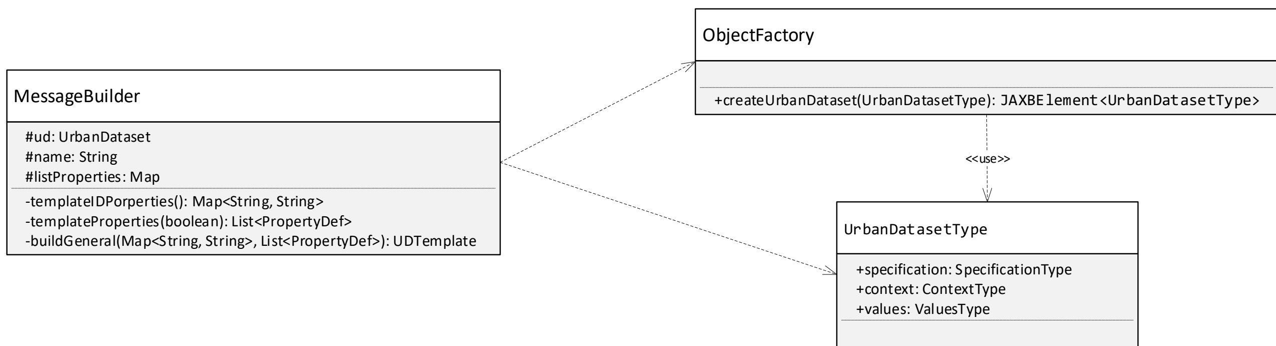

Figura 5. Relazione tra *MessageBuilder* e classi JAXB generate

Una volta costruite le singole parti del template XML, l'istanza di *MessageBuilder* provvede ad unirle all'interno del nuovo oggetto che è istanza di *UrbanDatasetType* (Figura 5). A questo punto è possibile creare un JAXBElement che può essere usato dal marshaller per generare la stringa che rispecchia la struttura del file XML attesa.

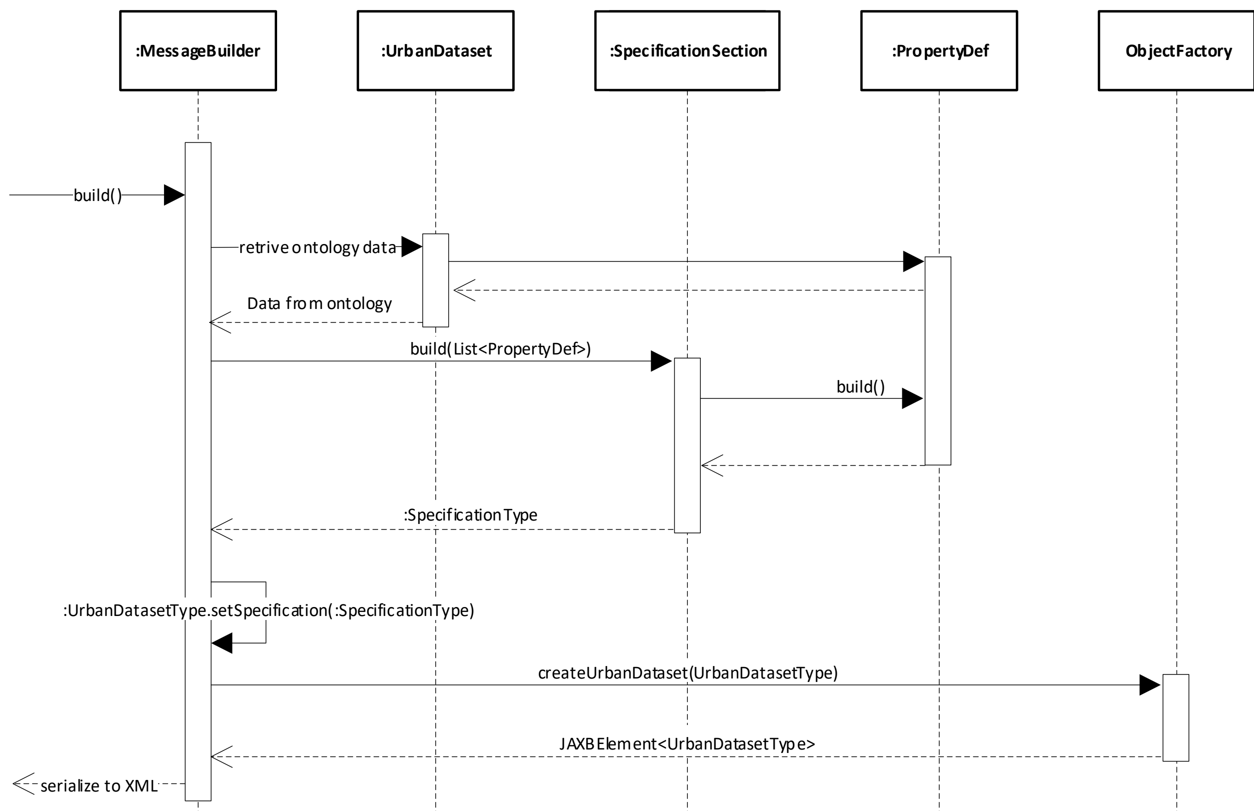

Figura 6. Diagramma di sequenza della creazione di un template XML

In Figura 6 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 e la creazione di diversi oggetti *PropertyDef* quante sono le proprietà dell'Urban Dataset e delle altre informazioni necessarie a completare il template. 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 6 è mostrato solo il caso di *SpecificationSection* che a sua volta delega la sezione delle proprietà al relativo metodo di ciascun

PropertyDef). Una volta riunite le tre sottosezioni nell'oggetto radice istanza di *UrbanDatasetType*, richiede alla factory generata dalla libreria JAXB di costruire l'oggetto che potrà essere serializzato in stringa o file dal Marshaller di JAXB.

