Vrije Universiteit Brussel

FACULTEIT VAN DE WETENSCHAPPEN
Vakgroep Computerwetenschappen
Web & Information Systems Engineering

Mobile, client-side context-provisioning via the integrated querying of online semantic web data

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

William Van Woensel

Academiejaar: 2013-2014

Promoters: Prof. Dr. Olga De Troyer, Prof. Dr. Sven Casteleyn

PhD Jury

- Prof. Dr. Olga De Troyer (*VUB - DINF*)
  (http://wise.vub.ac.be/olga-de-troyer)

- Prof. Dr. Sven Casteleyn (*VUB - DINF*)
  (http://wise.vub.ac.be/casteleyn)

- Prof. Dr. Beat Signer (*VUB - DINF*)
  (http://wise.vub.ac.be/beat-signer)

- Prof. Dr. Wolfgang De Meuter (*VUB - DINF*)
  (http://soft.vub.ac.be/soft/members/wolfgangdemeuter)

- Prof. Dr. Ir. Kris Steenhaut (*VUB - ETRO*)
  (http://www.etro.vub.ac.be/ETRO_Team/Personal_Page.asp?PM_ID=700)

- Prof. Dr. Oscar Díaz (*University of the Basque Country, Spain*)
  (http://www.onekin.org/content/oscar-diaz)

- Prof. Dr. Peter Dolog (*Aalborg University, Denmark*)
  (http://people.cs.aau.dk/~dolog/)

Abstract

Mobile devices capabilities have increased tremendously over the last few years, enabling mobile users to run resource-heavy applications, such as route planners, web browsers and games, at any time and any place. Much work has been done to facilitate interaction with mobile applications, in mobile settings and on devices with relatively small screens and cumbersome input features. For instance, multi-touch, gesture and voice recognition are deployed to easily execute certain actions, while obtrusiveness of mobile interactions is adapted to suit the user's current situation (e.g., in a meeting). However, mobile interactions still have inherent limitations, mainly due to the fact they are mobile in the first place; mobile users often do not have the time, or the comfortable desktop-setting, to interact with mobile applications. In order to cope with such limitations, the mobile user's context can further be leveraged. Context is defined as any piece of information related to application interaction, including information on the user's surroundings as well as the user and device. By automatically presenting information and services suiting the user's current context, mobile interactions can be enhanced. For example, considering the mobile user's current surroundings and preferences, he can be notified of nearby shops selling items on his shopping list, or nearby public transportation stations leading back to his hotel.

In this dissertation, we present a client-side framework to provision context in mobile settings, which exploits recent evolutions in mobile device technology and the World Wide Web. By leveraging increased mobile processing power and memory capacity, computationally intensive tasks, such as context interpretation, integration and dissemination, are performed locally on the mobile device itself. Furthermore, the machine-readable Semantic Web, the next step in the evolution of the Web, is utilized as an online platform for retrieving context data. Much useful context information, describing people, places and things in the user's vicinity, is already captured in small, machine-readable online web sources; including websites (e.g., shops, monuments) and online RDF files (e.g., person profiles). As websites are being increasingly semantically annotated, making the meaning of their content explicit, many of them have become fully-fledged semantic data sources as well. In order to achieve transparent, integrated query access to such small online semantic sources, this dissertation further presents a mobile, client-side query service. This query service comprises indexing and caching components to enable querying such an online dataset on mobile devices.

Samenvatting

Dankzij de enorme toename in capaciteiten van mobiele toestellen in de laatste jaren, is het mogelijk geworden voor mobiele gebruikers om computationele en/of data intensieve applicaties zoals route planners, web browsers en spelletjes uit te voeren, op eender welke plaats en moment. Er is reeds veel werk verricht om interacties met mobiele applicaties te faciliteren in mobiele omgevingen en op toestellen met relatief kleine schermen en moeilijk te hanteren invoermogelijkheden. Bijvoorbeeld, technieken zoals multi-touch, gebaar- en stemherkenning worden toegepast om bepaalde acties op een eenvoudige manier uit te voeren, terwijl de opdringerigheid van mobiele interacties aangepast kan worden naargelang de situatie van de gebruiker (bijv., in een vergadering). Ondanks deze inspanningen hebben mobiele toestellen echter nog steeds inherente limitaties, voornamelijk omwille van het feit dat ze in de eerste plaats mobiel zijn; mobiele gebruikers hebben vaak niet de tijd, of de comfortabele bureau-omgeving, om te interageren met mobiele applicaties. Om met zulke limitaties om te gaan kan de context van de mobiele gebruiker benut worden. Context wordt gedefinieerd als eender welk stuk informatie gerelateerd aan de interactie met de applicatie, inclusief informatie over de omgeving van de gebruiker, de gebruiker en het toestel zelf. Door automatisch informatie en diensten aan te bieden gerelateerd aan de huidige gebruiker's context, kunnen mobiele interacties verder gefaciliteerd worden. Bijvoorbeeld, wanneer rekening gehouden wordt met de huidige omgeving en voorkeuren van de mobiele gebruiker, kan hij op de hoogte gebracht worden van nabije winkels die producten verkopen op zijn boodschappenlijst, of nabije openbaar vervoerstations die ritten aanbieden terug naar zijn hotel.

In deze dissertatie stellen we een client-side raamwerk voor om context aan te leveren aan mobiele applicaties, waarbij recente evoluties in mobiele technologie en het Web benut worden. Dankzij de toename in mobiele verwerkingskracht en geheugencapaciteit, kunnen computationeel intensieve taken, zoals de interpretatie, integratie en disseminatie van context, lokaal op het mobiele toestel uitgevoerd worden. Voorts wordt het machine-leesbare Semantic Web, de volgende stap in de evolutie van het Web, gebruikt als online platform voor het verkrijgen van context data. Veel nuttige context informatie, die personen, plaatsen en dingen in de omgeving van de gebruiker beschrijven, is namelijk reeds beschikbaar in de vorm van kleine, machine-leesbare online web bronnen, inclusief websites (bijv., van winkels, monumenten) en online RDF bestanden (bijv., persoon profielen). Aangezien websites meer en meer semantisch geannoteerd worden, waarbij de betekenis van de inhoud expliciet gemaakt wordt, zijn veel van deze websites geëvolueerd tot volwaardige semantische data bronnen. Om transparente en geïntegreerde query toegang tot zulke kleine online semantische bronnen te verkrijgen, stelt deze dissertatie verder een mobiele, client-side query dienst voor. Deze query dienst omvat indexering en caching componenten om het queryen van zulke online datasets mogelijk te maken op mobiele toestellen.

Acknowledgement

First of all, a lot of gratitude goes to my father and grandmother, **Paul Van Woensel** and **Sonia Bruyndoncx**, for their motivation and aid throughout my studies, from highschool to university, through the bachelor, master, and PhD level. Without their excellent upbringing and backing, I would have not had the courage and motivation to make it this far. They gave me the opportunity to study and kept on supporting me, even when the first year at the university turned out not so great. In the same vein, I would like to thank all my family members for their invaluable support, including my brother **Vincent Van Woensel** and aunt **Chris Van Woensel** for their encouragement, as well as my cousin **Peter Vrancx** for our deep and colourful reflections on academic life.

With regards to this PhD, my greatest gratitude goes to Prof. Dr. **Sven Casteleyn**. Without his much-needed guidance and advice, as well as an incessant (and perhaps not always deserved) belief in my capabilities, I would have not made it past the first couple of years. He put great effort in councelling my PhD, including fundamental research work and especially in authoring research papers (which involved re-routing parts of my brain!). Thanks to his Spanish connections, I was able to conduct a research visit to the Politechnical University of Valencia, which provided me with worthwhile experience and research connections. Importantly, he was always ready to help, and *one could have not hoped for a better councellor and promoter*. My gratitude is also extended towards my other promoter, Prof. Dr. **Olga De Troyer**. She created an environment in which I could fully pursue my research interests, and gave valuable comments, insights and guidance during my PhD work. In the end, she was always there for help on any aspect of my PhD.

I would like to thank the members of my jury, including **Beat Signer**, **Wolfgang De Meuter**, **Kris Steenhaut**, **Peter Dolog** and **Oscar Díaz**, for providing me with useful comments and pointers on how to improve the quality of the work.

Special thanks go to **Estefanía Serral** and **Miriam Gil** from the Politechnical University of Valencia, for cooperating with me during the four months of my research visit. Thanks to their collaboration, I was able to further validate my PhD work and extend my research into new domains. I would also like to thank Prof. Dr. **Vicente Pelechano** for accepting the research stay at the PrOS research centre.

I would like to extend my gratitude towards all members of the **WISE research group**, for all the productive discussions as well as enjoyable chats, and especially for sitting through all of my trial presentations and giving useful feedback.

Finally, I would like to thank all of the people who I unwittingly forgot to mention here. Many people contributed in some way to my PhD, and I would like to thank all of them.

Contents

List of Figures

List of tables

List of codes

List of acronyms

ADERIS	Adaptive Distributed Endpoint RDF Integration System
AH	Adaptive Hypermedia
AJAX	Asynchronous Javascript And XML
API	Application Programming Interface
BTC	Billion Triples Challenge
CC/PP	Composite Capabilities/Preferences Profiles
CDT	Context Dimension Tree
CIS	Contextual Information Service
CLDC	Connected Limited Device Configuration
CoBrA	Context Broker Architecture
COIN	COntext-aware INjection
Context-ADDICT	Context-Aware Data Design, Integration, Customization and Tailoring
CSS	Cascading Style Sheets
CWA	Closed World Assumption
DARQ	Distributed ARQ
DB	Data Base
DCMI	Dublin Core Metadata Initiative
DOM	Document Object Model
DSL	Domain-Specific Language
FAR	Furthest-Away-Replacement
FOAF	Friend Of A Friend
GPS	Global Positioning System
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HyCon	HyperContext
ID	Identification

i-MoCo	i-MobileConference
IR	InfraRed
ISBN	International Standard Book Number
Java ME	Java platform, Micro Edition
JDK	Java Development Kit
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JSP	JavaServer Pages
LFU	Least-Frequently-Used
LGD	LinkedGeoData
LPS	Least-Popular-Sources
LRU	Least-Recently-Used
MANET	Mobile Ad-hoc NETwork
MIDP	Mobile Information Device Profile
NFC	Near Field Communication
OOHDM	Object Oriented Hypermedia Design Method
OS	Operating System
OWA	Open World Assumption
OWL	Web Ontology Language
P2P	Peer-to-Peer
PC	Personal Computer
PSSUQ	Post-Study System Usability Questionnaire
QR	Quick Response
RAM	Random Access Memory
RDF(S)	Resource Description Framework (Schema)
RDFa	Resource Description Framework in Attributes
REST	REpresentational State Transfer
RFID	Radio Frequency IDentification
RSSI	Received Signal Strength Indication
R-Tree	Rectangle-Tree
SAWSDL	Semantic Annotations for WSDL and XML Schema

SCOUT	Semantics-based COntext-aware Ubiquitous scouT
SemWIQ	Semantic Web Integrator and Query Engine
SIM	Source Index Model
SOA4ALL	Service Oriented Architectures for All
SOAP	Simple Object Access Protocol
SOCAM	Service-Oriented Context-Aware Middleware
SOFIA	Smart Objects For Interactive Applications
SPARQL	SPARQL Protocol And RDF Query Language
SQL	Structured Query Language
UI	User Interface
UML	Unified Modeling Language
UPC	Universal Product Code
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WebML	Web Modeling Language
WKT	Well-Known Text
(W)LAN	(Wireless) Local Area Network
WSDL	Web Service Description Language
WSMO	Web Service Modeling Ontology
WWW	World Wide Web
(X)HTML	(Extensible) HyperText Markup Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language Transformations
YARS	Yet Another RDF Store

Chapter 1

Introduction

1.1 Research context

The capabilities of mobile devices continue to increase with leaps and bounds. Advances in mobile hardware include increased processing power, memory capabilities and battery life, accompanied by improved screen quality and size. Devices are also outfitted with various sensing capabilities, including GPS, cameras and Radio Frequency IDentification (RFID) readers, allowing interaction with the environment and capturing mobile context. In addition, mobile wireless connectivity has approached broadband speeds and is now virtually ubiquitous, with the widespread deployment of WiFi and 3G/4G networks. Due to all these factors, mobile devices have become extremely popular consumer products, and are being used anywhere and anytime to perform any task. Recent studies show that more mobile devices are currently being sold than PCs[1], and revenues from mobile internet usage are estimated to overtake fixed broadband revenues by 2014[2].

In order to aid mobile users in performing their tasks, a number of research domains focus on tailoring device interactions to mobile users' needs. Since user input is still relatively cumbersome, techniques such as multi-touch, gesture and voice recognition are applied to facilitate executing actions [1, 2]. Accessibility support is provided to meet the needs of certain groups of users, such as elderly and disabled people [3, 4]. In addition, mechanisms are available to reduce mobile interaction obtrusiveness to suit the user's current situation (e.g., in a meeting) [5, 6]. This research in mobile interaction tailoring is being increasingly applied to the commercial devices. For instance, the Samsung Galaxy S3 model[3] supports a range of multi-touch interactions to perform much-used

[1] http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011, Smart phones overtake client PCs in 2011**,** Canalys, February 2012 (access date: 05/06/2013)
[2] http://www.pwc.com/gx/en/global-entertainment-media-outlook/segment-insights/internet-access.jhtml, PwC Global Entertainment & Media Outlook 2013-2017 (access date: 05/06/2013)
[3] http://www.samsung.com/global/galaxys3/ (access date: 05/06/2013)

actions; gestures such as putting the device face-down to quickly ignore obtrusive calls; and accessibility support, ranging from larger font sizes to audible feedback for vision-impaired users.

Despite these efforts, inherent limitations still remain. At any time and any place, mobile users often do not have sufficient time, or the comfortable setting, to comprehensively interact with their mobile device (e.g., at a bar, waiting for the train). To further facilitate mobile interactions, the mobile user's context can be leveraged. In context-aware computing, context is defined as any piece of information related to the human–computer interaction [7]. By utilizing context information, applications can automatically enhance interaction with the user, for instance by presenting useful information and services, or fine-tuning service behavior. In mobile settings, context information useful for enhancing mobile interactions (including current location and surroundings) continuously becomes available. Furthermore, the utility of mobile context-awareness is illustrated by other early examples of context-aware systems, such as mobile tourist guide systems [8, 9]. The particular context-aware settings that we envision can be described as follows.

In mobile, context-aware scenarios, thousands of physical entities (i.e., people, places and things) are encountered by the mobile user as he is walking around. These entities each may have associated descriptions (e.g., personal profile, monument history), which collectively describe the user's surroundings (or environment) and represent its environment context. By supplying this kind of environment context to mobile applications, interaction can be enhanced. For instance, mobile users can be automatically notified of useful entities, such as interesting people at a conference, or shops selling products on his shopping list. By investigating the mobile user's environment, his current situation can be inferred (e.g., at meeting), allowing the obtrusiveness of mobile interactions to be adapted accordingly (e.g., reducing auditory to visual signals). During web browsing, page elements related to the user's current environment, such as products sold by nearby shops or nearby restaurants during dinner time, can be highlighted. Currently, an excellent opportunity exists to exploit novel mobile sensing capabilities (e.g., RFID/NFC, GPS, camera, Bluetooth) to accurately capture the user's coordinates (e.g., via GPS) and unique object identifiers (e.g., via RFID/NFC) to enable the identification of the environment context.

Mobile context-acquisitioning systems encapsulate all the tasks required to realize such scenarios, including the capturing of context (e.g., current GPS coordinates), interpretation (e.g., inferring nearness to physical entities), integration (e.g., combining context from multiple sources) and dissemination (i.e., supplying the collected context to mobile applications). In doing so, these systems relieve mobile applications from the complexity of these tasks. For most existing systems, the underlying design rationale is the need for "thin" clients; resulting in the outsourcing of computationally intensive tasks to an external infrastructure, thus relieving the mobile devices of these tasks [10, 11]. This architecture design is called *centralized*, since all mobile clients communicate with an external, centralized entity (e.g., deployed on external servers or cloud service), which performs most or all context-acquisitioning tasks. To deal with issues resulting from such centralized architectures, including infrastructure requirements, server disconnections, privacy and context dynamicity, some approaches focus on delegating tasks to the mobile device [12, 13].

Web technology offers excellent advantages to mobile computing [14]. Due to the ubiquity of WiFi and 3G/4G, combined with the global URL addressing scheme, useful web resources can be transparently accessed anywhere and at any time. As virtually any mobile platform supports the open web standards and protocols, no additional software needs to be installed. The next step in the evolution of the web, the Semantic Web, presents even more benefits. In the Semantic Web, explicit semantics are assigned to web resources, making them machine-readable and thus directly consumable by software clients. In this vein, Semantic Web technology can be very useful in achieving mobile context-awareness. By leveraging the Semantic Web as an online information platform, relevant web resources can be directly re-used as machine-readable context information [12]. For example, this includes online semantic data on nearby people, places and things (i.e., physical entities). Semantic Web technology is also excellently suited to represent context information in general [11, 15]. As the meaning of the sensed context is made explicit, higher-level context data can be inferred by utilizing built-in reasoning mechanisms. By relying on domain-specific ontologies, context requests can be formulated independent from system-specific APIs [16]. Such ontologies, in combination with unique resource URIs, also facilitate data integration from heterogeneous context sources [12]. Other mobile computing domains also recognize the advantages of Semantic Web technology, such as mobile augmented reality [17, 18] and social networking [19].

The Semantic Web has grown tremendously over the last ten years, meeting the demand for freely available, machine-readable online data. Large data sources are being put online in semantic format, such as DBPedia[4] and LinkedGeoData[5], which collectively contain over a billion RDF statements. An increasing amount of websites are being enhanced with semantic annotations (e.g., RDFa [20], microdata [21]); due to their explicit content semantics, such websites can also be considered part of the machine-readable Semantic Web. The Web Data Commons initiative found that close to 13% of crawled webpages contain semantic annotations[6]. Online RDF files, for instance containing shop catalogues (e.g., via DCMI [22]) and personal profiles (e.g., via FOAF [23]), also form a large part of the Semantic Web. Sindice[7], a Semantic Web search engine, currently indexes ca. 708 million online Semantic Web documents.

1.2 Problem statement

Mobile context-provisioning frameworks encapsulate all required tasks to realize context-aware scenarios (such as illustrated in the previous section). These software frameworks perform a variety

[4] http://dbpedia.org (access date: 28/04/2013)
[5] http://linkedgeodata.org/ (access date: 13/06/2013)
[6] http://webdatacommons.org/2012-02/stats/stats.html (access date: 28/02/2013)
[7] http://sindice.com/ (access date: 28/02/2013)

of context-acquisitioning tasks, such as interpreting, integrating and disseminating context data. Two important observations can be made about these systems. In order to relieve the client-side from these computationally intensive processes, many systems have a centralized architecture, whereby mobile clients communicate with an external, centralized infrastructure performing all relevant tasks (e.g., a server). Secondly, in order to retrieve dynamic (e.g., sensor readings) and static (e.g., place descriptions) context data, these systems typically rely on distributed context provider components. Below, we elaborate on problems resulting from these observed properties.

Typically, so-called context provider components are responsible for supplying both dynamic and static context data to context-provisioning systems, which then perform interpreting and disseminating tasks. Context providers supplying static context data usually rely on a database [10, 24] or RDF repository [16] to store the data. In other cases, all static context data is stored in a central information system, which is directly accessible by the context-provisioning system [25–27]. In order to ensure accurate and up-to-date information, these kinds of approaches require content providers for data entry, for instance place owners or touristic services [25]. It can however be observed that much information on places and things is already available on the Web, typically in the form of websites. As mentioned, an increasing number of websites are being semantically annotated (e.g., using RDFa), making them a source of machine-readable context data. Other online Semantic Web sources, such as RDF files and large online datasets (e.g., DBPedia, LinkedGeoData), may also capture information on places and things. Consequently, these context provider approaches usually result in data duplication. Content providers will often find themselves updating two sources of information; their own online web presence (e.g., website, RDF data) and the centralized information system or particular context provider. Compounding matters, these systems or components typically represent closed data silos, where only proprietary software can access the information. This presents a larger participation threshold for content providers, compared to putting online information on freely accessible web servers.

Many systems delegate computationally intensive work towards an external infrastructure, including context interpretation, integration and provisioning tasks. Such architecture designs result in “thin” client applications, with only minimal processing, memory and storage footprints, which nevertheless have access to powerful context capturing functionality. However, this outsourcing also has its drawbacks. Infrastructure costs are incurred, especially when ensuring scalability and robustness via a redundant set of servers. Even when the offloaded tasks are outsourced to a third party such as cloud services, subscription costs will be charged. Furthermore, dynamic context sensed by the mobile device needs to be uploaded to the server, raising serious privacy concerns in some cases (e.g., location updates) [13]. In case connection to the server is lost, which may occur frequently in mobile settings, not even partial context-aware functionality can be offered [12]. As mobile device capabilities have increased greatly in the last years, the arguments for lightweight clients are rapidly evaporating. For instance, recent high-end mobile devices boast 1,5-2 Ghz quad-core processors and 2Gb RAM; configurations that were only introduced in mainstream high-end laptops and desktop computers a couple of years ago. By exploiting the computational power of mobile devices, the aforementioned problems with infrastructure costs, privacy, context dynamicity, and connection loss can be significantly reduced.

In conclusion, traditional solutions to context provisioning, where separate proprietary systems keep static context information, and computational work is outsourced to external infrastructures, are becoming less and less relevant. Machine-readable online semantic data, associated with people, places and things, can be re-used as location-related context data, removing the need for closed information silos. Moreover, by leveraging current mobile device capabilities and moving context-provisioning tasks to the client side, problems related with centralized architectures can be avoided.

1.3 Goal

The aim of this dissertation is to investigate the possibilities of client-side context provisioning in a mobile setting, whereby existing online semantic data is used as context information. An essential part of the research will consist of achieving transparent and integrated query access to such semantic data, originating from different online semantic sources. In particular, we focus on small online RDF sources put online by the various content providers (e.g., place owners), such as online RDF files and semantically annotated websites (e.g., via RDFa).

Important goals include:

- Performing relevant computational tasks, including context interpretation, integration and provisioning, as well as context sensing, on the mobile device;
- Interpreting raw context data (e.g., GPS readings) and provide it as high-level spatial information, indicating proximity or containment between the user and physical entities;
- Leveraging the Semantic Web as an information platform, allowing online semantic data to be directly used as context information;
- Supplying transparent and integrated query access to online semantic data, originating from multiple small online semantic sources (e.g., RDF files, websites);
- Providing expressive and integrated query access to the collected context information, both in a pull- and push-based way.

1.4 Approach

This dissertation presents a **mobile, client-side context-provisioning framework** called SCOUT (Semantics-based COntext-aware Ubiquitous scouT). SCOUT focuses on providing context information describing the user’s environment; specifically, descriptive information on physical entities (i.e., people, places and things) in the user’s vicinity. SCOUT collects, integrates and provisions this context information in a high-level way, via pull- and push-based SPARQL query access. As a client-side approach, SCOUT does not require an external infrastructure to outsource computational tasks or for centralized data storage. Instead, context sensing, interpretation and provisioning tasks are performed locally. This context-provisioning approach is called *non-centralized*, as no external, centralized entity is responsible for performing context-acquisitioning tasks. Instead, the mobile client devices themselves perform all relevant tasks locally.

At the same time, the Semantic Web is leveraged as an information system to obtain the required context information, namely descriptive information on nearby physical entities. In particular, we focus on small online semantic sources associated with physical entities (e.g., FOAF profiles, semantically annotated websites describing places and things). This presents content providers (e.g., shop owners, tourist services) with the opportunity to keep a single, machine-readable online information source up-to-date, utilizable by multiple approaches. Figure 1-1 illustrates this situation, where physical entities (e.g., people, monuments, restaurants) are described by associated online semantic sources. We detail other parts of the figure in the paragraphs below.

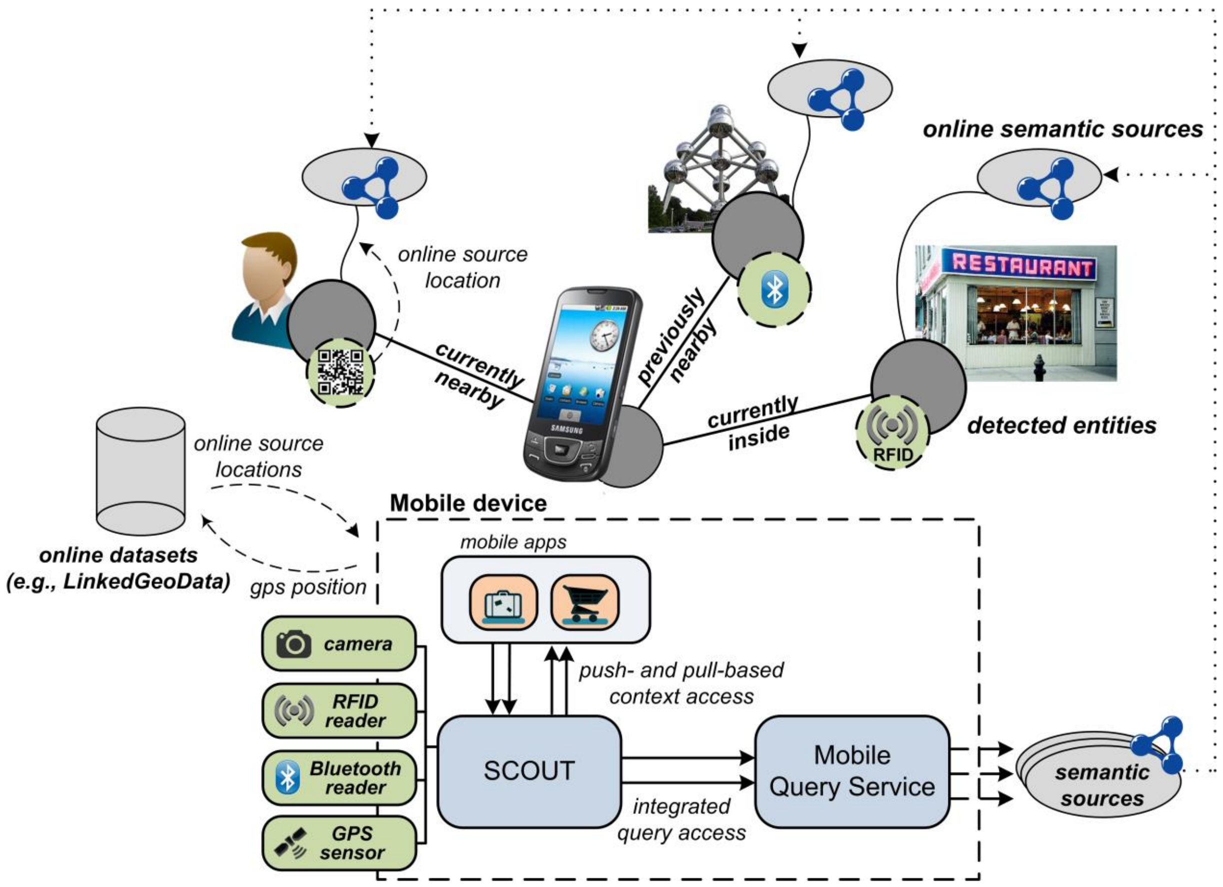


Figure 1-1. General approach overview.

The SCOUT framework features a layered architecture, reflecting its bottom-up approach to context acquisition. The bottom layer, the Detection Layer, is responsible for discovering location-related, online semantic data. For this purpose, SCOUT relies on various interchangeable detection techniques. A first set of techniques rely on tagging technologies such as RFID and QR codes. This is exemplified in Figure 1-1, where physical entities are tagged with QR codes, RFID tags and Bluetooth beacons, and the mobile device is outfitted with sensing technologies such as a camera, RFID reader and Bluetooth component. By utilizing these technologies, physical entities can be detected (i.e., by detecting their tags) and the associated online semantic sources can be located (e.g., by decoding the QR code or reading the RFID tag). Other detection techniques contact online semantic datasets such as LinkedGeoData, which, given the user's current GPS position, return summary information on nearby physical entities, including locations of associated online semantic sources.

The second layer, the Spatial Layer, is responsible for inferring high-level spatial information, based on sensing information from the Detection Layer (e.g., GPS coordinates, RFID detection distance). In order to represent spatial information in a high-level way, so-called spatial relations are kept between RDF resources representing the user and related physical entities, either denoting nearness or containment. Figure 1-1 illustrates such relations, where the mobile user was previously nearby the Atomium, and currently inside a restaurant and nearby another person (e.g., having lunch together). Certain low-level detection info (e.g., GPS readings) is also stored for each spatial relation, which may be useful for certain applications (e.g., for showing a map view). Mechanisms are in place to keep these spatial relations up-to-date, invalidating them in case the entities in question are no longer spatially related (i.e., nearby or contained).

Finally, the Environment Layer represents the most challenging layer. Based on the spatial relations and discovered online semantic data, it constructs and maintains an abstract Environment Model. Mobile applications may pose push- and pull-based semantic queries to this model, in order to reference any part of the user's current and past physical surroundings. For instance, Figure 1-1 shows a tourist and shopping mobile app relying on SCOUT to gain push- and pull-based query access to the user's collected context, respectively to find nearby points-of-interest and shops selling useful products. All context information, both internal data and data provisioned to applications, is represented using Semantic Web technology.

An important part of the Environment Model comprises the detected online semantic data, describing the user's current and past surroundings. Clearly, as the mobile user is moving around, this online dataset will grow to contain large amounts of data (i.e., more than 100Mb). Two observations can be made:

- Queries cannot simply be executed on the entire dataset, due to limitations in memory and processing capabilities. Instead, queries should be performed only on the part of the dataset containing query-relevant information.
- Downloading the entire online dataset for each individual query would lead to unacceptable performance. On the other hand, the dataset should also not be stored entirely on the mobile device, to minimize the storage footprint and cope with devices with limited storage.

Therefore, a straightforward solution, whereby all collected context is stored on the device and all queries are resolved on the entire dataset, is not an option.

For this purpose, this dissertation presents a **mobile query service**. This general-purpose query service enables the transparent and integrated querying of large amounts of online semantic data, originating from relatively small sources (e.g., RDF files, semantically annotated websites). As was the case for SCOUT, this integrated querying approach is deployed on the client-side. As illustrated in Figure 1-1, SCOUT relies on this query service to gain integrated query access to online data describing the user's surroundings. In doing so, SCOUT acts as a *client* of the mobile query service. Note that, due to the general-purpose nature of the query service, any system can act as client to gain integrated query access to an online semantic dataset.

In response to the aforementioned observations, the query service comprises two key components:

1/ a source identification component, to identify query-relevant parts of the online dataset;

2/ a cache component, which locally stores data likely to be re-used in the future.

In order to optimize data access, while still requiring only minimal processing and memory, the query service focuses on exploiting the semantics of RDF(S)/OWL data. More specifically, the source identification component indexes source metadata combinations (i.e., predicates, types) to perform the identification task; and one of the developed cache components organizes the cached data via shared source metadata. In fact, multiple variants of each component were developed keeping varying amounts of source metadata (or none at all), in order to investigate the utility of source metadata in optimizing data access (see also section 1.4.1 on experimental validation). In order to locally query the identified / cached semantic source data, the query service relies on an existing mobile query engine (e.g., Androjena[8], RDF On The Go [28]).

Moreover, the components are fine-tuned to our setting where data is captured in online files. In particular, we present a novel cache removal strategy called Least-Popular-Sources (LPS). In this strategy, cached source data is removed on a per-source level, independent of the actual cache organization. To determine candidates for removal, certain data properties and relations of source data are taken into account, as well as the time to re-download the source. Finally, the query service provides configurable support for the Semantic Web's Open World Assumption.

The SCOUT framework and mobile query service are implemented for the mobile Android OS, and currently support Android version 4.1.2 (Jelly Bean) with API level 16.

1.4.1 Validation

First, we present three mobile client-side applications, which rely on the SCOUT pull- and push-based context access to realize their own functionality. These applications, each with their own

[8] http://code.google.com/p/androjena/ (access date: 09/06/2013)

contributions in their respective fields, serve as proof-of-concepts of the SCOUT framework. Below, we shortly summarize these applications:

- **COIN**: COIN (COntext-aware INjection) [29] is a mobile client-side web augmentation approach, which automatically injects context-aware and personalized features into visited webpages. For instance, this includes annotating webpage content relevant to the user's current location, such as products sold by nearby shops. To achieve this adaptation, COIN requires descriptive information on the user's surroundings, as well as push-based access to keep features up-to-date.

- **AdaptIO**: AdaptIO [30] is a mobile system that automatically adapts the obtrusiveness of mobile interactions to suit the user's current situation. For instance, this includes reducing auditory notifications to a screen flash and icon while the user is in a meeting. To determine the user's current situation (e.g., in a meeting), AdaptIO requires descriptive information on the user's physical surroundings (e.g., whether he is in a meeting room), and needs to be kept up-to-date to determine their most recent situation.

- **Person Matcher**: the Person Matcher [31] computes the "compatibility score" between the user and people he encounters, based on overlaps between their FOAF networks. Example overlaps include having a friend in common, or sharing the same alma mater. The Person Matcher requires expressive push-based query access to be alerted whenever people are discovered in the vicinity, and to gain access to their online FOAF profile.

Second, we executed an extensive series of experiments to evaluate the performance of the mobile query service. We extracted a realistic query dataset consisting of 5000 RDF files measuring 526 Mb in size, which was partly obtained from the Billion Triple Challenge 2012 Dataset[9]. In order to evaluate the utility of source metadata (i.e., predicates, types) in optimizing data access, we evaluated different variants of the two components, namely the source identifier and cache.

1.5 Advantages

The outlined approach has a number of advantages compared to the state of the art, which we elaborate below.

- By deploying our solution on the client-side, we gain the following advantages:
  - *No infrastructure costs*: No external infrastructure for data storage or task outsourcing is required. As such, no server infrastructure or cloud service costs are incurred;
  - *Avoid centralization-related issues:* Problems related to centralized context acquisition architectures, such as privacy concerns, dynamicity of context data, and connectivity issues, are avoided.

[9] http://km.aifb.kit.edu/projects/btc-2012/ (access date: 09/06/2013)

- Leveraging the Semantic Web as an information platform yields the following advantages:
    - *Avoid data duplication*: in most cases, content providers (e.g., place owners, tourist services) already have a separate online (semantic) data source describing the place or object they are responsible for (e.g., shop, tourist attraction). In case of semantically annotated websites, which are both human-readable and consumable by software agents such as SCOUT, content providers only need to supply and keep up-to-date a single data source;
    - *Reduce participation threshold*: by letting content providers upload their data to any freely accessible web server, instead of a closed, proprietary system, the threshold to contributing data is reduced.
- Utilizing Semantic Web technology to represent context leads to the benefits below:
    - *Generic context requests:* No system-specific APIs need to be used to obtain context information. Instead, context information requests are formulated using well-known, domain-specific ontologies;
    - *Leverage technology features:* Knowledge sharing is made possible, as well as reasoning and inferring new context data;
    - *Facilitate data integration:* Integrating data from heterogeneous sources is facilitated, due to the use of well-known, domain-specific ontologies (in case multiple ontologies overlap, ontology alignment approaches can be employed) and the URI identification scheme that uniquely identifies each web resource;
    - *Support Open World Assumption:* Any new data source can add information on previously found web resources, due to the Semantic Web's Open World Assumption.
- Our focus on supplying location-related context data, discovered using different sensing technologies (e.g., tag readers, GPS), yields the following advantages:
    - *Support heterogeneous environments:* By utilizing multiple sensing technologies (RFID, QR, GPS) interchangeably and in parallel, physical surroundings not outfitted with tags or lacking positioning support (e.g., not covered by online datasets such as LinkedGeoData) are still supported;
    - *High-level spatial information:* Applications can easily obtain location-related context information while abstracting from low-level details (e.g., GPS readings), by referencing spatial relations indicating proximity or containment between physical entities and the user.
- By supplying push-and pull-based, integrated semantic query support, the following benefits are gained:
    - *Integrated query access:* Mobile applications can pose arbitrarily complex semantic queries, referencing any part of the user's current and past environment;
    - *Reactive to context changes:* Applications can become reactive to changes in the user's context, via the provided push-based query support.

1.6 Contributions

The contributions of this dissertation can be divided into primary and secondary contributions. Primary contributions concern research related to the SCOUT framework and mobile query service. Secondary contributions pertain to research related to the approaches utilizing the SCOUT framework.

1.6.1 Primary contributions

The primary contributions of this dissertation concern both the mobile, context-provisioning SCOUT framework and the mobile query service. Below, we elaborate on both sets of contributions.

Mobile SCOUT framework

- We present a mobile, client-side context-provisioning framework, which runs entirely on the mobile device and requires no external infrastructure. As such, it avoids the issues related with centralized architectures, such as scalability, privacy and connectivity issues.
- A context detection mechanism is proposed utilizing multiple detection techniques (e.g., utilizing RFID or GPS) interchangeably and in parallel, allowing SCOUT to be deployed in a range of heterogeneous environments supplying different context sensing infrastructures (e.g., RFID, QR, Bluetooth tags).
- The Semantic Web is leveraged as an information platform, to obtain descriptive information on the user’s environment. As a result, SCOUT does not rely on external context providers (pre-registered with the system) or proprietary information systems, which is typically the case in the state of the art.
- In SCOUT, spatial data is represented in a high-level way, via spatial relations linking nearby or contained physical entities. We present a mechanism to actively keep these spatial relations up-to-date, based on sensor data of varying accuracy (e.g., GPS coordinates, RFID detection distance).
- SCOUT supplies push-and pull-based, integrated semantic query support to the user’s collected environment context. To the best of our knowledge, no current work exists that supplies such push-based, integrated semantic query access.