3.2 Interfaccia di invocazione

Come già detto, l'interfaccia di invocazione da riga di comando è stata modificata e resa più coerente tra le varie applicazioni.

Dal momento che questa applicazione è integrata con il generatore di schematron, per distinguere tra XML e Schemarton si è inserito un parametro obbligatorio per distinguere l'esecuzione. Per indicare la generazione di XML deve essere inserito il parametro **-t**, oppure **--XMLmessage**, (***-s,*** o in alternativa ***–schematron****, per la generazione di file Schematron*). Per indicare la generazione del singolo Urban Dataset preceduto dal parametro ***-U,*** oppure ***--Urbandataset***. Al contrario, per indicare una lista di Urban Dataset da elaborare si è deciso di passare all'applicazione un file di testo contenente per ogni riga il nome di un Urban Dataset di cui richiedere il template XML indicando il nome del file preceduto dal parametro ***-F*** oppure ***--file***. È stata aggiunta anche la possibilità di indicare opzionalmente una directory dove effettuare il salvataggio dei risultati. Inserendo il parametro **-o**, oppure **--outdir**, è possibile indicare dove verranno creati i file risultanti dall'elaborazione. In caso contrario, l'output dell'applicazione viene generato all'interno della cartella /output ed ogni file avrà il nome dell'Urban Dataset di cui contiene il template.

4 Trasformatore di formato dei template di messaggio

In questo capitolo è descritto il lavoro svolto per la modifica del software responsabile della trasformazione dei documenti rappresentati gli Urban Dataset da XML a JSON e viceversa secondo le specifiche definite nei file XSD e JSONSchema che ne definiscono il formato.

4.1 Descrizione della nuova architettura

Il software è stato diviso in tre package, come mostrato in Figura 7. Il package *jsonschema.model* contiene le classi genate automaticamente dalla libreria JSONSCHEMA2POJO, come indicato in precedenza. I package *convert.xml2json* e *convert.json2xml* contengono le classi che svolgono le funzioni necessarie per la lettura e scrittura di documenti JSON per la trasformazione dei messaggi contenenti le informazioni relative agli Urban Dataset da XML a JSON e viceversa.

Partendo dalle classi generate della libreria JSONSCHEMA2POJO, queste sono state generate scegliendo Jackson come tool per la serializzazione (ovvero per rappresentare come stringa la struttura di classi che modella la struttura del file JSON) e non includendo i campi vuoti come è il caso delle proprietà che possono essere non presenti nel documento. Una cosa che si è scelta è relativa al package con cui le classi dovevano essere generate, nostro caso si è scelto di usare il package *jsonschema.model*.

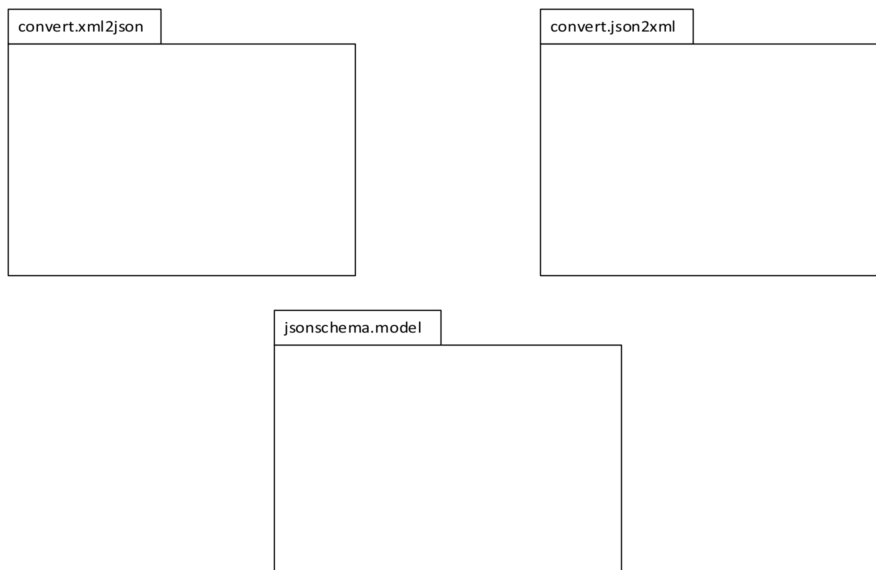

Figura 7. Organizzazione dei package

In Figura 8 è mostrato parte dello schema delle classi generate da JSONSCHEMA2POJO. In particolare, è stata definita una classe *ScpsUrbandatasetSchema10* che modella la radice dello schema del file JSON e che ha all'interno una unica sezione rappresentata dalla classe *UrbanDataset*. Questa classe contiene riferimenti alle tre sotto parti dello schema ovvero alle sezioni *specification*, *context* e *values*. Ciò è fatto modellando tre classi che le mappano ovvero *Specification*, *Context* e *Values*. Ognuna di queste ha a sua volta riferimenti a classi costruite in base alle sotto parti che la compongono e che non sono state riportate per questioni di sintesi. In ogni caso, dallo schema qui presentato, si può intuire che nel caso si tratti di informazioni non articolate, il contenuto di un campo JSON è stato modellato con un tipo di dato semplice. In casi più complessi è stata costruita una nuova classe contenente le vari sotto parti del campo.