Mobile query service

- We present a mobile, client-side query service that allows for the transparent, integrated querying of large amounts of small online sources. Configurable support for the Semantic Web’s Open World Assumption is provided. Importantly, this query service runs entirely on the mobile device, performing the source identification and local caching tasks (described below) at the client side.

- An indexing component is presented, which indexes metadata from online sources and identifies query-relevant data sources. By focusing on schema-level information, this index reduces memory and processing requirements while still allowing for high source selectivity. Typically, related work focuses on indexing data that is more resource-intensive to extract and store, such as instance data (i.e., RDF resources) or resource constraints.
- We demonstrate a caching component that caches source data for later re-use in query resolution. The cached data is organized via shared metadata, which enables fine-grained data retrieval while still reducing performance overhead. To our knowledge, the state of the art does not include works that organize the cache via the inherent semantics of data (i.e., types, predicates).
- A novel cache removal strategy is introduced, fine-tuned to our specific setting where data is captured in small online files. In such a setting, cache misses incur a large retrieval overhead due to source download times. To alleviate this issue, this removal strategy aims to reduce the number of cache misses (i.e., removed cached data referenced by a new query) and their associated source re-downloads.

1.6.2 Secondary contributions

The approaches utilizing the SCOUT framework have made contributions of their own to their respective research domains. Below, we mention these contributions briefly:

- **Person Matcher**: the Person Matcher [31] represents the first proof-of concept regarding the supplied SCOUT functionality. In particular, it stands for a mobile application that automatically finds interesting people in the user's vicinity. Given two FOAF profiles, one belonging to the user and one to a nearby person, it performs a detailed compatibility check. To realize its functionality, the Person Matcher relies on the pro-active data filtering supplied by SCOUT, in the form of push-based query support. More specifically, the application registers to be notified when a person with a FOAF profile becomes nearby. We presented a matching algorithm to perform a detailed crawling to find connections between the two FOAF networks. A connection is found when the first person is linked to the second person, either directly (e.g., via foaf:knows[10]) or indirectly (e.g., by co-authoring a document, as indicated by foaf:made[11]).
- **COIN**: COIN is a mobile, client-side approach to make existing websites context-aware, which is achieved via the on-the-fly injection of context-aware features. The initial approach, together with a prototype implementation, was proposed in [32]. Later on, a generic, conceptual framework for the approach was presented [29], including requirements for this type of approach, applicable adaptation methods, and an elaborated description of the general approach. The main contribution of COIN is that, in contrast to most existing adaptation

[10] http://xmlns.com/foaf/spec/#term_knows (access date: 11/08/2013)
[11] http://xmlns.com/foaf/spec/#term_made (access date: 11/08/2013)

approaches, it does not require a pre-engineering of the website. Instead, COIN automatically adds context-aware features to existing websites on-the-fly, as the user is visiting the page. To achieve this, the approach needed to be deployed at the client side, which was facilitated by relying on SCOUT as a client-side context-provisioning framework.

In fact, the COIN approach was motivated by the increased availability of semantic annotations in websites (e.g., RDFa)[12]. Due to its explicit semantics, context-relevant webpage content can be automatically identified and enhanced with context-aware features. As such, COIN represents an alternative way of exploiting semantically annotated websites. Instead of utilizing them as online semantic data sources (i.e., by extracting RDF data) as SCOUT currently does, COIN enhances the user's browsing experience while visiting such annotated websites.

- **AdaptIO**: AdaptIO [33] is an existing system, which automatically adapts the obtrusiveness of mobile interactions to suit the mobile user's situation. In collaboration with the original authors, the approach was extended to introduce a novel contribution; namely, supporting interaction adaption in dynamically discovered, a priori unknown environments [30]. This contribution was made possible via the SCOUT framework, which provisions environment context across heterogeneous environments with minimal infrastructure. To realize the extension of the approach, major changes needed to be made to the methodology and implementation. Firstly, the user was put in charge of defining his own situations (e.g., at work, in a quiet place) via a mobile interface. This avoided reliance on designer knowledge or machine-learning tools, which would be problematic in previously unknown environments. Secondly, the AdaptIO system was moved from the OSGi[13] platform to the mobile device, ensuring autonomy in environments lacking OSGi middleware.

1.7 Outline

The dissertation is structured as follows.

Chapter 2 provides background and related work. We present related work in mobile context-aware computing and correlate it to the SCOUT context-provisioning framework. Furthermore, we elaborate on the state of the art in related fields concerning data indexing and caching, which are the two main tasks performed by the mobile query service.

Chapter 3 describes the SCOUT mobile context-provisioning framework. An overview of the layered SCOUT architecture is given. Each layer is described in detail, including the Detection Layer, Spatial Layer and Environment Layer. Data structures, processes and supported technologies are elaborated for each layer, as well as the communication interfaces between the different layers.

[12] Alternative, yet less practical ways of obtaining webpage content semantics are also discussed; e.g., via site-specific wrappers.
[13] http://www.osgi.org/ (access date: 09/06/2013)

Chapter 4 presents the mobile query service, which provides transparent, integrated querying support for large amounts of small online RDF sources. The chosen approach is outlined and motivated, while issues, challenges, and requirements are formulated as well. Afterwards, each query service component is discussed in detail. We discuss how the Open World Assumption is supported by the mobile query service, thus allowing for robust data exploration. Throughout the chapter, we indicate the various optimizations required to deploy the query service on Android mobile devices.

Chapter 5 describes the performance experiments conducted to evaluate the query service in mobile querying scenarios. We elaborate on the experimental setup, including the querying scenario and real-life experimental dataset. Each query service component, including different component variants, is extensively evaluated. The chapter further reflects on the experimental results.

Chapter 6 discusses three approaches that leverage SCOUT to obtain expressive push- and pull-based query access to context information. In particular, the chapter summarizes each approach, sketches its main contribution to the respective research domain, and describes how the approach utilizes the SCOUT framework.

Chapter 7 deliberates the research results of this dissertation. We provide a summary of the work presented in the dissertation, accompanied by contributions, limitations and future work.

Chapter 2

Background and related work

The previous chapter introduced this dissertation. We discussed the research context, identified problems in state of the art, and presented our goals and resulting approach. Additionally, we discussed the benefits and delineated the contributions of our work.

In this chapter, the necessary background and state of the art is given to place this dissertation in a broader perspective. This chapter is structured as follows. Section 2.1 starts by presenting background, elaborating on existing classifications of context and context-awareness (section 2.1.1) as well as linking mechanisms between the physical and virtual world (section 2.1.2). Section 2.2 continues with the state of the art. Firstly, we discuss the usage of Web and Semantic Web technology in mobile computing (sections 2.2.1 and 2.2.2). Afterwards, we move on to aspects of related work where this PhD dissertation made significant contributions. In particular, sections 2.2.3 to 0 elaborate on context-capturing and provisioning features exhibited by SCOUT, while sections 2.2.8 and 2.2.9 detail data indexing and caching, two domains concerning the mobile query service. In section 2.3, we compare our work to background and related work. This involves positioning the SCOUT framework with regards to the introduced context classifications (section 2.3.1), and indicating our core contributions to the state of the art (sections 2.3.2 to 2.3.8). Additionally, we present a concrete overview of the extent to which other approaches support the features contributed by this dissertation (section 2.3.9). Finally, section 2.4 summarizes this chapter.

2.1 Background

In this section, we elaborate on background related to the SCOUT context-provisioning framework, including definitions of context and context-awareness (section 2.1.1) and linking mechanisms between the physical and virtual world (section 2.1.2).

2.1.1 Context and context-awareness

An often-referenced definition of context is “Any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the applications themselves” [7]. Context information can further be classified into four primary context types [7], namely location (where), identity (who), time (when) and activity (what; i.e., what the entity is doing). A more recent work [34] represents context in an n-dimensional space, where each dimension represents a context type (or facet), and a vector in this space characterizes a context situation. Five context facets are suggested; network profile, user description/preferences, terminal (device) characteristics, location, and environment (i.e., physical context properties, for instance captured by sensors).

A context-aware application may exploit the user’s context in various ways. The first published definition (to our knowledge) described context-aware applications as software that “adapts according to its location of use, the collection of nearby people and objects, as well as changes to those objects over time” [35]. As this definition does not cover non-adaptive software, for instance displaying context information, this definition was later generalized to: “A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user’s task” [7]. In the same vein, [36] differentiates between context-enabled and context-aware applications, whereby the former display and store context information, and the latter adapt their behavior to user context. When however considering more recent context-aware applications, we observed that the above definition is still not broad enough. For instance, it does not cover applications pro-actively notifying the user of information or services related to the current context [30], which are not necessarily related to his current task (e.g., serendipitously finding nearby shops selling products on the user’s shopping list). Clearly, any context-awareness definition has the risk of being quickly outdated. As such, we propose a very broad definition, namely “Applications are context-aware in case they consume context information, and utilize it in some way to realize their functionality”.

Context-aware applications may exhibit various features. An initial taxonomy defined two orthogonal feature dimensions [35], namely information/services and manual/automatic. In the manual case, applications present information (e.g., list of people) or services (e.g., list of printers) to the user, whereby items related to the user’s context are emphasized (e.g., nearby people and printers). In the automatic case, applications directly retrieve certain information (e.g., the profile of a nearby person) or directly execute a particular service (e.g., print on nearby printer), based on the available context. Later on, an additional “contextual augmentation” feature was introduced [37], associating digital information with the user's context (e.g., leaving virtual stick-e notes to physical places). Over the years, more advanced context-aware features have been applied, such as automatic features based on current user activity (e.g., rejecting phone calls [38]) or automatically pushing location-relevant virtual notes [39, 40]. As such, it can be said that any (sufficiently fine-grained) context-feature taxonomy will again be quickly outdated as new innovative approaches are developed.

Context-aware mechanisms are of high value in mobile computing, as it is desirable that applications react to their current location, time and other environment attributes on mobile devices [41, 42]. In fact, context-awareness is most relevant when the user is mobile, due to the resulting high dynamicity of context data [43]. In line with these observations, a large amount of context-provisioning approaches have been developed for mobile settings. When discussing state of the art, we elaborate on mobile context-provisioning aspects related to the SCOUT framework (sections 2.2.3 to 0).

2.1.2 Linking mechanisms

In order to collect context information describing the user's surroundings, SCOUT discovers links between physical entities and associated online information. In this section, we discuss existing mechanisms to link the physical to the virtual world.

In [44], a classification of linkage mechanisms is presented to link physical entities to virtual counterparts, be it associated digital information or services. These include:

1/ *Tagging*, where an identification tag (or beacon) is attached directly on or nearby the entity and loaded with (links to) associated data (e.g., URL). For instance, technologies such as Radio-Frequency Identification / Near Field Communication (RFID / NFC), Infrared (IR) or Bluetooth can be used.

The CoolTown project [14, 44–47] relies on tagging, in order to link physical entities to online web resources (via URLs stored on tags). For example, this includes linking a printer to a webpage providing troubleshooting info and/or printing services. A more recent commercial initiative was Touchatag[14] from Alcatel Lucent, which linked RFID tags to online application actions. For instance, a physical cube could be outfitted with RFID tags on each side, where each tag can be read by an RFID reader to invoke online multi-media player actions (e.g., play, stop). Quick Response (QR) codes are put on a physical entity, and can be scanned by a camera and decoded to concrete content. In practice, QR codes often resolve to an online location (i.e., URL) supplying more information on the scanned object. Microsoft Tag[15] supplies a commercial suite of tools to easily create tags (Microsoft Tags, QR codes, NFC tags) and manage the associated online content.

[14] http://store.touchatag.com/ (access date: 06/06/2013). This initiative ran until 01/10/2012.
[15] http://tag.microsoft.com/ (access date: 08/06/2013)

2/ *Computer vision*, where image-recognition is used to identify objects and retrieve their associated data.

An early example of the computer vision method is the DigitalDesk project at Xerox EuroPARC [48], which used cameras and image recognition to identify objects on ordinary desks and make the associated electronic documents available. However, this method requires powerful computing resources and has results of varying quality [44]. More recently, machine learning techniques have proven promising in realizing computer vision, in particular for augmented reality [49].

3/ *Positioning*, where an entity's absolute position (e.g., in a certain radius around the user) is exploited to retrieve its associated data. When outdoors, GPS can be used to determine absolute location; when indoors, technologies such as WiFi RSSI [50] and ultrasound techniques [51] can be employed to approximate absolute positions.

An important advantage of this linking mechanism is that it does not require outfitting the physical environment with tags (tagging method), which is a process called physical registration in [14]. Also, it avoids the computational complexity of image-recognition (computer vision method). For this linking method, a service (e.g., database) is typically required to connect the absolute coordinates of static entities in the covered area to associated information and services (e.g., as in [25]). For instance, the DBPedia Mobile app[16] utilizes the freely accessible online semantic DBPedia[17] dataset (which is a Semantic Web version of Wikipedia), in order to find information on places and things in the user's vicinity. As a drawback, it can be noted that the positioning method does not directly support physical entities with dynamic locations (e.g., people, moving art exhibits). Also, it requires a server infrastructure to handle service requests, preferably with multiple redundant servers to avoid bottlenecks and single points-of-failure.

2.2 State of the art

In this section, we discuss relevant state of the art. Firstly, we mention aspects that are related to this PhD dissertation, but where no contributions were made. This includes the usage of (Semantic) Web technology in mobile computing (section 2.2.1 and 2.2.2).

Afterwards, we elaborate on aspects where this dissertation made significant contributions. Regarding the SCOUT framework, we elaborate on approaches integrating the physical and the virtual world (section 2.2.3), as well as approaches exhibiting various related context-acquisitioning features (sections 0 to 0). Concerning the mobile query service, we discuss related data indexing (section 2.2.8) and client-side caching (section 2.2.9) approaches. For an overview of our core contributions with regards to this state of the art, we refer to section 2.3.

[16] http://wiki.dbpedia.org/DBpediaMobile (access date: 08/06/2013)
[17] http://dbpedia.org (access date: 08/06/2013)

2.2.1 Use of Web technology in mobile computing systems

SCOUT relies on proven web technology for the online hosting and retrieval of context-relevant data. In this section, we discuss how web technology is used in mobile computing as communication medium or information platform.

Open web standards and protocols (such as HTTP, XML and SOAP) are supported on virtually any software platform, meaning any mobile device can support these technologies without requiring the installation of middleware [14, 52], such as e.g., CoBrA [11]. Furthermore, global URL addressing scheme that allows transparently addressing web resources (i.e., independent of the hosting servers' physical locations) [14] is an important web-based technique for our context.

To our knowledge, the CoolTown project [14, 44–47] is the first attempt at leveraging web technology to create context-aware applications, whereby the aforementioned advantages are exploited. The CoolTown project focuses on creating and managing web presences for physical entities, providing information and services (e.g., personal profile) related to the physical entity (e.g., user). These web presences are available via an interactive web interface and a remote API that can be accessed via web protocols. By relying on web technology, web presences can be hosted on any web server and accessed via ubiquitous web communication protocols. Moreover, in case the web presences are hosted locally (i.e., in the place itself), a local WLAN connection is sufficient to obtain web presences. The HyCon approach [53] discusses context-aware searching (also called Geo-Based Searching), whereby contextual information is used as an additional search constraint. Using reverse geocoding, the user's GPS position is converted to a postal address and passed on to a web search engine. As such, it exploits the wealth of existing web content that is related to specific locations. Other context-aware approaches also leverage web technology; for instance, the Context Toolkit [36, 43] relies on HTTP for communication between components, and the contextual information service described in [10] uses HTTP to send queries to context sources.

Similar to the CoolTown project [14, 44–47] and the HyCon approach [53], SCOUT leverages online data related to places, objects or things in the user's vicinity. Examples of such data sources are online RDF files, and semantically annotated websites from which RDF data can be extracted. By relying on web technology for hosting and retrieval, the aforementioned advantages are gained; namely, minimal middleware requirements, global addressing scheme, and ubiquitous availability.

2.2.2 Use of Semantic Web technology in context-aware computing

In order to facilitate data integration, as well as allow expressive and rich query access, SCOUT utilizes Semantic Web technology. In this section, we discuss other approaches in the domain of context-aware computing also leveraging Semantic Web technology.

The value of Semantic Web technology for context-aware applications has been well recognized [11, 12, 15, 16, 54–57]. By using Semantic Web technology to represent contextual information, explicit semantics can be assigned to the data, using well-known standards and a wide range of existing, domain-specific ontologies. This enables better knowledge sharing and reasoning in ubiquitous environments [11, 12]. By exploiting the built-in reasoning mechanisms, additional context may be inferred, while ontology alignment mechanisms can be applied to deal with issues such as overlapping ontologies [12, 16]. Importantly, Semantic Web technology allows resolving issues inherent to newly discovered, heterogeneous mobile environments. This includes integrating a priori unknown heterogeneous data sources [12, 15, 16], resolving uncertainty in sensor readings [54], and improving matching between the currently available context and application needs [12, 16]. Furthermore, since Semantic Web technology supports the Open World Assumption, any data source can specify additional information on context entities. For these reasons, other mobile computing domains also rely on Semantic Web technology, such as augmented reality [17, 18] and social networking [19]. These mobile systems locally access and manipulate RDF data via mobile RDF stores such as Androjena[18], RDF On The Go [28], and i-MoCo [58], and systems such as MobiSem [12].

The Context-ADDICT approach [15] provides a unified query view over context sources, and relies on ontologies to better enable the integration of previously unknown data sources and reason over the data. The MobiSem Context Framework [12] provides support for the increasing amount of mobile applications accessing online Semantic Web information, by supplying programmatic access to locally replicated RDF data. By relying on Semantic Web technology, the integration of data from multiple, heterogeneous data sources is facilitated, while also enabling the enrichment of context with additional semantics (e.g., timestamps). The distributed context management framework presented in [16] allows mobile applications to pose SPARQL queries to dynamically discovered context providers. Firstly, by utilizing well-known, domain-specific ontologies, mobile applications can pose context data requests without requiring a priori knowledge on the specific environment, context sources or providers (as opposed to relying on certain APIs). Secondly, by relying on techniques such as subsumption, indirect matches can be made between application needs and the available context, based on ontological knowledge. For instance, in case turned-on devices in the user's vicinity, such as radios and TVs, are annotated with a subtype of DistractionDevice, an application may infer that the user is busy regardless of the specific device. To cope with overlapping ontologies and ontology heterogeneity, an ontology alignment service may be employed [12, 16].

CoBrA-ONT [11] presents a set of OWL ontologies for the Context Broker Architecture (CoBrA) middleware architecture, to better enable knowledge sharing, context reasoning and privacy protection in ubiquitous environments, as opposed to sets of Java objects representing context (e.g., as in the Hydrogen system [59]). The SOCAM system [57] presents an OWL context ontology, which enables reasoning over context data and deriving high-level context, as well as resolving context conflicts. The ONCOR ontology framework [54] aims to detect and resolve uncertainty, noise and conflicting sensor information. Sensor output is represented via an ontology class corresponding to the sensor's location and detection granularity (e.g., for a low-range Bluetooth beacon, Left-Wing-Level-1). Via the part-of relations in the ontology, the system can detect whether some information is

[18] http://code.google.com/p/androjena/ (access date: 09/06/2013)

in conflict (e.g., presence in two different 2nd floor rooms), and subsequently resolve the conflict (e.g., infer presence on the 2nd floor).