All'interno del package *convert.xml2json* (Figura 9) sono presenti le classi che operano la trasformazione delle informazioni da formato XML a formato JSON. La classe *JsonBuilder* implementa l'interfaccia *IBuilderTemplate* allo stesso modo della classe *MessageBuilder* vista nel capitolo precedente. Nella classe *JsonBuilder* sono presenti i metodi per effettuare la trasformazione. La classe prende in ingresso, tramite il costruttore, il riferimento alla radice della struttura di oggetti costruita sulla base del parsing effettuato usando le classi generate a partire dal documento XSD. All'interno del metodo privato *buildJson* vengono invocati i metodi per trasformare le varie sezioni del documento e presenti nelle classi

SpecificationJsonBuilder, *PropertyJsonBuilder*, *ContextJsonBuilder* e *ValuesJsonBuilder*. In questo modo viene creata una struttura di oggetti conforme alla struttura del docuemnto JSON e che mappa tutte le informazioni presente nel documento XML. Il risultato può essere facilmente serializzato come stringa attraverso le primitive di Jackson.

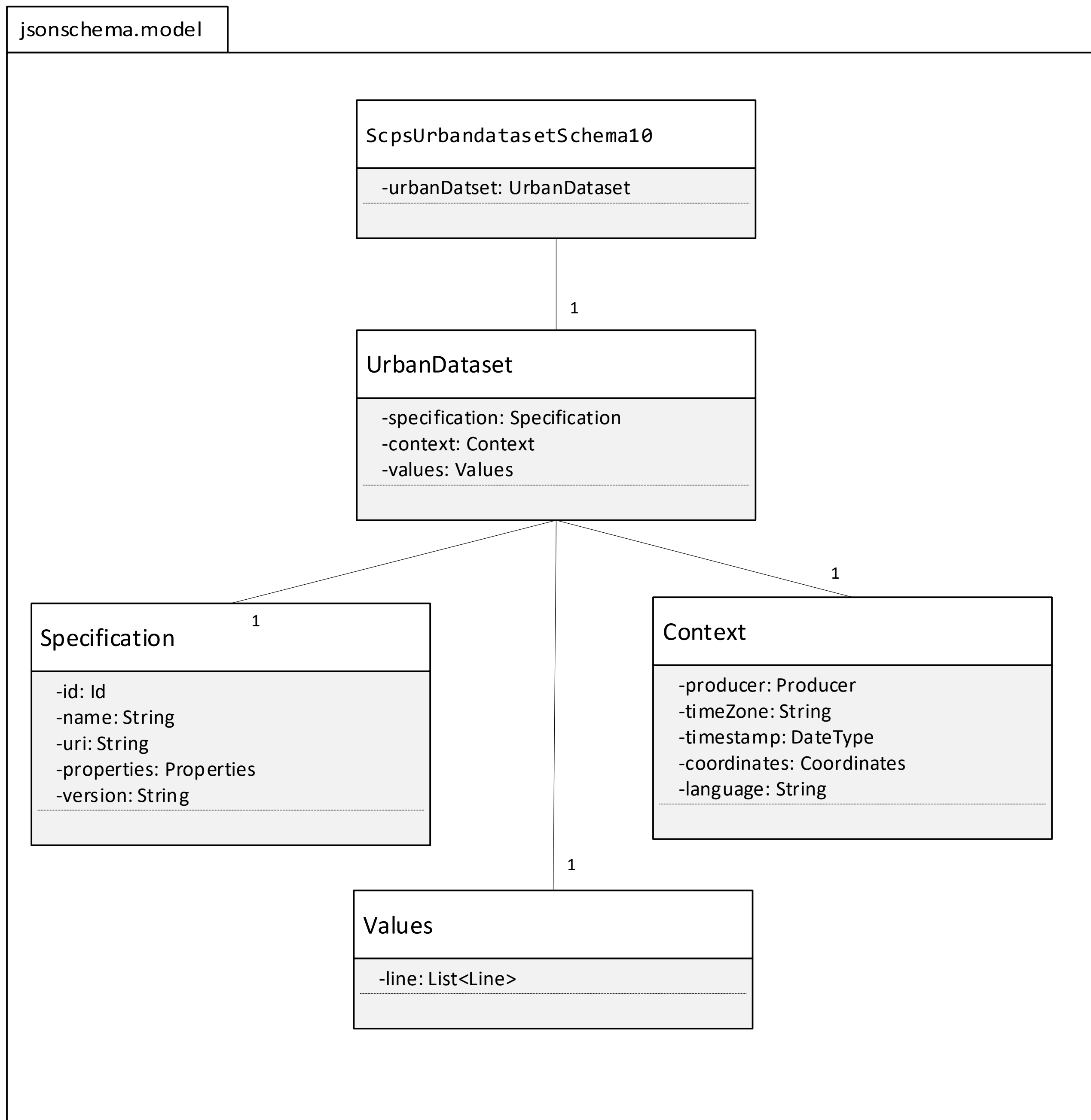

Figura 8. Diagramma delle classi generate da JSONSCHEMA2POJO

La serializzazione effettiva verrà gestita a seconda del metodo dell'interfaccia *IBuilderTemplate* chiamato. L'elaborazione effettuata dal metodo privato buildJson di *JsonBuilder* da come risultato un oggetto di tipo UDJsonTemplate che contiene le impostazioni per la serializzazione su file che verranno eseguite dai metodi chiamanti sulla base delle opzioni utente.

In Figura 10 è mostrato il diagramma sequenza delle principali azioni che vengono svolte durante la trasformazione di un documento XML in formato JSON. La richiesta di *build()* di un documento JSON trasformato è causa prima di tutto del recupero delle informazioni dal file XML e la creazione degli oggetti di tipo *Specification*, *Context* e *Values*. A questi oggetti viene richiesta la generazione della corrisopndente

sezione JSON sulla base della corrispondente sezione XML che viene passata come parametro al momento della costruzione dell'oggetto.

Il risultato viene poi serializzato in stringa per mezzi della libreria Jackson e salvato su file secondo le richieste dell'utente.

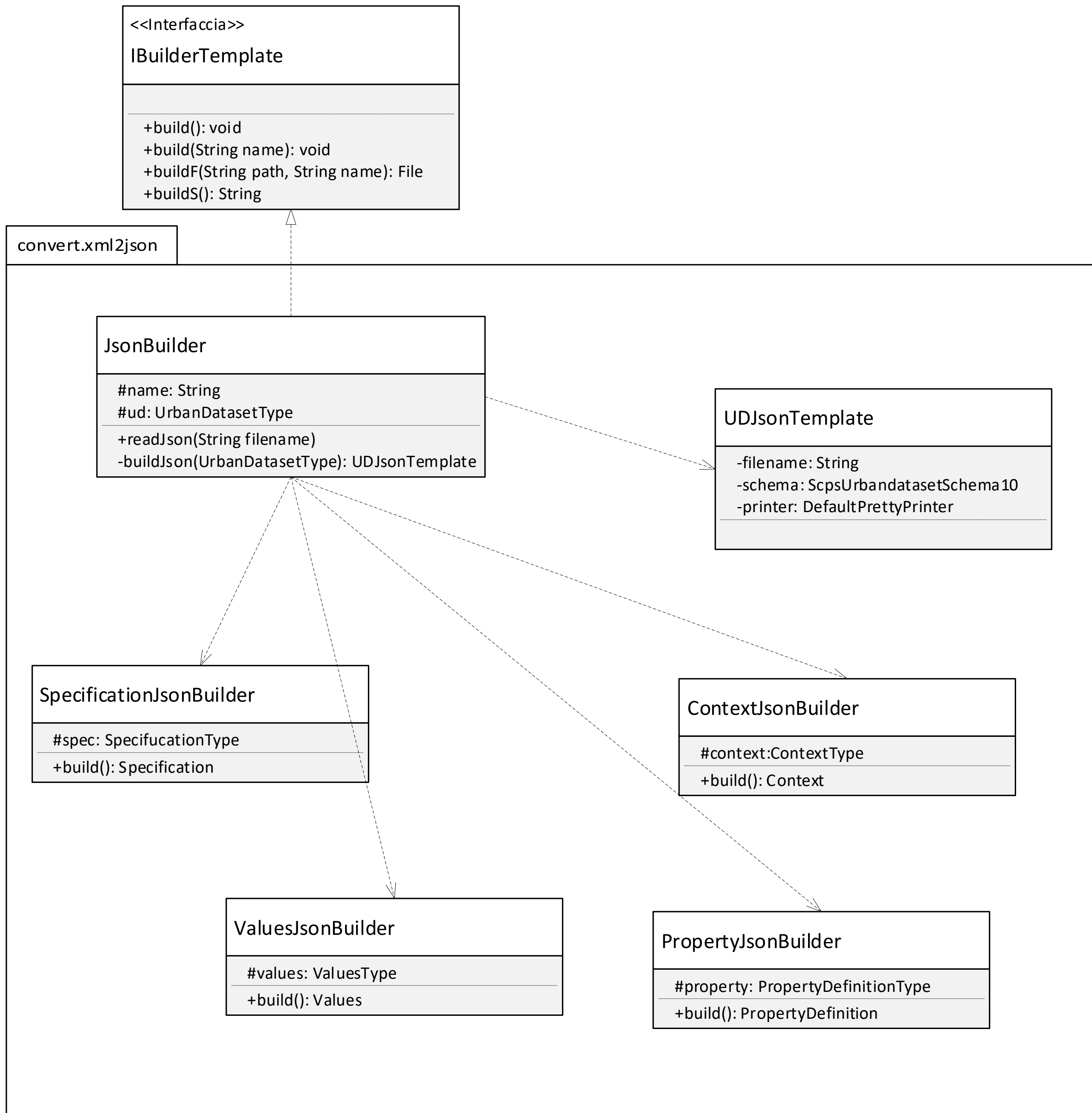

Figura 9. Diagramma delle classi del package convert.xml2json

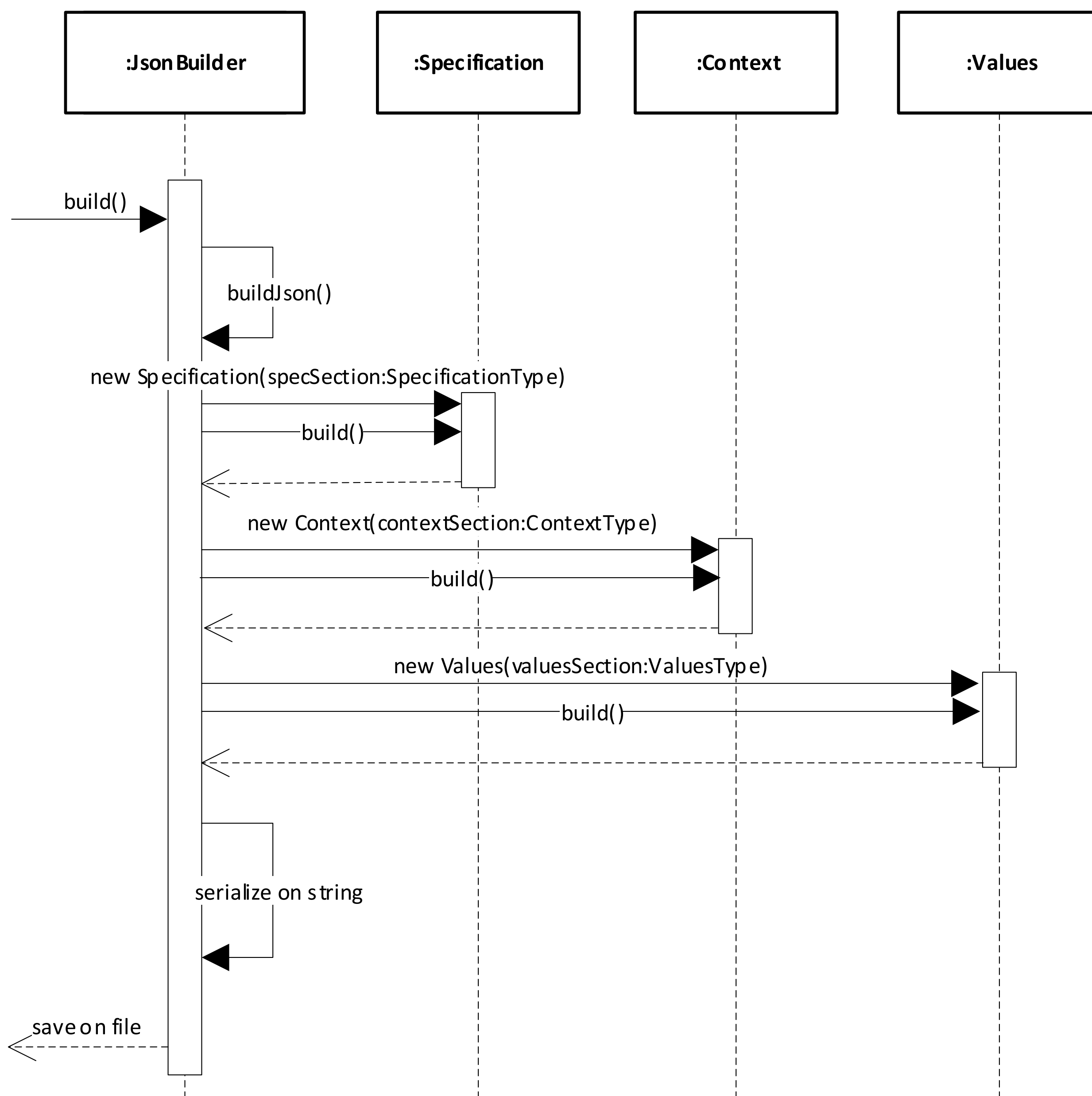

Figura 10. Diagramma di sequenza della trasformazione da XML a JSON

Il caso della trasformazione da JSON a XML ha un comportamento molto simile. La differenza principale è che si è voluto riutilizzare, almeno in parte, le classi già sviluppate per il generatore di template. Pertanto, il meccanismo di trasformazione procede in modo analogo al caso appena visto (ovvero da XML a JSON). Il primo passo e effettuare un parsing del file JSON utilizzando lo schema delle classi generate a partire dal file JSONSchema. Dopo di ciò, si estraggono le informazioni importanti e si passa il risultato alle classi *SpecificationSection* e *ContextSection* del package template.message che provvedono a generare la parte di documento XML che gli compete. Il caso della sezione Values è leggermente diverso in quanto in precedenza non era previsto l'inserimento di valori che nell'ontologia non sono presenti.

Per ovviare a questo è stata definita la nuova classe *ValuesXmlBuilder* nel package convert.json2xml (Figura 11) per fare in modo che anche i valori dei dati raccolti vengano trasformati correttamente nel nuovo formato. Ad orchestrare tutto il processo c'è la classe XmlBuilder che implementa l'interfaccia *IBuilderTemplate*.

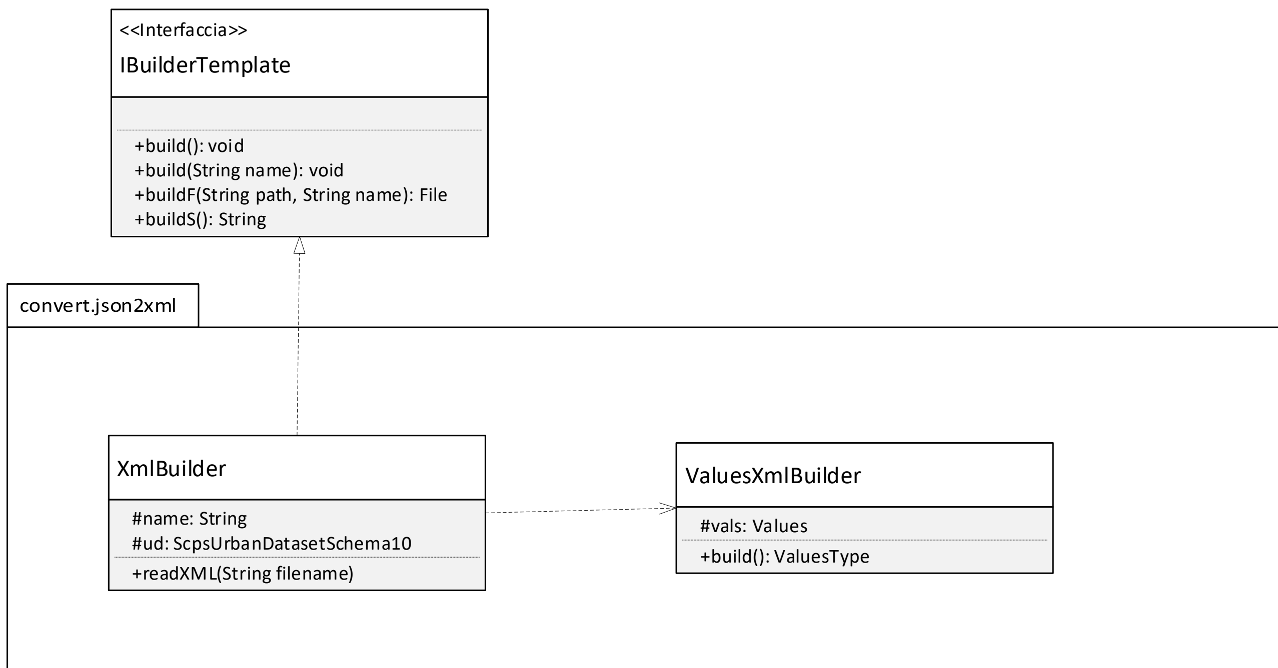

Figura 11. Diagramma delle classi del package convert.json2xml

Il processo di trasformazione avviene in maniera simile a quanto visto nel caso precedente.