In addition to utilizing Semantic Web technology for representing context information, SCOUT leverages the Semantic Web as an online information platform for static context data. During our discussion on state of the art, we discuss other approaches also utilizing the Semantic Web for this purpose (section 2.2.5).

2.2.3 Integrating the physical and the virtual world

While discussing relevant background, we indicated a number of mechanisms to link the physical and the virtual world (section 2.1.2). SCOUT utilizes such links to collect context information (digital) on the user's surroundings (physical). Below, we give an overview of other approaches using these mechanisms to effectively integrate both worlds.

A location-aware platform is presented in [50] that combines multiple linking mechanisms. More specifically, the platform relies on so-called location sources to obtain place identifiers; these can be explicit sources, utilizing the positioning method, or implicit, relying on tagging or computer vision. Location sources are statically prioritized based on their accuracy, allowing less accurate sensing technologies (e.g., WiFi RSSI) to be employed in case more accurate ones (RFID / IR tags) are not available.

The CoolTown [14, 44–47] research project aims to create a tighter link between real-world physical entities and their web representations. Linking is achieved via URLs, which point to online web resources associated with physical entities. These URLs are automatically sensed, either directly or indirectly, from the user's physical surroundings [14]. With direct sensing, a beacon (e.g., infrared, Bluetooth) or tag (e.g., RFID/NFC) put in the vicinity of the physical entity directly provides the URL of the entity's associated web resource. In the case of indirect sensing, a beacon or tag presents an identifier (e.g., ISBN, UPC barcode) that can be converted, via a resolver service, into the corresponding web resource URL. This allows using different resolvers in different places with location-specific resolution results (e.g., resolving an ISBN number in a library / book shop leads to the official book webpage / book purchasing page). A process called physical registration is described in [14], where a place responsible (e.g., librarian) manually links physical entities to their web resources. This process includes printing URL barcodes and placing beacons / tags nearby physical entities, and potentially registering identifiers (e.g., ISBN) with the place-specific resolver. Consequently, this research project relies on the tagging method.

An open lookup infrastructure for tagged products is presented in [60]. The infrastructure comprises a set of resource repositories, where a resource represents a piece of data on a tagged product (ranging from expiration dates to URLs). Useful resources can be found in both manufacturer- and third party-repositories, and the infrastructure likewise relies on resolver services (see CoolTown project) to convert detected tag IDs into particular repository addresses. In analogy to CoolTown, the proXimity system [61] aims to give the Web a physical presence in the world, whereby physical artifacts are augmented by online hypertext and semantic information. The coupling of the Web and physical world is achieved by electronically marking physical artifacts using embedded, non-networked devices containing online locations. Therefore, both the open lookup infrastructure and proXimity system rely on the tagging method. The HyperContext (HyCon) approach [53] discusses context-aware searching, called Geo-Based Search, whereby contextual information (i.e., location) is used as an additional constraint for searching digital information. This Geo-Based Search is applied on the Web, by using reverse geocoding to convert the user's GPS position to a postal address and feeding it to a web search engine. As such, HyCon relies on the positioning method, and utilizes existing online search engines to find location-specific information.

Physical Hypermedia applications [26, 27] are hypermedia systems where nodes represent either digital data or real-world physical objects, while the links between them may also be digital or physical. Consequently, these systems are excellent examples of integrating the physical and the virtual worlds, and obscure the boundaries between both. The work in [26] and [27] represent extensions to the Object Oriented Hypermedia Design Method (OOHDM), a web application design methodology, in order to support physical hypermedia. Digital information about real-world objects is accessed by encoding the location of the object in the request (i.e., as a URL parameter), whereby the given location is either a concrete identifier (e.g., sensed via IR) or a set of absolute coordinates. Different location models are supported; e.g., symbolic models map concrete identifiers to objects, while geometric models map absolute coordinates. As such, both tagging and positioning methods are supported.

2.2.4 Context-acquisition software frameworks

Our approach realizes bottom-up context acquisitioning, where low-level sensor data (e.g., read via GPS, RFID and QR) enables the retrieval of descriptive online semantic information, as well as the creation of high-level spatial relations. To provide access to the collected context information, a push- and pull-based, integrated query interface is provided. Below, we elaborate on other software frameworks that realize this bottom-up process of context sensing, interpretation and provisioning.

The Context Toolkit [36, 43] is one of the first attempts at creating a re-usable library for context-aware applications. To separate context sensing/interpretation from regular application logic, the Context Toolkit provides a library of re-usable and composable widgets (cfr. GUI toolkits). A widget provides one of the primary context types discussed in [7] (i.e., identity, location, time, activity) and provides push- and pull-based access to the context. A widget relies on generator components, which sense raw context data; and interpreter components, which abstract this raw data into high-level information. A server is a special kind of composed widget that provides all primary context types for a certain physical entity [7, 36]. Widgets may be distributed across the network, and publish their capabilities to a centralized discoverer component. Mobile applications contact this discoverer to locate useful widgets. Comparably, [16] presents a distributed context management approach that relies on context providers to publish high-level context information. Mobile applications can address service directories to find suitable providers, or send a broadcast over the network. After discovering useful providers, an application issues SPARQL queries to obtain the desired context data. The Service-Oriented Context-Aware Middleware (SOCAM) [57] also relies on context providers, which provide low-level context data or convert given raw context data to high-level context. Mobile applications utilize a Service Locating Service to locate useful context providers; found providers are contacted via their API in a pull- and push-based way.

In contrast to the distributed approaches above, where most tasks are performed on an external infrastructure, other approaches focus on performing most (or all) context acquisition tasks on the mobile device. The mobile, client-side context-provisioning framework Hydrogen [59] has a layered architecture, enabling the separation of the different context-acquisition concerns. This architecture comprises an Adaptor Layer, containing adapters that retrieve sensor data while hiding hardware-specific details; and a Management Layer, comprising a Context Server that provides pull- and push-based context retrieval. The stated reason for deploying the framework on the device is to reduce the effects of network disconnections. As Hydrogen only supports adapters encapsulating local device sensors, the Context Manager allows peer-to-peer context sharing with other devices to obtain non-sensed context information.

The CoolTown project also comprises a distributed software framework [47] that allows the development, deployment and management of web presences. A web presence is a representation of a physical entity on the Web, and offers various information and services related to the entity. Concretely, a web presence consists of a set of modules, whereby each module delivers 1/ an interactive (form-based) user interface, accessible from browsers, and 2/ module APIs for remote or local programmatic access. Web presences are either deployed on the physical entity itself (e.g., hardware device such as a printer, or on person’s mobile device) or on a server running their web presence software. Mobile applications can access the remote module API of web presences to obtain more information on the entity, or to invoke services. Available modules include the directory, discovery, autobiographer, observer and control modules. The directory module stores and manages relationships, which indicate spatial relations between the physical entity and other entities (i.e., nearness, containment), and also keeps the related entities’ web presence contact info (i.e., a URL). The discovery module encapsulates the sensing and updating of positional relations, using technologies such as RFID and Bluetooth. In addition, this module obtains related entities’ associated web presence contact info, by for instance reading URLs from RFID tags. The autobiographer module allows modules to add events to a log, which is monitored by the observer module. In order to be alerted in case a certain event is logged (e.g., person leaves the room), applications can register rules with the observer module. Finally, control modules allow interacting with the physical entity and execute its services (e.g., printing).

2.2.5 Obtaining static context data

SCOUT leverages the Semantic Web as an information platform, retrieving static context data in the form of online semantic data describing the user’s surroundings. Below, we discuss how other approaches, utilizing context providers or centralized context storage, currently supply and maintain static context data. Also, we elaborate on other approaches that re-use online (Semantic) Web data as static context information.

Distributed context-acquisition frameworks (e.g., [16, 36, 57]; see section 2.2.4) typically rely on distributed context providers to retrieve dynamic (e.g., sensor readings) and static (e.g., place descriptions) context information. In order to store static context data, context providers usually utilize a database [10, 24] or RDF store [16]. In other application scenarios, centralized storage is employed to keep context- (or location-) relevant data. For instance, web application methodologies supplying location-specific information are typically deployed on a centralized server, which stores all data (e.g., Physical Hypermedia approaches [26, 27]). To avoid bottlenecks and single-points-of-failure [41], a server infrastructure (with redundant servers) should be available, making these kinds of systems expensive to deploy on a large (e.g., city-level) scale. In [25], a location-aware web application is presented that supplies location-specific information to users, and likewise stores the data on a central server. Importantly, the approach focuses on facilitating the entering and managing of location-specific data. Content providers can define places by drawing rectangular boxes on a predefined map, and associate (structured) data with newly defined place.

The HyCon [53] system re-uses existing online, location-specific information as static context data, by accessing online search engines. The Context-Aware Data Design, Integration, Customization and Tailoring (Context-ADDICT) approach [15, 62–64] supports the re-use of existing online data, via local wrapper components encapsulating remote data sources. The MobiSem Context Framework [12] locally replicates RDF data from existing online Semantic Web datasets, thus re-using existing online semantic data as static context data. In the case study of the MobiSem system, the Sindice[19] Semantic Web search engine is utilized to retrieve FOAF profiles related to people in the user's calendar, and DBPedia to obtain data related to the user's current location. The Context-ADDICT and MobiSem systems are discussed in more detail in the next section (section 2.2.6). DBPedia Mobile[20] is a location-aware mobile app, which automatically populates a map view with semantic information from the online DBPedia[21] dataset. Starting from the map, users can explore linked information, navigating through the DBPedia dataset as well as interlinked datasets such as GeoNames[22]. mSpace Mobile [65] is a mobile application relying on multiple online semantic sources, providing the user with a faceted interface to access location-specific information as well as explore linked information. Behind the scenes, both apps issue SPARQL queries to the query endpoints of the respective online datasets to obtain the required data.

[19] http://sindice.com/ (access date: 09/06/2013)
[20] http://wiki.dbpedia.org/DBpediaMobile (access date: 09/06/2013)
[21] http://dbpedia.org/ (access date: 09/06/2013)
[22] http://www.geonames.org/ (access date: 09/06/2013)

2.2.6 Contextual information services

Following the definition from [10], we consider a contextual information service to supply a unified interface over distributed context sources. SCOUT suits this definition, as it supplies integrated query access over online semantic sources describing the user's surroundings. Below, we elaborate on the state of the art regarding contextual information services.

The approaches presented in [10, 24] are deployed on a server infrastructure. The systems accept queries posed by mobile applications, distribute them across registered context providers/sources, and integrate the results. Both approaches require context sources to register with the system, and provide varying support for heterogeneous context sources. In particular, the approach presented in [10] requires a global conceptual schema to be created at design time, which covers the data provided by registered context sources. In contrast, the middleware presented in [24] dynamically matches the schemas of new context sources to a set of global schema's, thus providing better support for heterogeneous sources. The constraints put on the context sources also vary. Context sources in [10] need to implement (a subset of) their specific query interface, for instance allowing a query to specify meta-attributes such as the desired accuracy (e.g., location granularity) and timeliness (i.e., data freshness) of the returned data. In [24], the context source is responsible for keeping the results of the schema matching process (which matches local schemas to a set of global schemas), and converts incoming queries to match the local schema.

The Context-ADDICT approach [15, 62–64] also offers an queryable, integrated view over context data. In contrast to the above approaches, it focuses on integrating information from dynamically discovered data sources (e.g., web pages, web services, sensor networks), and does not require a priori registration. In addition, fewer constraints are put on data sources, and responsibilities are instead moved to the system itself. More specifically, for each data source type, the system requires: 1/ an automatic semantic extractor component, which extracts an ontology from the particular source type, and 2/ a wrapper component, used by the query resolver to obtain data from the source type given a query. As a result, context sources may have varying computational capabilities, or none at all. In order to discover new data sources, Context-ADDICT utilizes the Data Source Discovery Service, whereby the discovery criteria vary on the specific scenario. To reduce the amount of information that needs to be queried and cached, an application-specific and pro-active context-aware filtering of sources is performed. At design time, an application designer creates a Context Dimension Tree (CDT), which captures the context dimensions relevant for the specific application (e.g., time, location). At runtime, this CDT is instantiated with the user's current context (e.g., current location, time) and used to filter sources relevant to the application needs and user context.

The MobiSem Context Framework [12] provides a Data Access API, which allows mobile applications to programmatically access Semantic Web data originating from various online sources. To cope with data connection issues, the framework pro-actively replicates Semantic Web data on the user's device for local access. It is observed that 1/ only a small selection of data should be replicated, to deal with mobile device restrictions, while 2/ a mobile user's information needs also continuously change. To tackle these issues, the developed data replication strategy continuously selects subsets of remote data sources for replication, based on the user's (current and future) context. When connectivity allows, the selected data is pro-actively and transparently replicated on the user's device. For instance, by accessing the user's calendar, FOAF profiles of people the user is meeting with can be replicated (future context), as well as data related to his current surroundings (current context). It can be noted that MobiSem retrieves relevant online data and queries it locally, as opposed to distributing posed queries across online sources (such as in [10, 15, 24]).

2.2.7 Exploiting mobile device capabilities

An important characteristic of SCOUT is that it does not rely on an external infrastructure to take care of computational tasks. Instead, it exploits modern mobile device capabilities to perform these tasks locally. In this section, we elaborate on other context-provisioning systems leveraging mobile device capabilities.

Many approaches reference the need for lightweight and "thin" client applications as the rationale for developing centralized architectures (e.g., [10, 11]). In these architectures, data storage and computational tasks, such as context management, data integration and query resolving, occur on an external, centralized infrastructure. We observe that, with the ever increasing capabilities of mobile devices (regarding processing power, memory capabilities), the need for lightweight clients has been significantly reduced. Reflecting this observation, systems are increasingly exploiting device capabilities, to avoid heavy infrastructure requirements [40, 61], efficiently deal with dynamic context [13] and reduce consequences of connection losses [9, 59].

The context-aware tourist GUIDE system [9] aims to provide up-to-date and context-aware hypermedia information, and is an early example of exploiting mobile device capabilities. The GUIDE setup comprises a central server that keeps touristic information, local base stations providing wireless coverage, and a client-side application. To cope with potential connection loss, the mobile client caches the touristic information, and continuously adapts downloaded webpages by inserting cached touristic data corresponding to the user's current context. [13] presents a context-aware multimedia content filtering approach, which filters multimedia content by matching its metadata to the user's context. The filtering process is performed in a distributed way, whereby the client side filters based on dynamic context (e.g., location), and the server side filters based on static context (e.g., user preferences). By performing dynamic context filtering at the client side, privacy issues are avoided (e.g., sending user's location across the network) and bandwidth usage is significantly reduced (e.g., by avoiding frequent location updates to be sent).

The MobiSem Context Framework [12] aims to support mobile applications accessing online Semantic Web data. To cope with data connection issues in mobile scenarios, the framework pro-actively replicates Semantic Web data on the user's device, and provides mobile applications with programmatic access via their Data Access API. Comparably, to cope with data source transiency (i.e., sources becoming unavailable, for instance because of connectivity issues), the Context-ADDICT approach [15] caches context data. While the MobiSem framework runs entirely on the mobile device, Context-ADDICT targets both server-side and local (mobile device) deployment. In order to cope with device limitations and reduce the amount of data to be stored and processed, the MobiSem data replication strategy only replicates small selections of data, based on their relevance to the user's context.

The Hydrogen approach [59], a mobile context-aware framework, focuses on supporting typical mobile scenarios, whereby for instance device limitations and network disconnections may pose problems. To increase robustness with regards to network issues, the architecture includes a local context server deployed on the mobile device, which provisions local context (collected via local sensors and services) to mobile applications in a push- and pull-based way. In order to cope with device limitations, Hydrogen does not store vast amounts of context history.

2.2.8 Data indexing

In order to transparently query a large set of small, online RDF sources on mobile devices, the mobile query service performs source indexing and caching (see Chapter 4). In this section, we elaborate on the state of the art regarding data indexing. In the following section, we discuss related work concerning client-side data caching (section 2.2.9). As mentioned, we refer to section 2.3 for an overview of our core contributions with regards to this state of the art.

Data indexing is utilized in a number of related domains to locate useful data sources and optimize data access; including context-aware systems, query distribution approaches, and RDF stores. Below, we elaborate on indexing applied in these fields.

2.2.8.1 Context-aware systems

Distributed context-acquisition systems (see section 2.2.4) typically index the capabilities of context providers, allowing components to easily locate useful providers. In the Context Toolkit [36, 43], widgets sense, interpret and aggregate context, and publish their capabilities to a centralized discoverer component; for instance, capabilities include supported context types and interpretation mechanisms [43] (the concrete format is not elaborated). Mobile applications contact the discoverer component to locate useful widgets. In the SOCAM system [57], context providers sense or interpret context, and publish their capabilities in the form of OWL expressions to a Service Locating Service. Mobile applications send a query to the Service Locating Service, indicating the desired kind of context. In response, the service applies semantic matching to match the query to the published capabilities. In [16], context providers publish their capabilities as a set of sensed OWL classes and properties. Mobile applications either contact a directory that stores these capabilities, or directly obtain them in response to a broadcast on the local network. By leveraging an ontology alignment service (e.g., [66]), mobile applications can deal with ontology heterogeneity.

In order to automatically distribute queries across query-relevant context sources, contextual information services (see section 2.2.6) should also support indexing the capabilities of context sources. However, [24] requires clients to manually specify query-relevant context providers, while [10] does not go into detail on the indexing and query distribution step. In the MobiSem system [12], RDF data is pro-actively replicated from pre-configured online datasets to the user's device. No indexing is performed to reduce the query dataset; while performance problems are already reported when processing datasets with more than several hundred triples, which is far from acceptable performance. In Context-ADDICT [15, 62–64], the indexing step includes extracting a Data Source Ontology from each data source, which is subsequently integrated into the Domain Ontology. Mobile applications specify queries using the Domain Ontology, allowing the system to dynamically and transparently identify query-relevant sources.

2.2.8.2 RDF Query distribution

RDF query distribution approaches typically rely on indices to identify sources that contain data relevant to posed queries. During query distribution, the posed query is divided into subqueries, distributed across the identified data sources, and the integrated results returned. For instance, these indices may also store additional information to optimize the query distribution plan, as we explain below.

The Distributed ARQ (DARQ) [67] and Semantic Web Integrator and Query Engine (SemWIQ) [68] systems keep an index with summary info on each dataset. DARQ keeps so-called service descriptions, including found predicates, constraints on subjects and objects that occur together with these predicates, and statistical information (e.g., selectivity estimations for triple patterns with certain predicates). The SemWIQ system maintains a catalog per data source, which keeps a list of classes and their number of instances, as well as a list of properties and their number of occurrences. In both cases, the index is kept in RDF format. Given a posed query, these indices are used to determine which semantic query parts (i.e., triple patterns) should be sent to which datasets. This leads to a query execution plan, which is later optimized using existing database techniques. After construction, the optimized query execution plan is executed, the results are joined and finally returned. The Adaptive Distributed Endpoint RDF Integration System (ADERIS) system [69] keeps only very limited summary info, namely contained predicates, and directly sends subqueries to relevant datasets. Instead of relying on pre-calculated statistics (as in DARQ) to optimize query execution plans, it focuses on dynamically re-ordering result joins based on runtime selectivity statistics (obtained from analyzing subquery results). The approach in [70] focuses on indexing occurring path structures (i.e., predicate sequences) into source-index hierarchies. These source-index hierarchies enable the identification of query endpoints that can handle combinations of query triple patterns or "paths", in order to reduce the number of local joins on the individual results.

Since Semantic Web query endpoints are typically autonomous, obtaining accurate and up-to-date statistics can be problematic, due to limited access rights and unexpected content changes. To an extent, the DARQ and SemWIQ approaches take this issue into account by collecting relatively simple summary information. The ADERIS system almost entirely avoids this issue, by mainly utilizing runtime selectivity statistics. By relying on found path structures [70], the impact of content changes is reduced, as it can be assumed that these (schema-based) structures will change less often than instance data.

2.2.8.3 RDF stores

Many RDF stores focus on keeping extensive indices to speed up access to RDF data, trading index space and update efficiency for retrieval time. Androjena[23] is a port of the well-known Jena RDF store[24] to the Android platform. To speed up query access, this store uses three hash tables, respectively indexing the subjects, predicates and objects of kept RDF triples. Given a query triple pattern with concrete values (e.g., subject URI), Androjena uses one of these indices to identify stored triples containing the given value. In case multiple concrete values are given (e.g., subject and predicate URIs), Androjena leverages a predefined ordering of indices to identify the most selective index (e.g., it appeared the predicate index was less selective in practice). Our caching mechanism relies on a similar indexing structure for quick cached data retrieval (see Chapter 4), section 4.7.2.6). More specifically, the cache keeps indices for subject types, predicates and object types. Since these indices keep schema-level information instead of instance data (i.e., resources), they contain significantly less entries.

To fully optimize query access, the Yet Another RDF Store (YARS) approach [71] keeps 6 quad indices to cover all potential quad access patterns. A quad represents an RDF triple accompanied by a context component (e.g., indicating provenance). As an example access pattern, s:p:?:? represents the case where a concrete subject and predicate are specified, and variables are given for the object and context component. Concretely, a quad index is implemented as a B+-tree, and each stored quad is kept as a key in the tree. By leveraging the range query support of B+-trees, they only require 6 quad indices to cover all access patterns. The HexaStore [72] similarly relies on a sextuple indexing scheme to cover each potential triple access pattern. Each index represents an ordering of subject, predicate and object, leading to six indices (= 3!). For instance, a predicate-subject-object index connects each predicate resource to a vector of subject resources, whereby each subject is linked to a list of object resources. A path through such an index represents a stored triple. Therefore, it can be observed that the indexes from HexaStore and YARS actually store the triples as well. Since both approaches require multiple indices, triples are thus stored multiple times, requiring a multitude of the storage space that would normally be needed. In the HexaStore system, it was calculated that the worst-case increase in storage space is five-fold (due to the internal re-use of vectors and lists). It is also noted in [72] that these kinds of indices have high update and insertion costs, since all indices need to be updated. Finally, both approaches perform dictionary encoding, whereby string values (e.g., URIs, literals) are mapped to integer ids. Dictionary encoding enables less data to be stored[25] and optimizes query processing, since integer comparisons are cheaper. RDF On The Go [28] utilizes

[23] http://code.google.com/p/androjena/ (access date: 08/06/2013)
[24] http://jena.apache.org/ (access date: 08/06/2013)
[25] At least, in case resources occur in multiple stored triples, which is typically the case.

YARS to handle RDF data on the mobile Android platform, while i-MobileConference (i-MoCo) [58] relies on the HexaStore for RDF data handling on the mobile iOS platform. RDF On the Go was ported from the Jena store to the Android platform (cfr. Androjena), and has a special focus on supporting spatial data via R-trees.

2.2.9 Client-side data caching

As mentioned, the mobile query service performs source indexing and caching (see Chapter 4) to transparently query a large set of small, online RDF sources on mobile devices. In this section, we elaborate on state of the art regarding client-side caching.

Client-side data caching fully exploits the storage capacity and processing power of client devices, increasing performance and scalability of distributed systems [73, 74]. Below, we elaborate on existing types of client-side caching systems (section 2.2.9.1), and discuss cache replacement policies for mobile scenarios (section 2.2.9.2).

2.2.9.1 Data caching approaches

Most existing caching approaches are based on client-server architectures, where all necessary data can be obtained from the server. Specifically, in traditional data-shipping techniques, clients perform queries locally on data retrieved on-demand from the server. Afterwards, clients cache the obtained data for later re-use [73]. In case a query requests data that was not cached locally, called a cache miss, the missing tuples (or pages) are obtained by sending their identifiers to the server. These kinds of caching mechanisms cannot be directly applied in our setting, where specific data cannot be obtained on-demand from a single server. Instead, the required data is spread across online files, whereby a file needs to be fully downloaded before its contained data can be accessed.

Query caching (or semantic caching) caches query results instead of retrieved data. The cached query results are re-used to resolve future queries by using query folding techniques [75]. In case the cached query results are insufficient to answer a new query (i.e., cache miss), the system generates a remainder query to obtain the missing data from the server in a fine-grained way. Other context-aware systems also utilize caching. For instance, the Context-ADDICT approach [15, 62–64] proposes to use caching to deal with the transiency of data sources (i.e., sources becoming unavailable, for instance because of connectivity issues). However, concrete caching mechanisms are not elaborated. MobiSem [12] pro-actively replicates online RDF data locally on the device, to deal with data connection issues. To optimize this process, only data relevant to the user's context is replicated.

2.2.9.2 Cache replacement policies

A cache replacement policy determines what data should be discarded in case the storage medium is full. For instance, data can hereby be selected that has a low likelihood of being required in the future. To estimate this likelihood, a replacement policy typically assumes a certain locality of reference, indicating what kind of data items will be frequently referenced. For instance, temporal locality indicates that items that have been recently referenced will likely be referenced again. In order to indicate the likelihood of data items being referenced in the future, a replacement policy associates a removal value with each data item.

A lot of work has already been put in developing cache replacement policies for mobile settings. These policies typically rely on semantic locality, which is based on general properties and relations of data items. For instance, in [73], semantic locality indicates that query results, associated with physical locations closest to the user, will be frequently referenced. Similarly, the Furthest-Away-Replacement (FAR) policy [76] assumes that cached data, which is located in the user's movement direction and currently nearby, will be frequently referenced.

2.3 Contributions to the state of the art

In this section, we first position SCOUT with regards to existing definitions of context and context-aware features (section 2.3.1). Then, we indicate the contributions of our approach (sections 2.3.2 to 2.3.8) compared to current state of the art (see section 2.2).

At the end of the section (section 2.3.9), we present an overview that clearly indicates the extent to which other approaches exhibit (some form of) our contributed features.

2.3.1 Context and context-awareness

Our approach focuses on collecting descriptive online information on the user's environment, including nearby places, people and things (called physical entities) and keeps general user info as well, such as user preferences and device details. With regards to the above classifications, SCOUT thus collects identity [7], or user description / preferences and terminal characteristics [34]; and location context types [7, 34], including data on nearby physical entities and high-level spatial information. A mobile user's information needs are often related to his surroundings [12, 77]; for instance, finding nearby hotels matching the user's preferences [78] or getting tourist information related to nearby points-of-interest [9]. Consequently, nearby physical entities (e.g., hotels, points-of-interest) are often relevant to interactions between the user and mobile applications, in particular during information lookups. Given the context definition mentioned above, descriptions of these entities may thus be characterized as context. To collect this kind of context data, our approach is grounded in the discovery of links between physical entities and online data (see section 2.1.2).

Finally, SCOUT makes no assumptions on context-aware applications and their features, and instead focuses on supplying generic query access to the collected context. In order to support mobile applications that need to be reactive to the user's context in some way (e.g., adaptive software), push-based query access is provided as well.

2.3.2 Integrating the physical and the virtual world

Similar to [50], SCOUT leverages multiple linking mechanisms in parallel to locate online data sources describing physical entities in the user's vicinity. However, in contrast to the approach in [50], SCOUT utilizes multiple linking mechanisms to cope with newly discovered, a priori unknown environments. Firstly, the positioning method is supported, identifying online data sources associated with entities in a certain radius around the user. For this purpose, online semantic datasets (e.g., LinkedGeoData[26]) are contacted with the user's absolute position (e.g., obtained via GPS), which is resolved to online data source locations. This resembles indirect sensing from the CoolTown project [14, 44–47], whereby the online semantic dataset serves as resolver service. During indirect sensing, SCOUT thus leverages existing online services, which is comparable to HyCon [53]. Secondly, SCOUT supports the tagging method, whereby tags (e.g., RFID tags), put nearby physical entities, are read to obtain the entities' associated online source locations. In CoolTown, this is called direct sensing.

2.3.3 Context-acquisition software frameworks

It can be noted that the distributed approaches mentioned in [16, 36, 57] enable mobile applications to obtain a wide range of context information in well-outfitted smart spaces. At the same time, they relieve mobile devices of resource-intensive tasks such as context interpretation. On the other hand, they require context providers/widgets with computational capabilities, even in case of static context data, and thus need an external infrastructure hosting their software. In the CoolTown approach [47], web presences, providing information and services related to an entity, are hosted on the physical entity itself (e.g., hardware device such as a printer) or on a proprietary server. Therefore, their software needs to be installed on all these devices or on online servers. In contrast, SCOUT supports minimally outfitted environments without infrastructure, whereby all related computational tasks, such as context interpretation and integration, are performed on the device. As static context data, SCOUT re-uses online semantic data associated with the user's surroundings, which can be hosted on any web server.

The discussed distributed systems require mobile applications themselves to locate useful context providers or widgets. Similarly, the CoolTown web presence architecture requires mobile applications to manually navigate spatial relations, in order to obtain more information on nearby physical entities. In contrast, SCOUT offers an integrated, push- and pull-based queryable view over the integrated context information. Moreover, as in [16], SCOUT allows mobile applications to pose SPARQL queries referencing well-known domain-specific ontologies, making them independent of the specific provider / widget APIs (in contrast to e.g., [36, 57]). In the same vein, a serious drawback of the Hydrogen approach [59] is that the context is represented using discrete Java objects, ruling out expressive, unified query access and reasoning.

Finally, inspired by the CoolTown approach, SCOUT also creates high-level spatial relations (indicating nearness, containment) between the user and detected physical entities. As an extra step, SCOUT utilizes a generic, flexible mechanism to determine spatial relations, whereby a range of (inaccurate) location and detection data (e.g., detection range or maximum detection distance) can be employed.

[26] http://linkedgeodata.org/ (access date: 13/06/2013)

2.3.4 Obtaining static context data

The approaches relying on centralized data storage [25–27] share a problem common with other centralized architectures, namely that they necessitate an external infrastructure with its associated costs and other drawbacks. Distributed context providers [16, 36, 57] require computational capabilities, and thus also need to be deployed on an external infrastructure. Moreover, in order to ensure accurate and up-to-date information, all of the systems in this paragraph require static context data to be entered and maintained by content providers, for instance place owners or touristic services [25]. Even when data input is facilitated [25], it may be observed that much information on places and things is already available on the web, typically in the form of websites. Increasingly, websites are being semantically annotated (e.g., using RDFa) and are thus becoming fully-fledged (machine-readable) semantic sources. Other online semantic data sources include RDF files and online datasets (e.g., DBPedia, LinkedGeoData), which may also contain information on places and things. As a result, content providers will often need to maintain two information sources; their own online web presence (e.g., website, RDF data) and the context provider or centralized information system. Making matters worse, data stored by these systems can typically only be accessed by proprietary software, resulting in closed data silos. This increases the participation threshold for content providers, as compared to putting online data on freely accessible web servers.

In mobile scenarios, the concept of huge, freely accessible online semantic datasets (such as DBPedia and LinkedGeoData[27]) is very attractive, since they allow mobile apps to easily and efficiently obtain a wide range of context-specific information. Existing mobile systems such as DBPedia Mobile[28] and mSpace Mobile [65] access the query endpoints of such large online semantic datasets to realize their functionality (e.g., location-aware map view). SCOUT also utilizes such online semantic datasets to automatically discover online data sources describing the user's physical surroundings (in addition to other methods, such as tagging). However, in practice, many of the associated online query endpoints only offer access to a subset of their data (due to server hardware limitations), while they are also not (yet) supported by an infrastructure that allows scaling to a large number of clients. As such, they are currently not suited for widespread use by mobile clients. In SCOUT, we instead rely on relatively small online semantic sources, which can be individually and cheaply hosted on any web server.

2.3.5 Contextual information systems

A cited advantage of the approaches discussed in [10, 24] is that they only require a lightweight mobile client, since the server infrastructure takes care of data integration and query resolution. However, they require an external infrastructure, whereas significant increases in mobile device capabilities have empowered SCOUT to handle the necessary computational tasks locally on the mobile device. It can also be noted that these approaches rely on context sources with (varying)

[27] http://linkedgeodata.org/ (access date: 09/06/2013)
[28] http://wiki.dbpedia.org/DBpediaMobile (access date: 09/06/2013)

computational capabilities (e.g., to resolve subqueries), which need to be registered with the system. Although these sources are able to supply the information system with highly dynamic and up-to-date data (e.g., sensor readings), they present unnecessary overhead in case of static data.

Comparable to SCOUT, the Context-ADDICT [15, 62–64] and MobiSem [12] contextual information systems are deployed on the mobile device. Both systems perform pro-active data filtering, based on the relevance of the data to the user's context. In case of Context-ADDICT, this context-relevance is determined per application, via a CDT designed by the application designer. Therefore, a large workload is incurred at design time [62–64]. Moreover, this design-time specification requires knowledge on all possible data schema's beforehand. For MobiSem, context-relevance is determined based on various pre-configured factors, meaning mobile applications may only query data deemed context-relevant by MobiSem. As a result, a major drawback of the MobiSem approach is that certain query-relevant information may not be present in the final query dataset[29]. In contrast, SCOUT dynamically analyzes posed queries and automatically selects query-relevant data, and does not require a design time effort or a priori knowledge of used ontologies. In addition, it ensures that all potential query-relevant data is present in the final query dataset. On the other hand, this re-active strategy means query-relevant, uncached source data may have to be retrieved at query time, incurring runtime costs. We discuss mechanisms dealing with this issue in Chapter 4. We also note that MobiSem and Context-ADDICT only respectively supply programmatic / query-based context access in a pull-based way.

Finally, as mentioned, the Context-ADDICT approach dynamically discovers new sources at runtime. This is similar to SCOUT, where the Detection Layer (see Chapter 3, section 3.2) performs the discovery task and passes on location-relevant data sources to the query service[30]. On the other hand, the MobiSem approach relies on pre-configured online data sources, while the contextual information services in [10, 24] require context sources to be pre-registered.

2.3.6 Exploiting mobile device capabilities

Some related approaches exist that focus on performing all tasks locally on the device, such as MobiSem [12] and Hydrogen [59]. Although it is mentioned that Context-ADDICT [15, 62–64] can be deployed on both the client and server side, it does not deal with the particular challenges occuring from deployment on mobile devices (e.g., memory/processing limitations). MobiSem deals with mobile device limitations by only replicating relatively small amounts of semantic data to the device, and thus does not deal with issues resulting from processing large amounts of semantic web data. Comparably, to cope with device limitations, Hydrogen does not store vast amounts of context

[29] In fact, this is a potential drawback of the Context-ADDICT approach as well, although issues are less likely to occur since context-relevance can be determined per application.
[30] The query service may also be re-used by other applications, in which case other discovery criteria would be used depending on the application goal (e.g., an online academia finder).

history. In contrast, SCOUT fully exploits the memory, storage and processing capabilities of modern mobile devices to locally process large amounts of semantic data.

2.3.7 Data indexing

In this section, we highlight the contributions of the mobile query service compared to the state of the art on data indexing in fields such as context-aware systems (section 2.3.7.1), RDF query distribution (section 2.3.7.2) and RDF stores (section 2.3.7.3).

2.3.7.1 Context-aware systems

In our mobile query service, the Source Index Model (SIM) component indexes new online data sources, to later enable the identification of sources containing query-relevant data. More specifically, the SIM keeps combinations of metadata found in the sources, consisting of subject/object types and predicates. When a query is posed, the SIM extracts similar metadata from the query and automatically matches it to source metadata. Consequently, the indexed information resembles that of [16], where OWL classes and properties are kept; and of SOCAM [57], where OWL expressions are stored. However, in [16], the context consumer itself is responsible for checking whether a provider's OWL classes and properties matches its information needs. The Context Toolkit [43] requires context consumers to contact a centralized discoverer, and similarly check which context widgets serve desired context. In analogy to the Service Locating Service from SOCAM [57], the SIM instead automatically identifies useful context sources, given a posed query. However, the Service Locating Service is deployed on a server infrastructure (OSGi[31]) and does not run on the mobile device. In contrast, the SIM presents a standalone, mobile solution.

2.3.7.2 RDF Query distribution

Although we share a common goal, namely identifying query-relevant sources, the referenced query distribution approaches focus on keeping index information to optimize the query distribution process (for instance, statistical info and path structures to optimize local joins). Moreover, none of these approaches are deployed on a mobile device and deal with their hardware limitations. To speed up access times on mobile devices, the SIM component relies on a fast multi-level index, instead of representing the index data in RDF (as in DARQ [67] and SemWIQ [68]). Furthermore, the SIM focuses on collecting summary info that is more efficiently obtainable, namely metadata combinations (see Chapter 4, section 4.4). Finally, it can be noted that the SIM keeps pre-calculated index data, which may be problematic in case of source content changes (as mentioned above). However, this issue is reduced by keeping schema-level data instead of instance-level data (such as resource constraints and instance counts), which is less likely to change over time. Furthermore, a mechanism is in place to frequently update index information (see Chapter 4, section 4.5.3). Finally, our mobile query service also applies dictionary encoding (see Chapter 4, section 4.7.1.1).

[31] http://www.osgi.org/ (access date: 09/06/2013)

2.3.7.3 RDF stores

Although the RDF On The Go [28] and i-MoCo [58] stores prove that RDF stores can be deployed on mobile devices, the employed indexes still require a multitude of storage and memory space, as well as high insertion and update costs. The SIM index structure is similar to HexaStore, as it is a multi-level index where each path represents a metadata combination found in a source (see Chapter 4, section 4.4.1). However, the SIM keeps schema-level data (i.e., types, predicates) instead of instance data (i.e., resources) and supports only one access pattern, making it much smaller in size and quicker to update.