4.2 Interfaccia di invocazione

Anche in questo caso, l'interfaccia di invocazione da riga di comando è stata modificata e resa più coerente tra le varie applicazioni.

Dal momento che questa applicazione può effettuare la trasformazione tra XML e JSON in entrambi i sensi sono stati previsti due parametri per indicare la scelta. Questi sono: **-j**, oppure **--toJSON**, per indicare la volontà di trasformare un file XML in JSON; **-x**, oppure **--toXML**, per il processo inverso.

Dal momento che nella fase di trasformazione è stata integrata anche la validazione (mostrata nel capitolo successivo) è necessario inserire anche gli schemi verso cui i file devono essere validati. In particolare, sono previste due validazioni diverse: una verso lo schema XSD o JSONSchema a seconda che il file sorgente sia XML o JSON, l'altra verso il file Schematron. Per questo sono previsti due parametri opzionali che indicano gli schemi da usare. Viene usato **-sc**, o in alternativa **--schema**, seguito dal nome del file per passare il file XSD o JSONSchema, mentre viene usato **-sch**, o **--schematron**, seguito dal nome del file per passare il file Schematron. In assenza di questi il sistema usa gli schemi di default impostati. Nel caso dello Schematron, provvederà a generare lo schema corrispondente sulla base del nome di Urban Dataset contenuto nel file di Urban Dataset che viene passato senza parametri.

Infine, è possibile specificare anche una directory in cui salvare il risultato della trasformazione attraverso il parametro **-o**, oppure **--output**, seguito dal path.

5 Sistema di validazione dei messaggi

In questo capitolo è descritto il lavoro svolto per la realizzazione del software responsabile della validazione di documenti, contenenti informazioni sugli Urban Dataset, in formato XML e JSON.

5.1 Caratteristiche funzionali

I documenti XML e JSON hanno una struttura che può essere descritta attraverso regole precise e definite all'interno di file di tipo XSD e JSONSchema, rispettivamente. Attraverso questi schemi è possibile verificare che i file XML e JSON siano conformi alle regole e quindi validi. In questi file di schema è definita solo la struttura del file e quindi la sintassi. Nel caso del formato XML, possibile validare anche che il contenuto sia conforme a delle specifiche (ad esempio che è presente una proprietà opzionale ne è presente anche un'altra). In questo modo è possibile effettuare una validazione di tipo semantico del contenuto. La definizione di tali regole è fatta definendole all'interno di un file di tipo Schematron. La validazione verifica che queste regole siano rispettate.

Il lavoro che deve svolgere questo software è semplicemente la verifica di documenti verso gli schemi XSD, JSONSchema e Schematron e restituire un messaggio di successo o una indicazione degli errori presenti.

5.2 Descrizione dei requisisti

I formati XML e JSON per rappresentare le informazioni relative agli Urban Dataset sono definiti da due file, rispettivamente XSD e JSONSchema che contengono le regole sintattiche per la redazione di tali documenti. L'applicazione deve prendere in input i file sia di schema sia quelli contenti le informazioni dell'Urban Dataset e verificare che la sintassi sia congruente con lo schema. In caso di errore deve essere restituita la motivazione (o le motivazioni) del fallimento ed eventualmente salvare su fil il risultato.

I documenti in formato XML devono poter essere validati con un file di tipo Schematron. Tale file contiene le regole logiche per cui devono essere presenti determinate combinazioni di campi in base al tipo di Urban Dataset. Il software deve fornire la possibilità di validare gli Urban Dataset con questo schema e restituire eventuali messaggi di errore.

I file di schema devono essere opzionali. In caso di loro assenza, nel caso di XSD e JSONSchema, si devono usare quelli di default che vanno quindi indicati a livello di configurazione dell'applicazione. Nel caso Schematron, deve essere analizzato il documento xml, recuperato il nome dell'Urban Dataset e generato il file Schematron corrispondente che verrà usato per la validazione.

5.3 Descrizione dell'architettura

Il software è composto da un unico package (Figura 12) in cui è stata definita l'interfaccia *IValidate*. Tale interfaccia definisce i principali metodi per poter invocare la validazione dei file come descritto in precedenza. Questa interfaccia è implementata da altre tre classi, una per ogni possibile tipo di validazione, ovvero *XMLValidator*, *JSONValidator* e *SCHValidator*. Ognuna di queste classi permette la validazione dei documenti e genera messaggi di errore, anche sottoforma di file, in caso ce ne siano.

Nel caso XML con XSD, la validazione viene effettuata attraverso gli strumenti forniti dalla libreria SAX fornita insieme alla JVM. Tale libreria fornisce in output, in caso di errori, delle eccezioni che vengono catturate all'interno della classe *XMLValidator* per poter presentare all'utente i motivi dell'eventuale fallimento ed una indicazione della posizione all'interno del file da validare. Gli errori, se presenti, sono mostrati a schermo e salvati su file di testo.

Nel caso JSON con JSONSchema, la validazione è effettuata attraverso la libreria JSON Schema Validator [4] che fornisce, come nel caso precedente una comoda interfaccia per la validazione e genera eccezioni, catturate dalla classe *JSONValidator*, per gli errori che si presentano con indicazioni sul tipo e sulla posizione dell'errore. Gli errori, come nel caso XML, sono mostrati a schermo e salvati su file di testo.

Nel caso della validazione XML con file Schematron, la validazione è effettuata attraverso delle trasformazioni XSL e definita nella classe *SCHValidator*. Lo standard di validazione Schematron è definito attraverso l'applicazione in cascata di diverse trasformazioni XSL sul file Schematron da usare per la validazione ed alla fine applicare la trasformazione del file risultante al file XML da validare. Il risultato è un nuovo file XML

contente il risultato della validazione. Per applicare le diverse trasformazioni XSL è stata definita una lista dei file XSL da applicare in sequenza e contenuta nella variabile *xslSchemas* (come mostrato in Figura 12) usata la libreria Saxon [5], che fornisce le primitive per effettuare questa operazione. Il file XML risultante è poi analizzato per estrarre i messaggi di errore. Questo file è ulteriormente trasformato per generare un documento HTML contenente il risultato della validazione in modo da fornire un output grafico più accattivante.

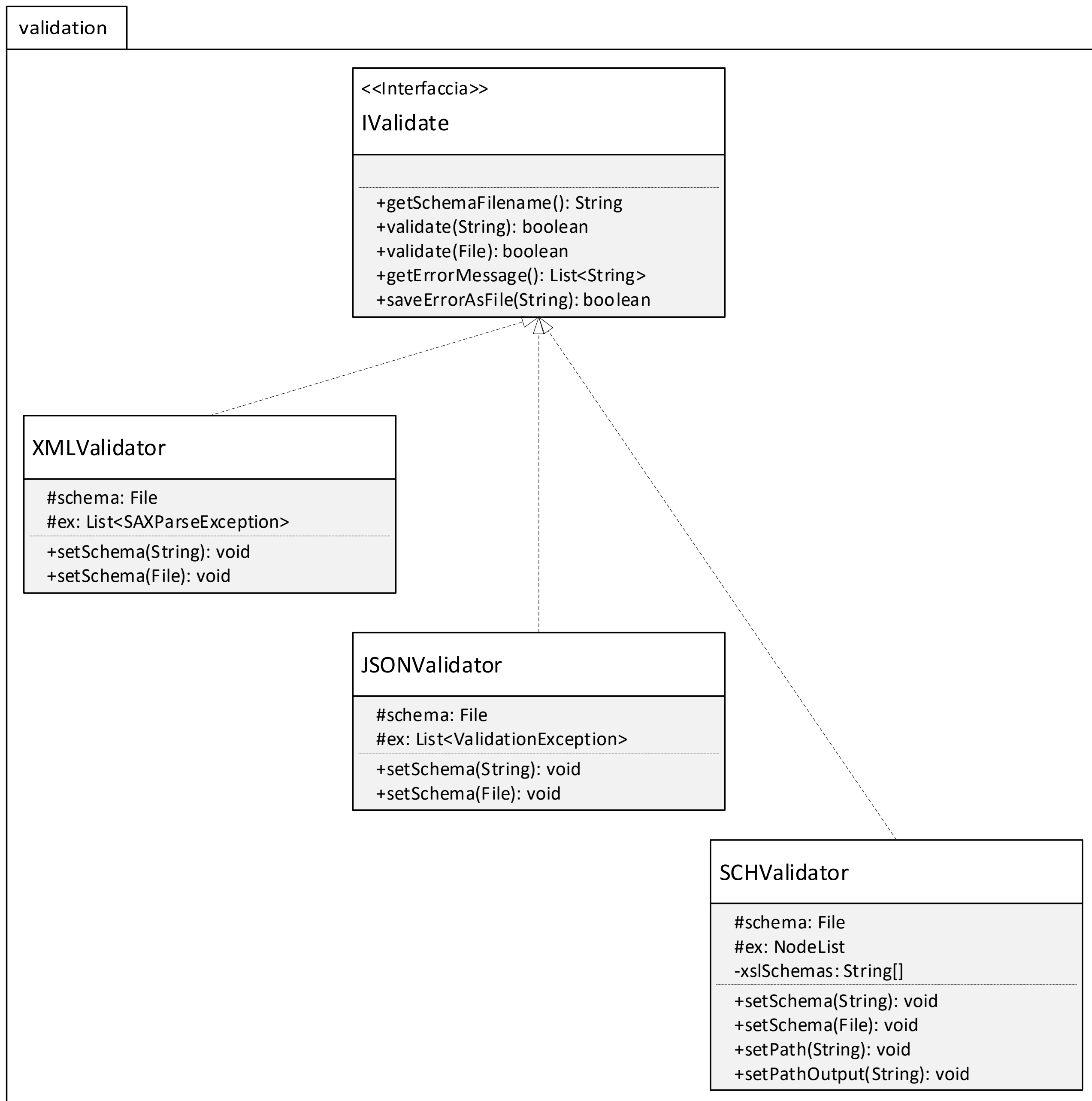

Figura 12. Diagramma delle classi del package validation

5.4 Interfaccia di invocazione

Questa applicazione permette la validazione secondo diverse modalità, per cui è prevista la presenza di tre parametri da usare in alternativa per indicare il tipo di validazione da usare. In particolare, sono previsti i parametri: -x, oppure **--xsdschema**, per validare file XML con XSD; **-j**, oppure **--jsonschema**, per validare file JSON con JSONSchema; **-s**, oppure **--schematron**, per validare file XML con Schematron.