2.3.8 Client-side data caching

In this section, we indicate the contributions of the mobile query service with regards to client-side data caching, including existing data caching approaches (section 2.3.8.1) and cache replacement policies (section 2.3.8.2).

2.3.8.1 Data caching approaches

In our setting, query caching [75] has several drawbacks. Firstly, there is no possibility to efficiently retrieve non-cached data items via a remainder query, since the necessary data is captured in online files. Secondly, in our setting, new online sources are continuously discovered, which potentially contain data relevant to past and future queries. To cope with this, cached query results could be pro-actively updated with results from the newly encountered sources. On the other hand, this would also mean new sources will be ignored in case they are unrelated to a cached query, potentially leading to an under-utilization of storage space. Furthermore, queries unrelated to previously cached queries would yield a much higher execution time, since all encountered, query-relevant online sources need to be downloaded.

Our mobile query service utilizes a cache to store online source data discovered by the client (see Chapter 4, section 4.5). Multiple cache components were developed; in the Meta Cache component, cached triples are grouped via their shared semantics, i.e., predicates and subject/object types. Note that this is similar to semantic caching, which instead utilizes posed queries to define the shared semantics of cached data. In case of MobiSem [12], mobile applications can only access a priori replicated information, as deemed context-relevant by MobiSem. Instead, our cache component stores the online data discovered by the client, and thus more closely reflects application needs. Moreover, it allows re-using the caching mechanism outside of context-aware scenarios. Although Context-ADDICT mentions the use of caching, it does not elaborate on concrete caching mechanisms.

2.3.8.2 Cache replacement policies

We present a replacement policy called Least-Popular-Sources (LPS), which deals with our specific setting where required data is captured in online files. While the employed cache component may organize the cached data in any arbitrary way (e.g., Meta Cache organizes via shared metadata), the replacement policy always removes data on a per-source level. To reduce query-time overheads, the

LPS policy adopts a strategy where “popular” sources, which have a large impact on the cache (i.e., are involved in many cache units) and contain popular metadata (i.e., metadata often found in other sources), are retained. The source download cost can also be considered, in order to make sources that are expensive to download (i.e., have long download times) less likely to be removed. As shown in our evaluation (see Chapter 5, section 5.4.2), the LPS policy can reduce the number of cache misses considerably, as well as the number of required source re-downloads per cache miss. Finally, by not relying on a mobility-based removal strategy (as in [73, 76]), the mobile query service can be re-used by other systems that do not focus on querying location-based data.

2.3.9 Overview

This dissertation touches many aspects, both from the domain of mobile context-awareness and RDF querying. In this section, we present an overview that clearly indicates our core contributions compared to the state of the art. Section 2.3.9.1 deals with the SCOUT context-provisioning framework, while section 2.3.9.2 concerns the mobile query service.

2.3.9.1 SCOUT context-provisioning framework

In Table 2-1, we highlight the extent to which other approaches exhibit (some form of) the contributed features yielded by the SCOUT framework.

(A) Client-side deployment
(B) Multiple linking mechanisms in parallel
(C) Dynamic source discovery
(D) Re-use online data
(E) Supply & maintain spatial info
(F) Integrated context access
(G) Query access method
(H) Push- and pull-based access
(I) Semantic Web – Data integration
(J) Semantic Web – Reasoning

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
Hydrogen [59]	X							X		
MobiSem [12]	X			X[b]		X			X	
Context-ADDICT [15, 62–64]	X[1]		X	X[a,b]		X	X			
Location-aware platform [50]		X								
HyCon [53]				X[a]						
DBPedia Mobile[32]				X[b]						
mSpace mobile [65]				X[b]						
CoolTown [14, 44–47]					X					
CIS [10]						X	X			
Context-aware middleware [24]						X	X			
Distrib. context man. [16]							X[a]		X	
SOCAM [57]								X		X
Context Toolkit [36, 43]								X		
CoBrA-ONT [11]										X
SCOUT	X	X	X	X[a,b]	X	X	X[a]	X	X	X

Table 2-1. Overview of contributions for the SCOUT context-provisioning framework.

Below, we shortly elaborate on approaches indicated by a superscript index in the table.

(A) Client-side deployment

[1] Although both external and local deployment locations are envisioned, Context-ADDICT [15, 62–64] is not tailored to deal with issues occurring at any side (e.g., limitations of mobile hardware vs. communicating frequent location updates).

[32] http://wiki.dbpedia.org/DBpediaMobile (access date: 15/07/2013)

(D) Re-use online data
[a] The indicated approaches re-use online Web data.
[b] The indicated approaches re-use online Semantic Web data.

(G) Query access method
[a] The indicated approaches supply *semantic* query access.

2.3.9.2 Mobile query service

In Table 2-1, we highlight the extent to which other approaches exhibit (some form of) the contributed features yielded by the mobile query service.

(A) Mobile, client-side system	*(D) Automatic indexing*
(B) Data filtering	*(E) Schema-based indexing*
(C) Support online files	*(F) Data caching*

	A	*B*	*C*	*D*	*E*	*F*
Androjena[33]	X			X		
RDF On The Go [28]	X			X		
i-MoCo [58]	X			X		
MobiSem [12]		X[a]				X[1]
Context-ADDICT [15, 62–64]	X[1]	X[a]	X[1]	X[1]	X	
Query distribution [67–70]				X		
Path-based query distrib. [70]				X	X	
SemWIQ [68]				X	X	
Mobile query service	X	X[b]	X	X	X	X

Table 2-2. Overview of contributions for the mobile query service.

Below, we shortly discuss approaches indicated by a superscript index in the table.

(A) Mobile, client-side system
[1] As mentioned before, Context-ADDICT [15, 62–64] may be deployed both locally and externally, but does not deal with issues occurring at any side.

[33] http://code.google.com/p/androjena/ (access date: 15/07/2013)

(B) Data filtering

[a] The indicated approaches supply *pro-active* data filtering.

[b] The indicated approaches supply *re-active* data filtering.

(C) Support online files

[1] While Context-ADDICT [15, 62–64] supports a wide range of online sources by provisioning local wrapper components, it does not consider the issues resulting from accessing any particular type of source, such as individual online files.

(D) Automatic indexing

[1] We note that the indexing process is not fully automatic for Context-ADDICT, as an additional design-time effort is required to enable filtering based on the user's context.

(F) Data caching

[1] MobiSem [12] locally replicates or caches context-relevant data for local access, but does not organize it in a particular way to optimize retrieval times or memory requirements. Furthermore, MobiSem only caches very limited amounts of data.

2.4 Summary

This chapter gave an overview of background and state of the art, necessary to correctly position this dissertation. Regarding background, the chapter presented definitions of context and context-awareness, and elaborated on mechanisms of linking the physical to the virtual world, which are leveraged by SCOUT to retrieve online semantic data describing the user's environment. Afterwards, the chapter reviewed state of the art. In this review, we first discussed how Web and Semantic Web technology is currently being utilized in mobile computing. Then, we detailed aspects of related work where this dissertation made significant contributions. This involved discussing other approaches that exploit the aforementioned linking mechanisms to close the gap between the physical and digital world. We detailed context-acquisitioning frameworks, realizing bottom-up context capturing by abstracting low-level sensor data into high-level context. Systems were discussed where static context data is supplied via separate context provider components or centralized information systems, requiring content providers (e.g., place owners) to enter the data separately into the system. Also, we indicated systems that, comparable to SCOUT, leverage the Semantic Web as an online information platform. So-called contextual information services were detailed, which provide a unified, integrated query access over distributed context sources. SCOUT also belongs in this category, since it provides transparent, integrated access to online data sources describing the user's surroundings. We discussed state of the art pertaining to the full utilization of mobile device capabilities, an important characteristic of SCOUT.

The chapter further reviewed related work concerning the mobile query service. We indicated state of the art regarding data indexing and client-side caching in related fields. In these domains, data indexing is utilized to locate useful data sources and optimize data access. In distributed context-aware systems, the capabilities of context providers are often indexed to enable the easy identification of useful providers. RDF query distribution requires the indexing of data sources, so query execution can be distributed across query-relevant sources. RDF stores typically keep extensive indices in order to speed up access to RDF data, thus trading memory and storage space (as well as update times) for faster data retrieval. Client-side data caching aims to fully exploit the storage and processing capabilities of client devices. We presented traditional client-side caching systems as well as query caching (or semantic caching) approaches, and discussed the reasons why they cannot be directly applied to our particular mobile setting. Afterwards, the chapter mentions existing cache replacement policies that are fine-tuned to mobile settings.

Finally, we compared our work to the background and state of the art. Firstly, this involved positioning the SCOUT framework with regards to the presented classifications of context and context-awareness. We argued how SCOUT suits these definitions by supplying descriptive information on the user's surroundings, and by providing push- and pull-based context access features. Afterwards, for each relevant aspect, we indicated the core contributions of the SCOUT framework and mobile query service compared to the state of the art. In addition, an overview was given to indicate the extent to which the contributed features (i.e., features contributed by this dissertation) were supported by other related approaches.

Chapter 3

SCOUT, a Mobile Context-Provisioning Framework

In the previous chapter, we discussed the state of the art related to this dissertation. The chapter reviewed methods for linking the physical and the digital worlds, as well as context-provisioning approaches, whereby we focused on related aspects such as mobile context acquisitioning, integrated context access, and leveraging mobile device capabilities. We further presented related work concerning the mobile query service, reviewing data indexing and caching techniques in related domains (e.g., RDF stores, query distribution).

This chapter introduces SCOUT as a mobile, client-side context-provisioning framework. We discuss the layered architecture and elaborate on each layer, presenting its components, data structures and processes. Afterwards, we review the implementing package structure of each layer. Throughout the chapter, we motivate design decisions and indicate how the different layers cooperate to realize expressive, push- and pull-based query access to environment data. This chapter is structured as follows. First, we discuss the layered architecture of the SCOUT framework (section 3.1). In the following sections, we give an overview of each layer, being the Detection Layer (section 3.2), Spatial Layer (section 3.3), Environment Layer (section 3.4), and Applications Layer (section 3.5). In section 3.6, we review the core implementation of the different layers. Finally, section 3.7 provides a summary of this chapter.

3.1 Architecture overview

SCOUT is a mobile, client-side context-provisioning framework, which leverages the Semantic Web as an information platform for descriptive information on the user's physical environment. The different context-acquisitioning tasks performed by SCOUT are clearly separated and encapsulated in distinct layers, allowing different technologies and mechanisms to be plugged in at each level. Figure 3-1 shows the layered architecture of the SCOUT framework. Below, we shortly summarize each layer. A more elaborated explanation is given in the following sections.

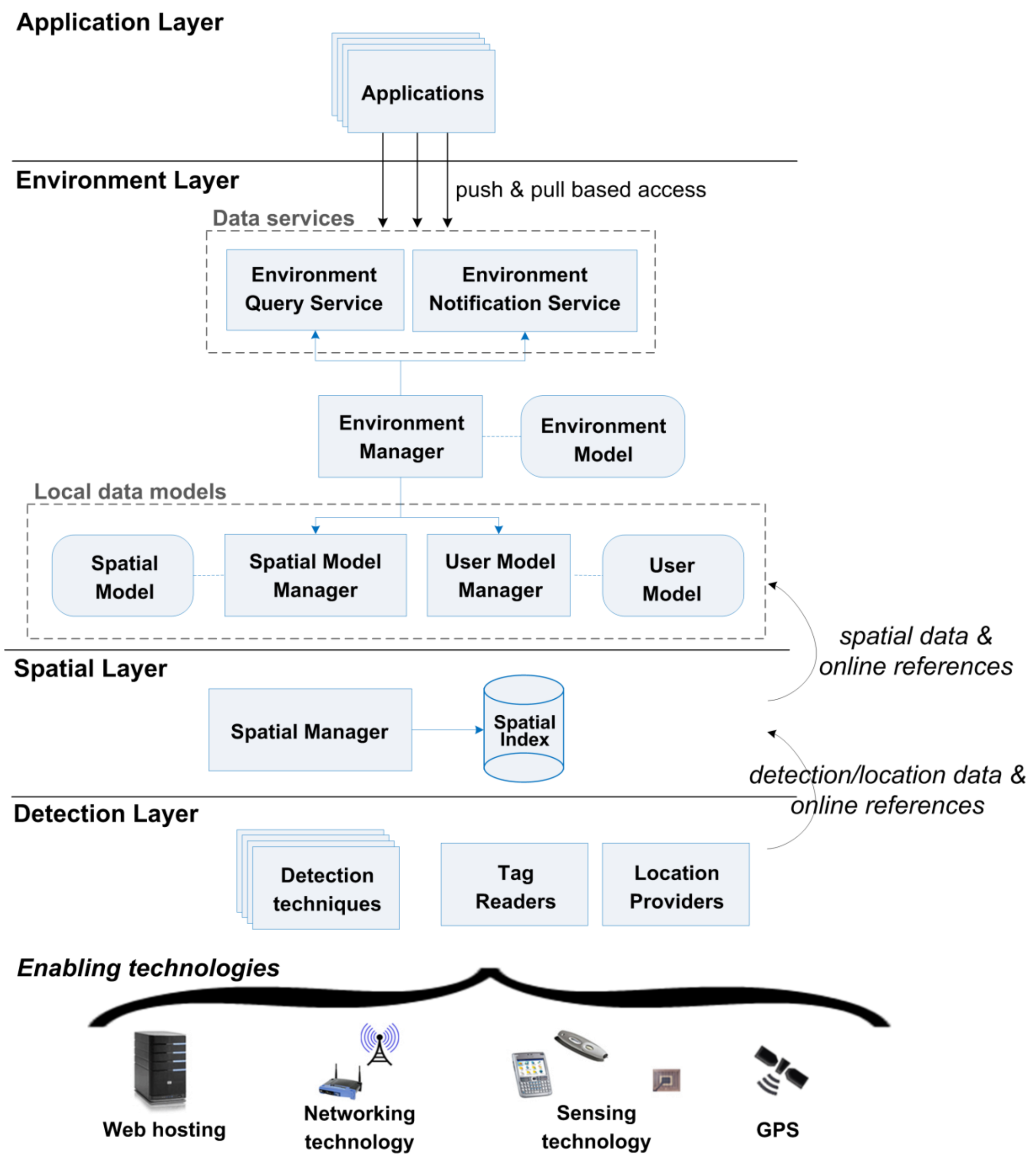


Figure 3-1. SCOUT architecture overview.

The bottom layer, the **Detection Layer**, is responsible for dynamically detecting physical entities in the vicinity, and locating their descriptive online semantic data sources. To perform these two tasks, the layer keeps so-called detection techniques. By supporting various detection techniques interchangeably, SCOUT can be deployed across new, heterogeneous environments outfitted with a range of different linking mechanisms. The detected locations of online data sources, as well as detection and location data (e.g., GPS coordinates, detection range) useful for inferring spatial information, is dynamically passed to the Spatial Layer for interpretation.

The goal of the **Spatial Layer** is to abstract low-level detection data supplied by the Detection Layer into high-level spatial information. This spatial information denotes whether the user is currently nearby a physical entity (e.g., a person) or inside a place (e.g., a building), or to denote proximity or containment between physical entities themselves. By provisioning high-level spatial data, mobile applications can easily obtain knowledge on nearby entities while still abstracting from low-level details (e.g., absolute coordinates, detection ranges). A flexible mechanism is in place, centered on a spatial index, to actively keep spatial information up-to-date. The inferred spatial information is communicated to the Environment Layer, accompanied by the detected locations of online data.

The **Environment Layer** keeps models and components enabling expressive, push- and pull-based query access across the environment data. The *Spatial Model* keeps spatial relations between physical entities and the user, reflecting the spatial information passed from the Spatial Layer. The layer further keeps a *User Model*, storing information useful for personalization (e.g., device properties, user characteristics and preferences). The *Environment Model* is an abstract, integrated view on the user's current and previous environments, and comprises the concrete User Model, Spatial Model, and the detected online semantic data sources. To transparently access the detected semantic dataset, consisting of large amounts of small online sources, the layer utilizes the mobile query service. We discuss this query service in more detail in Chapter 4. Mobile applications can access the Environment Model in a push- and pull-based way, respectively via the Environment Notification Service and Environment Query Service.

Finally, the **Application Layer** comprises mobile applications built on top of the SCOUT framework. These applications utilize the context services supplied by the Environment Layer to obtain expressive, push- and pull-based access to environment context. Three mobile applications currently rely on these services to realize their functionality: the *Person Matcher* [31], a mobile app that identifies nearby people of interest; *COIN* [29, 32], a client-side web augmentation approach; and *AdaptIO* [30], which adapts mobile interaction obtrusiveness. Chapter 6 details these applications.

In the sections below, we elaborate on each of these layers.

3.2 Detection Layer

The Detection Layer automatically detects physical entities in the user's vicinity, and locates their associated online semantic data. To achieve this, the layer keeps detection techniques, which rely on linking mechanisms connecting the physical to the digital world (see Chapter 2, section 2.1.2).

Below, we mention useful linking mechanisms, discussing their benefits and drawbacks (section 3.2.1). Then, we elaborate on detection techniques and how they utilize these mechanisms (section 3.2.2). An overview of the core implementation of this layer can be found in section 3.6.1.

3.2.1 Linking mechanisms

In the context-aware literature, two linking mechanisms are typically used to connect the physical world to the virtual; tagging and positioning (see Chapter 2, section 2.1.2). For tagging, an identification tag (or beacon) is attached directly on or nearby the physical entity and loaded with related (locations of) data. Example tagging technologies include Radio-Frequency Identification (RFID) / Near Field Communication (NFC), Quick Response (QR) codes or Bluetooth. In case of positioning, the physical entities' absolute positions are exploited to retrieve their associated data. For instance, clients can contact an online service (e.g., LinkedGeoData[34]) to obtain information describing entities in a certain radius around the user.

Tagging requires outfitting physical spaces with RFID/NFC tags, QR codes or Bluetooth beacons (in [14], this process is called physical registration). As such, this linking mechanism incurs a serious setup effort. On the other hand, tagging directly supports entities with dynamic locations (e.g., people, moving art exhibits). Moreover, tagging technologies are already being deployed on a sizable scale. According to a recent study, usage of QR codes in Europe doubled in 2012, leading to an estimated 17,4 million users scanning QR codes[35]. RFID tags are seeping into everyday life as well, embedded inside personnel ID cards and public transportation tickets, put on library books and retail goods, and used to quickly perform wireless payments[36]. Although the outlook seems promising, neither QR codes nor RFID tags are yet deployed on large enough scale whereby arbitrary physical spaces are connected to online information. Furthermore, it should be noted that not all tagging technologies currently support the fully automatic detection of tagged entities, and require some user interaction. For instance, QR codes need to be manually scanned with a camera, while low-distance RFID readers require close proximity.

Compared to tagging, positioning does not rely on an outfitted environment. Instead, it only requires the user's mobile location, captured either via outdoor (e.g., GPS) or indoor (e.g., IR beacons; WiFi RSSI [50]) positioning technology. However, a service is also needed that keeps absolute positions of

[34] http://linkedgeodata.org/ (access date: 13/06/2013)
[35] http://www.comscore.com/Insights/Press_Releases/2012/9/QR_Code_Usage_Among_European_Smartphone_Owners_Doubles_Over_Past_Year (access date: 04/03/2013)
[36] http://www.paypass.com/ (access date: 13/06/2013)

physical entities in the region, resulting in infrastructure costs. This drawback can be reduced by relying on existing datasets and APIs, such as LinkedGeoData and the Google Places RESTful API[37]. LinkedGeoData provides a query endpoint that can be used to retrieve semantic data on nearby entities, in a radius around the user's current location. While the Google Places API does not allow retrieving arbitrary semantic data, it supplies limited structured data such as name, type, opening hours, etc., as well as a URL pointing to more information[38]. Finally, it can be noted that the positioning mechanism does not directly support physical entities with dynamic locations (e.g., people), as this would require continuously updating the online server with new locations.

From the above, it may be concluded that both linking mechanisms suffer from shortcomings, ranging from inadequate deployment and necessitating manual effort to requiring external online services. Some of these issues will likely be resolved over time, by for instance deploying tags on a larger scale, or increasing RFID reader range to reduce manual effort. However, at this time, it is clear that a context-acquisition framework should focus on leveraging both linking mechanisms to robustly support heterogeneous environments.

3.2.2 Detection techniques

In order to support a wide range of heterogeneous environments, multiple linking mechanisms need to be used interchangeably and in parallel. The Detection Layer comprises detection techniques, which rely on linking mechanisms to *detect physical entities* in the user's surroundings, and *locate associated semantic data*. Reflecting the nature of SCOUT as a client-side approach, with minimal reliance on external infrastructures, detection techniques are grouped into two categories; direct detection techniques, which directly detect physical entities in the user's vicinity (e.g., via RFID readers), and indirect detection techniques, which rely on an external service for detection. Below, we elaborate on both categories.

Direct detection techniques utilize a hardware component (e.g., RFID reader, camera) to directly detect physical entities and locate their online semantic data. For instance, an RFID reader automatically detects nearby tags and reads their contents (e.g., URL); while users can point their smartphone cameras at QR codes to decode them into URLs. As such, these detection techniques leverage the tagging mechanism (see section 3.2.1). In our case, tags need to be loaded with the locations of the associated semantic data. Currently, the Detection Layer provides support for the following direct detection techniques: RFID tags, QR codes, and Bluetooth beacons.

Indirect detection techniques utilize external services for detection, whereby the local client passes on a piece of mobile context (e.g., current location). For instance, the client encodes the user's location in a query to the LinkedGeoData query endpoint, which returns semantic data on nearby physical entities. In the result data, the entity's resource URI (or e.g., rdfs:seeAlso value) can point to an online semantic source. Consequently, these detection techniques typically leverage the

[37] https://developers.google.com/places/documentation/ (access date: 13/06/2013)
[38] https://developers.google.com/places/documentation/details#PlaceDetailsResults (access date: 04/03/2013)

positioning linking mechanism (see section 3.2.1). Comparable to direct detection, these online services need to return the locations of online metadata describing physical entities. Currently, the Detection Layer supplies techniques that utilize the LinkedGeoData query endpoint and the Google Places API. A custom “location directory” service is also supported, which needs to be deployed on a remote server. As this location directory allows quickly and easily setting up new environments, this service was often used in experiments.

Aside from online semantic data locations, a detection technique also needs to return detection data that can be used to infer spatial relations in the Spatial Layer (see section 3.3). This detection data can range in accuracy from the entity’s precise coordinates (e.g., retrieved from an online service in case of indirect detection) to the technology’s maximum detection range (e.g., in case of the RFID detection technique). Furthermore, a detection technique may also *directly* return associated metadata, as opposed to online locations of such metadata. For instance, in case of high-capacity RFID tags, this metadata can be directly read from the tag; or indirect detection techniques can directly retrieve this data from the online services. Importantly, we envision this directly returned metadata to encode geographical data, such as the physical entity’s geometry (e.g., for buildings) and absolute coordinates, described using a domain-specific ontology (e.g., GeoFeatures[39]). This information can be used to infer spatial relations more accurately (see section 3.3.1).

3.3 Spatial Layer

The responsibility of the Spatial Layer is to supply, and maintain, high-level spatial information, based on detection information received from the Detection Layer. This spatial information denotes whether the user is nearby, or inside, another physical entity (e.g., building); and whether physical entities themselves are nearby each other or containing one another.

A flexible mechanism is in place to infer and maintain spatial information. This mechanism is centered on a spatial index, which stores the spatial shapes of detected physical entities and the user. Currently, the spatial index is implemented using an R-tree. We consider an entity’s spatial shape to denote its geometric shape (i.e., point, circle or polygon), as well as its absolute location (e.g., vertex coordinates). Using the spatial index, it can be checked whether a newly detected entity is nearby (or inside / containing) the user or another, previously detected physical entity.

Below, we elaborate on the process flow occurring in the Spatial Layer whenever new entities are detected, or when the user’s absolute position changes (section 3.3.1). The core implementation of this layer is reviewed in section 3.6.2.

3.3.1 Inferring high-level spatial information

The Spatial Layer receives notifications from the Detection Layer whenever new entities are detected. Also, the layer is notified in case the mobile user’s location changes, since this may lead to

[39] http://www.mindswap.org/2004/geo/geoOntologies.shtml (access date: 08/03/2013)

some entities (no longer) being spatially related. Below, we elaborate on the process flow that ensues when receiving such notifications.

In order to infer spatial information, the Spatial Layer relies on a spatial index, which stores the spatial shapes of physical entities (including detected entities as well as the user). Therefore, a first step after receiving a detection event is to approximate the spatial shape of the detected entity, including its geometric shape and absolute location. This step may be skipped in case of a location update event, as the passed user's spatial shape already includes the correct geometric shape[40] and location (e.g., obtained via GPS). Afterwards, the new spatial shape is passed to the spatial index, returning entities that are spatially related to the detected entity or the user. After searching, the spatial shape is added to the spatial index, possibly replacing its previous entry. Based on the search results, two lists are created, respectively containing 1/ physical entities nearby the particular entity, and 2/ entities that are no longer nearby. This spatial info is passed to the Environment Layer, enabling the layer to manage spatial relations (see section 3.4.2.2). Below, we elaborate on each step in more detail.

Step 1. Approximating the spatial shapes

For detected entities, associated spatial shapes need to be derived based on the available detection data. The accuracy of the derived spatial shapes depends on the precision of the detection data. Three cases are possible:

1/ *Exact spatial shape is known:* A detection technique may directly return metadata describing the physical entity, in addition to the location of online semantic data (see section 3.2.2). In case this metadata encodes a geographical description (described using a supported format), the exact spatial shape is immediately available. For instance, such metadata can be directly returned by an online service utilized by the detection technique, and can encode the entities' coordinates and geographic shape (e.g., point, polygon).

2/ *Precise detection range is known:* In case the exact spatial shape is not known, it needs to be approximated. Certain technologies, such as RFID, can return the exact range at which the entity was detected [79]. If the detection range is known, it may be inferred that the entity's geometric shape is a point located on a circle's edge, which is centered on the user's position (at detection time) with as radius the detection range.

3/ *Maximum detection range is known:* In case no other detection data is available, we need to rely on the maximum detection range of the employed detection technique. In this case, it is inferred that the entity represents a geometric point located somewhere inside a circle, centered on the user's position (at detection time) with as radius the detection range. Clearly, this represents the most imprecise approximation.

[40] For instance, the user's geometric shape can be a point or (small) circle.

Step 2. Utilizing the spatial index

In the second step, the spatial index is searched to retrieve entities that are nearby or inside/contained by either the detected entity, in case of a detection event; or the user entity, in case of a location update. To perform the search, the spatial index requires two pieces of information. Firstly, it needs the particular entity's spatial shape, possibly approximated in the previous step. Secondly, it requires the maximum distance where entities are still considered to be nearby, called the "nearness distance". This information is passed to the spatial index in the form of a spatial query.

After the search has been performed, the entity in question is added to the spatial index, together with its spatial shape. In case the entity was already present (e.g., it was detected before), its entry will be replaced by the new spatial shape. In section 3.6.2.3, we elaborate on the implementation of the spatial index.

Step 3. Determining spatial information

The final step involves determining 1/ which physical entities are spatially related with the particular entity (i.e., detected physical entity or the user), and 2/ which entities are no longer spatially related. The physical entities returned from the spatial index search (see above) constitute the first list.

In order to obtain the second list, the Spatial Layer first requires the list of physical entities *previously* found to be spatially related to the particular entity. For this purpose, the Environment Layer is contacted, where the necessary information is encoded in the Spatial Model[41] (see section 3.4.2). Afterwards, it is determined whether these physical entities are still spatially related to the entity in question. In more detail, this is done by checking whether the spatial shapes of the aforementioned physical entities are still nearby the particular entity's spatial shape, given the predefined nearness distance; or whether the spatial shapes are still containing one another, given their geometric shapes and absolute locations.

After constructing the two lists, the Environment Layer is contacted, passing along the entity in question together with its list of spatially related / non-spatially related entities. The related spatial shapes, which may be potentially relevant to mobile applications (e.g., mobile apps with a map view), are also supplied. Based on this information, the Environment Layer creates, updates or invalidates high-level spatial relations between physical entities. For more information on this step, we refer to section 3.4.2.2.

3.4 Environment Layer

The Environment Layer provides applications with an abstract, integrated view of the user, his environment, and the physical entities in it. This view is called the *Environment Model*. The Environment Layer maintains two local, concrete models, which provide vital information for this view. The *User Model* stores the user's characteristics, preferences, and device information; while

[41] Keeping this data separately in this layer would result in duplicating the spatial data.

the *Spatial Model* encodes high-level spatial information between the user and physical entities, in the form of spatial relations. Both models have associated components to manage access and keep the information up-to-date.

A third, essential part of the Environment Model constitutes the online semantic dataset, which comprises descriptive information on physical entities in the user's vicinity. To achieve integrated query access to this online dataset, the Environment Layer employs the mobile query service, which is detailed in Chapter 4. Mobile applications utilize the *Environment Notification Service* and *Environment Query Service* to achieve push- and pull-based access to the Environment Model, respectively.

Below, we elaborate on the two concrete models and their manager components. Then, we detail the Environment Model. The core implementation of this layer is summarized in section 3.6.3.

3.4.1 User Model

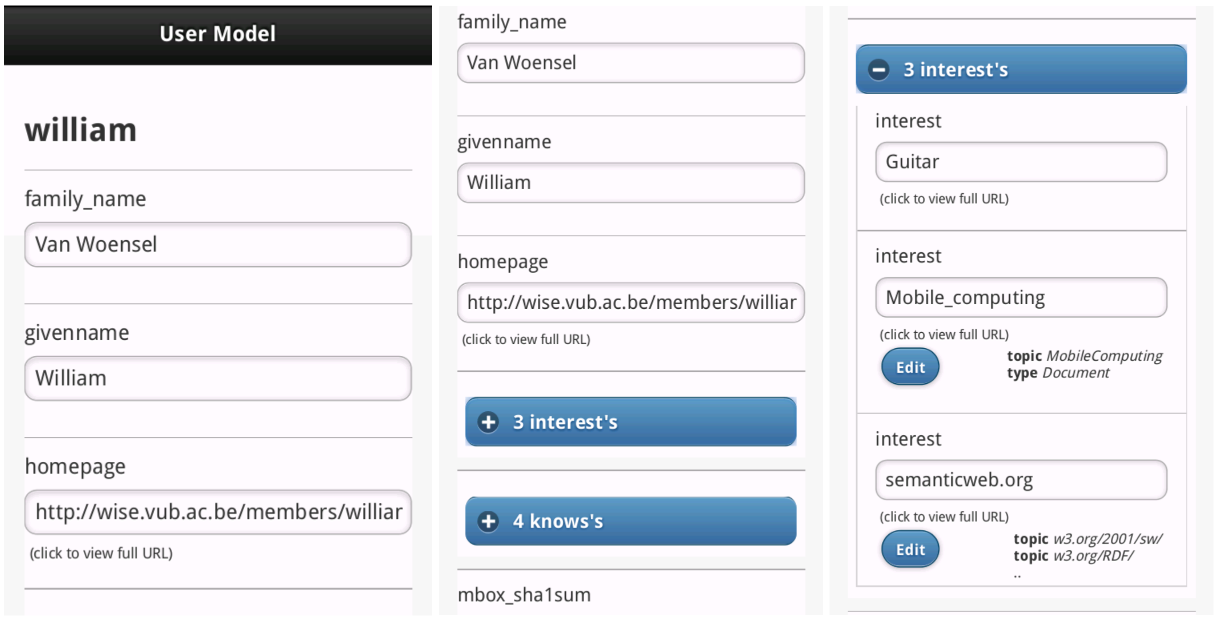
The User Model stores the user's characteristics and preferences, as well as device information, respectively using ontologies such as Friend of a Friend[42] (FOAF) and Composite Capabilities/Preferences Profiles[43] (CC/PP). As a result, the User Model enables mobile applications to personalize their content and functionality to the user and his device. An example User Model instance, as well as the User Model ontology can be found in Appendix A and on http://wise.vub.ac.be/william/phd/index.htm#scout.

The User Model management component keeps the User Model data in an RDF graph. We supply the user with a mobile configuration tool, which allows him to update and remove personal information from the User Model. We shortly discuss this tool below.

3.4.1.1 Mobile User Model configuration tool

In Figure 3-2 and Figure 3-3, we show screenshots of the developed configuration tool, which features a mobile user interface to update the User Model.

[42] http://xmlns.com/foaf/spec/ (access date: 14/06/2013)
[43] www.w3.org/Mobile/CCPP/ (access date: 14/06/2013)



User Model

william

family_name
Van Woensel

givenname
William

homepage
http://wise.vub.ac.be/members/williar
(click to view full URL)

family_name
Van Woensel

givenname
William

homepage
http://wise.vub.ac.be/members/williar
(click to view full URL)

3 interest's

4 knows's

mbox_sha1sum

3 interest's

interest
Guitar
(click to view full URL)

interest
Mobile_computing
(click to view full URL)
Edit
topic *MobileComputing*
type *Document*

interest
semanticweb.org
(click to view full URL)
Edit
topic *w3.org/2001/sw/*
topic *w3.org/RDF/*
..

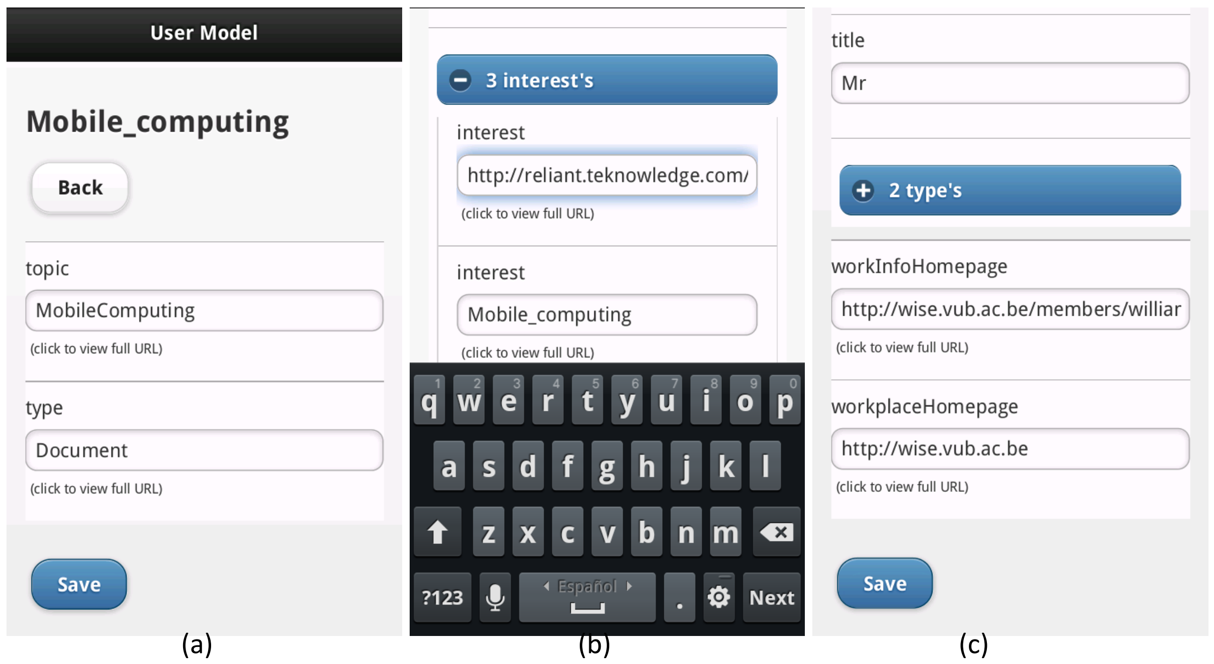
(a) (b) (c)

Figure 3-2. *SCOUT Environment Layer:* Screenshots of the mobile User Model configuration tool (1).

The mobile UI consists of a navigable form, loaded with information from the User Model (see Figure 3-2/a). An important goal of the UI is to provide a good overview of the user model data, to increase usability and cope with limited mobile screen sizes. Firstly, properties sharing the same predicate (e.g., interest) are grouped into a collapsible list (see Figure 3-2/b), which can be unfolded to show the individual properties (see Figure 3-2/c). Secondly, property values which themselves have associated properties (e.g., Mobile_computing interest, see Figure 3-2/c), are accompanied by a short summary. To view and edit all of its properties, the user can click the Edit button at the left bottom of the property value's field. After clicking, the form is loaded with the property value's own description, as shown in Figure 3-3/a. In order to navigate back to the previous form, the user clicks the Back button. Property values can be navigated this way up to an arbitrary depth.

An effort is also made to make RDF values more readable, by reducing URIs to their local names[44]. This is exemplified in Figure 3-2/c, where the URI http://ebiquity.umbc.edu/ontology/research.owl#Mobile_computing is reduced to its local name Mobile_computing. When clicking on a field, the full URI is shown (see Figure 3-3/b). At any time, the user can click the Save button to store the updated information (see Figure 3-3/c).

[44] An exception is made for URLs pointing to online information, as indicated by the related predicate (e.g., foaf:workInfoHomepage; see Figure 3-3/c)



User Model

Mobile_computing

Back

topic

MobileComputing

(click to view full URL)

type

Document

(click to view full URL)

Save

(a)

3 interest's

interest

http://reliant.teknowledge.com/

(click to view full URL)

interest

Mobile_computing

(click to view full URL)

(b)

title

Mr

2 type's

workInfoHomepage

http://wise.vub.ac.be/members/williar

(click to view full URL)

workplaceHomepage

http://wise.vub.ac.be

(click to view full URL)

Save

(c)

Figure 3-3. *SCOUT Environment Layer:* Screenshots of the mobile User Model configuration tool (2).

In section 3.6.3.1, the implementation of the configuration tool is detailed.