Il parametro **-sc**, oppure **--schema**, può essere usato per passare gli schemi da usare nella validazione, in caso di loro assenza vengono usati gli schemi di default indicati nel file di configurazione per il caso XSD e JSONSchema. Nel caso Schematron ne verrà generato uno, estraendo il nome dell'Urban Dataset definito nel file da validare, utilizzando l'applicazione per la generazione di documenti Schematron sviluppata in precedenza e descritta nel rapporto tecnico del terzo anno di progetto.

6 Nuova versione dell'interfaccia web

In questo capitolo è descritto il lavoro svolto per l'estensione del software responsabile della presentazione del contenuto dell'ontologia attraverso interfaccia web. Oltre a quanto presentato nel report precedente, sono state aggiunte le funzionalità di validazione e trasformazione dei documenti e la generazione di documentazione relativa agli Urban Dataset a partire dalle informazioni presenti nell'ontologia.

6.1 Validazione

L'applicazione di validazione è stata aggiunta al servizio web tramite una pagina apposita (Figura 13). La nuova pagina mostra un campo per il caricamento del file da validare ed una *combo box* per scegliere lo schema da usare per la validazione tra una lista di quelli disponibili sul server. Il contenuto della lista dipende dall'estensione del file inviato. Se l'estensione è di tipo XML, saranno mostrati solo gli schemi XSD. Se l'estensione è di tipo JSON, saranno mostrati solo gli schemi JSONSchema presenti sul server. Gli schemi disponibili sono configurabili attraverso un file di test presente sul server in cui vanno indicati gli schemi secondo un formato specifico.

Figura 13. Schermata della nuova pagina di validazione

Il processo di validazione prevede anche di effettuare la validazione Schematron in automatico. Pertanto, dopo l'invio del file da validare al server, nel caso sia richiesta la validazione di un documento XML, questo verrà prima validato con lo schema XSD selezionato ed in seguito ne verrà analizzato il contenuto per estrarre il nome dell'Urban Dataset a cui fa riferimento. A questo punto viene generato il file Schematron corrispondente ed effettuata la validazione. Nel caso di validazione di un file JSON, questo verrà validato con un JSONSchema, verrà trasformato in XML e a questo punto si procederà con la validazione Schematron come il caso precedente.

All'interno della pagina web verranno mostrati eventuali messaggi di errore per indicare le motivazioni di fallimento nei vari passaggi di validazione.

6.2 Trasformazione

Al servizio web è stata aggiunta anche una pagina per la trasformazione. Questa pagina integra molte funzionalità della validazione in quanto, prima della trasformazione viene sempre effettuata la validazione. In questo caso viene effettuata la validazione XSD e JSONSchema in questo ordine nel caso della trasformazione da XML a JSON, in ordine inverso in caso contrario. La validazione Schematron viene fatta prima della trasformazione nel caso il file di origine è di tipo XML, altrimenti la validazione Schematron deve essere fatta dopo la trasformazione dal momento che questo tipo di validazione è attuabile solo con ile XML. Il processo si conclude con la validazione XSD o JSONSchema del file trasformato. Naturalmente al validazione viene scelta secondo l'output della trasformazione: se il risultato è un JSON allora sarà validato con JSONSchema, XSD altrimenti.

Figura 14. Schermata della nuova pagina di trasformazione

Il processo viene interrotto se una delle validazioni dovesse fallire mostrando nella pagina web gli errori incontrati in modo da essere di aiuto all'utente ed apportare correzioni. In questo caso non viene mostrata la scelta di più schemi per la validazione ma vengono usati gli schemi indicati come principali nelle opzioni del server.

6.3 Generazione della documentazione

Il sistema fornisce anche uno strumento per la generazione di una documentazione leggibile per le persone. La documentazione è generata attraverso la trasformazione XSL del file dell'ontologia. È stata aggiunta una nuova pagina dove è possibile richiedere la generazione completa della documentazione completa per tutti gli Urban Dataset. Navigando sui singoli Urban Dataset, è possibile ottenere la documentazione specifica per quell'Urban Dataset. Tale documentazione viene automaticamente aperta come pagina web in una nuova scheda del browser. Nel caso della documentazione completa, è possibile anche scegliere di scaricare la pagina web come file separato su disco locale.

7 Conclusioni

In questo documento si è descritto il lavoro svolto per la definizione, la realizzazione ed il miglioramento 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 durante questo periodo di estensione, sfruttando quanto sviluppato precedentemente, è continuato rivisitando e migliorando alcune applicazioni sviluppate in precedenza per la generazione e trasformazioni di documenti contenenti informazioni relative agli Urban Dataset. Tali applicazioni integrano classi generate automaticamente a partire dagli schemi XSD e JSONSchema per facilitare lo sviluppo ed aumentarne la flessibilità in caso di modifiche degli schemi future.

È stata sviluppata una nuova applicazione che si occupa di validare i documenti verso gli schemi XSD, JSONSchema e Schematron e queste applicazioni sono state integrate nella nuova versione del servizio web in modo da garantire una più semplice fruizione.

Al server web è stato anche aggiunto un servizio di generazione della documentazione relativa agli Urban Dataset definiti nell'ontologia.

Grazie a queste applicazioni sarà possibile sfruttare meglio 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.

8 Riferimenti bibliografici

1. Jaxb. Disponibile all’indirizzo: https://docs.oracle.com/javase/8/docs/technotes/guides/xml/jaxb/index.html
2. Jsonschema2pojo. Disponibile all’indirizzo: https://github.com/joelittlejohn/jsonschema2pojo
3. Jackson. Disponibile all’indirizzo: https://github.com/FasterXML/jackson
4. JSON Schema Validator. Disponibile all’indirizzo: https://github.com/everit-org/json-schema#json-schema-validator-
5. Saxon. Disponibile all’indirizzo: https://www.saxonica.com/documentation/documentation.xml

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).