3.4.2 Spatial Model

The Spatial Model encodes high-level spatial information about the user's environment. More specifically, it keeps time-stamped spatial relations between the user and physical entities, as well as between physical entities themselves. As a result, the Spatial Model enables mobile applications to become location-aware, without having to deal with low-level data (e.g., position readings or sensing data). Based on notifications from the Spatial Layer (see section 3.3), the Spatial Model manager component keeps the Spatial Model up-to-date.

Below, we detail the contents of the Spatial Model (section 3.4.2.1). Then, we elaborate on how it is kept up-to-date (section 3.4.2.2).

3.4.2.1 Spatial Model contents

The Spatial Model keeps spatial information in the form of high-level spatial relations. Concretely, the model corresponds to a reified RDF graph, representing each spatial relation as a reified RDF statement. These spatial statements have as subject and object the URI resources identifying the participating entities, while their predicate indicates the type of spatial relation. The Spatial Model ontology (with namespace prefix "sm") defines these predicates, which include sm:isNearby, sm:contains and sm:containedIn, respectively denoting nearness and (reverse) containment. In case two physical entities are no longer spatially related, the spatial relation is invalidated, which involves replacing the predicate of the spatial statement by sm:wasNearby, sm:contained and sm:wasContainedIn, respectively.

By reifying these spatial statements, additional facts can be stated about them. Currently, the Spatial Model keeps timestamps for each spatial statement using the sm:from and sm:to predicates, indicating the interval during which the spatial relation was valid. Other pieces of information could also be kept about these spatial statements, such as the level of certainty (e.g., in case of approximated positions; see section 3.3.1). Since spatial relations are reciprocal (e.g., a user *contained in* a building means that the building *contains* the user), each conceptual spatial relation is serialized twice in the model. In addition, the Spatial Model keeps useful information passed from the layers below. In particular, it stores the spatial shapes of the detected entities, which indicate the physical entity's geometric shape and coordinates, and serializes these shapes as RDF data using the GeoFeatures[45] ontology. This RDF-encoded location is indicated using the sm:lastKnownLocation predicate. For instance, this spatial data can be useful for mobile applications keeping a map view.

Figure 3-4 shows the graphical representation of an example Spatial Model. In this model, the user (http://wise.vub.ac.be/william) was contained in (sm:wasContainedIn) the athletics track (http://www.vub.ac.be/campus/atletiekpiste). Currently, the user is nearby (sm:isNearby) 't Complex (http://tcomplex.be/rdf) and contained in (sm:containedIn) the HealthCity fitness centre (http://healthcity.be/etterbeek). Furthermore, 't Complex (http://tcomplex.be/rdf) itself is also located nearby (sm:isNearby) the HealthCity fitness centre (http://healthcity.be/etterbeek). Each spatial statement is reified and accompanied by timestamps (sm:from, sm:to), indicating the start-time, and potentially end-time, of the spatial relation. Each of entity URI resources are also linked to their last known absolute location (indicated by sm:lastKnownLocation), which is represented using the GeoFeatures ontology (for brevity, we only show the full location for the user).

[45] http://www.mindswap.org/2004/geo/geoOntologies.shtml (access date: 08/03/2013)

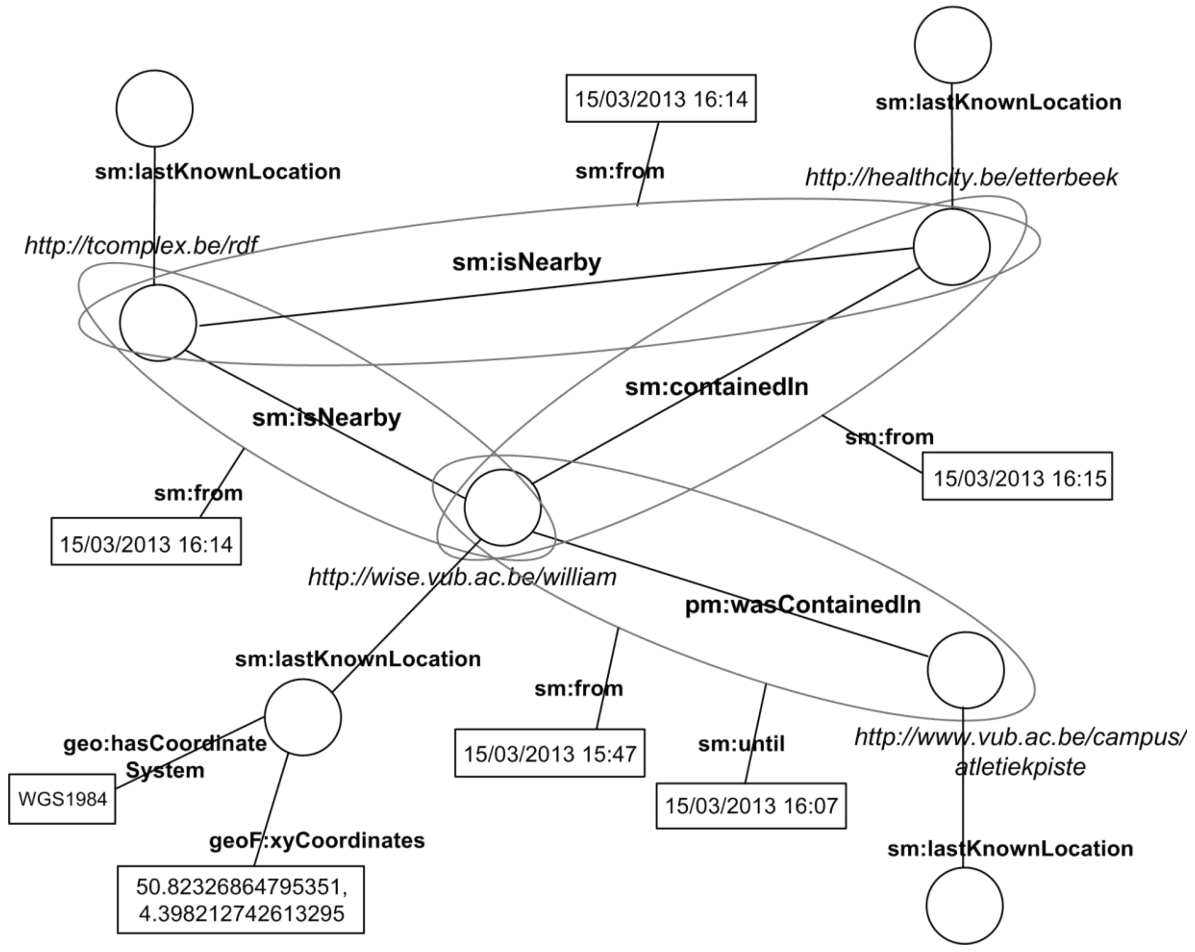


Figure 3-4. *SCOUT Environment Layer:* Example Spatial Model.

The Spatial Model ontology, as well as an example Spatial Model instance, can be found in Appendix A and on http://wise.vub.ac.be/william/phd/index.htm#scout. Below, we elaborate on the mechanism keeping the Spatial Model up-to-date.

3.4.2.2 *Keeping the Spatial Model up-to-date*

In case the user's location has changed, or a new physical entity was detected, the Spatial Layer passes two lists of entities; respectively keeping spatially related physical entities, and entities that are no longer spatially related (see section 3.3.1).

For each of the entities in the first list, a new spatial relation is created with the particular entity (i.e., user or detected entity) if none yet exists. For the second list, each existing spatial relation between the listed entities and the entity in question is invalidated. As mentioned, this involves replacing the spatial relation predicate (e.g., replacing sm:isNearby with sm:wasNearby) as well as adding an invalidation timestamp (indicated by sm:to). For each creation and invalidation event, the RDF-encoded locations of the participating entities are updated in the Spatial Model, thus supplying mobile applications with the entities' last known locations (indicated via sm:lastKnownLocation). The user's last known location is continuously updated in the model as he is moving around, based on location updates from the Spatial Layer.

3.4.3 Environment Model

The Environment Model represents an abstract, integrated view on the user's physical environment and context, comprising the concrete User Model, Spatial Model and assembled online semantic sources describing physical entities. It allows mobile applications to fire queries referencing any part of the user's current and previous physical surroundings, as well as the user's profile, without having to separately query each data source. Instead, the Environment Model manager combines the necessary data sources at query time, executes the query, and returns the results.

An important part of the Environment Model is the online semantic dataset describing the user's current and previous surroundings. In order to access this dataset, the Environment Model manager component relies on our mobile query service (see Chapter 4), which allows for the transparent querying of an online semantic dataset comprising small semantic sources. In order to outline our relevant dataset, the Environment Model manager communicates the online locations of semantic data, describing nearby physical entities, to the query service. When executing a query, the manager contacts the query service to obtain a selection of the online dataset relevant to the given query. This query-relevant selection, together with the concrete User Model and Spatial Model, is assembled into an RDF graph on which the query is executed.

Mobile applications can access the Environment Model in a push- and pull-based way, respectively via the Environment Notification Service and Environment Query Service. In section 3.6.3.2, an overview of the Environment Model implementation can be found.

3.5 Application Layer

The Application Layer comprises mobile applications that utilize the context services supplied by the Environment Layer. Below, we shortly summarize each application, focusing on how they utilize these services to realize their functionality. In section 3.6.4, we discuss in more detail how these mobile applications invoke the SCOUT API. For more information on these applications, we refer to Chapter 6.

- The *Person Matcher* [31] mobile application pro-actively identifies nearby people of interest, and pushes them to the user. The app demonstrates the data filtering capabilities of the SCOUT framework, by utilizing the push-based query access to be notified of nearby people and their online FOAF profiles.
- *COIN* [29, 32] stands for a client-side web augmentation approach, which injects context-aware features into existing websites on-the-fly. In this way, visited websites can be automatically enriched to suit the mobile user's needs, which are often related to their current environment (e.g., find the closest metro station connecting to the university campus). To achieve this

augmentation task, COIN thus requires access to rich environment context, while it should also be notified of context changes to keep injected features up-to-date. For this purpose, it utilizes the push-and pull-based context access services provided by SCOUT.

- *AdaptIO* [30] dynamically adapts the obtrusiveness of mobile interactions to suit the user's current situation. In order to accurately define and determine obtrusive situations (e.g., in a meeting), across heterogeneous and previously unknown environments, AdaptIO requires rich environment context, potentially collected from minimally outfitted surroundings. AdaptIO relies on SCOUT to retrieve the desired context, as SCOUT is particularly suited to collect context in heterogeneous (and minimally outfitted) environments, by utilizing multiple detection techniques interchangeably and in parallel.

3.6 Implementation

This section summarizes the core implementation of the different layers, being the Detection Layer (section 3.6.1), Spatial Layer (section 3.6.2), and Environment Layer (section 3.6.3). These sections are accompanied by UML class diagrams showing the core classes[46] used in the implementation. Finally, we illustrate how three existing mobile applications, situated in the Application Layer, concretely invoke the SCOUT context services API (section 3.6.4).

3.6.1 Detection Layer

This section presents a bottom-up overview of the Detection Layer implementation. We start by discussing classes that wrap concrete sensing technologies (e.g., GPS, RFID), which enable detection techniques to leverage linking mechanisms (e.g., positioning, tagging) while abstracting from technology-specific details and drivers (section 3.6.1.1). Then, we detail the detection technique implementation (section 3.6.1.2).

3.6.1.1 Sensing technologies

This section discusses abstract sensing classes, including TagReader and LocationProvider, which encapsultate the sensing capabilities of mobile devices (i.e., tag reading and position capturing). Subclasses of these abstract classes wrap specific sensing hardware and drivers, for instance the Cathexis IDBlue RFID reader[47] and the Android GPS API.

To cope with the heterogeneity of sensing technologies across mobile devices, we utilize the Abstract Factory design pattern [80]. We also observe that most sensing technologies inherently work in a push-based way, whereby new sensing results are pushed to interested clients (in our case, detection techniques). For instance, the user utilizes the camera to capture QR codes, after which the codes

[46] To reduce complexity in these diagrams, private members, constructors and getter/setter methods are left out, as well as typically required classes such as Exceptions.
[47] http://idblue.com/ (access date: 13/06/2013)

need to be pushed to the QR detection technique. In other cases, sensing actions should be automatically performed at certain moments, after which the results are pushed to related detection techniques. For instance, as the mobile user is walking around, RFID readers should frequently read nearby tags, in order to detect the user's new surroundings; whereby reads occur after certain time intervals, or after the mobile user has moved a certain distance. In this example, the RFID detection technique needs to be automatically kept up-to-date of the data read by the RFID reader. The abstract sensing classes implement the Observer design pattern [80], whereby listeners are notified in case of new sensing results and can specify the desired sensing moments. We discuss the abstract sensing classes, together with their implementing subclasses, below.

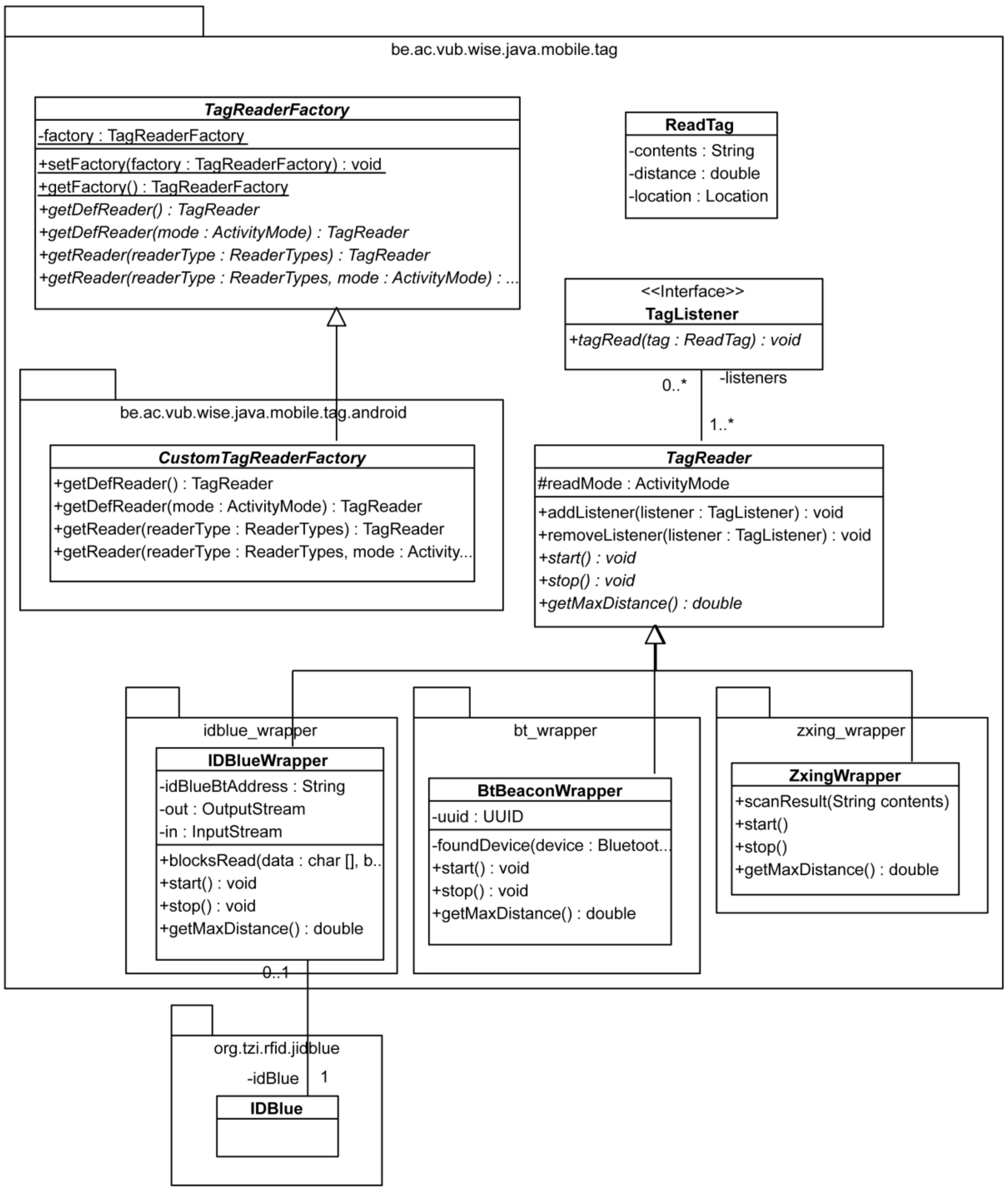

be.ac.vub.wise.java.mobile.tag
TagReaderFactory
-factory : TagReaderFactory
+setFactory(factory : TagReaderFactory) : void
+getFactory() : TagReaderFactory
+getDefReader() : TagReader
+getDefReader(mode : ActivityMode) : TagReader
+getReader(readerType : ReaderTypes) : TagReader
+getReader(readerType : ReaderTypes, mode : ActivityMode) : ...
ReadTag
-contents : String
-distance : double
-location : Location
<<Interface>>
TagListener
+tagRead(tag : ReadTag) : void
0..*
-listeners
1..*
be.ac.vub.wise.java.mobile.tag.android
CustomTagReaderFactory
+getDefReader() : TagReader
+getDefReader(mode : ActivityMode) : TagReader
+getReader(readerType : ReaderTypes) : TagReader
+getReader(readerType : ReaderTypes, mode : Activity...
TagReader
#readMode : ActivityMode
+addListener(listener : TagListener) : void
+removeListener(listener : TagListener) : void
+start() : void
+stop() : void
+getMaxDistance() : double
idblue_wrapper
IDBlueWrapper
-idBlueBtAddress : String
-out : OutputStream
-in : InputStream
+blocksRead(data : char [], b..
+start() : void
+stop() : void
+getMaxDistance() : double
bt_wrapper
BtBeaconWrapper
-uuid : UUID
-foundDevice(device : Bluetoot...
+start() : void
+stop() : void
+getMaxDistance() : double
zxing_wrapper
ZxingWrapper
+scanResult(String contents)
+start()
+stop()
+getMaxDistance() : double
0..1
org.tzi.rfid.jidblue
-idBlue
1
IDBlue

Figure 3-5. *SCOUT Detection Layer*: TagReader package.

Figure 3-5 shows the TagReader package. This package contains classes supporting the tag reading capabilities of mobile devices (e.g., RFID), and is used by detection techniques relying on the tagging linking mechanism (e.g., direct detection techniques). The TagReader class wraps tag reading technologies such as RFID, QR and Bluetooth. TagReader provides push- and pull-based access to tag contents, whereby the former involves notifying registered TagListeners with the tag contents. These tag contents are wrapped in a ReadTag instance, keeping the read string as well as the detected range of the tag and its absolute location (if any). There are currently three implementation classes:

IDBlueWrapper, which wraps the Bluetooth driver communicating with the external Cathexis IDBlue pen (see org.tzi.rfid.jidblue package); QRWrapper, which wraps the Zxing QR code reader[48]; and BtBeaconWrapper, which wraps communication with a Bluetooth beacon.

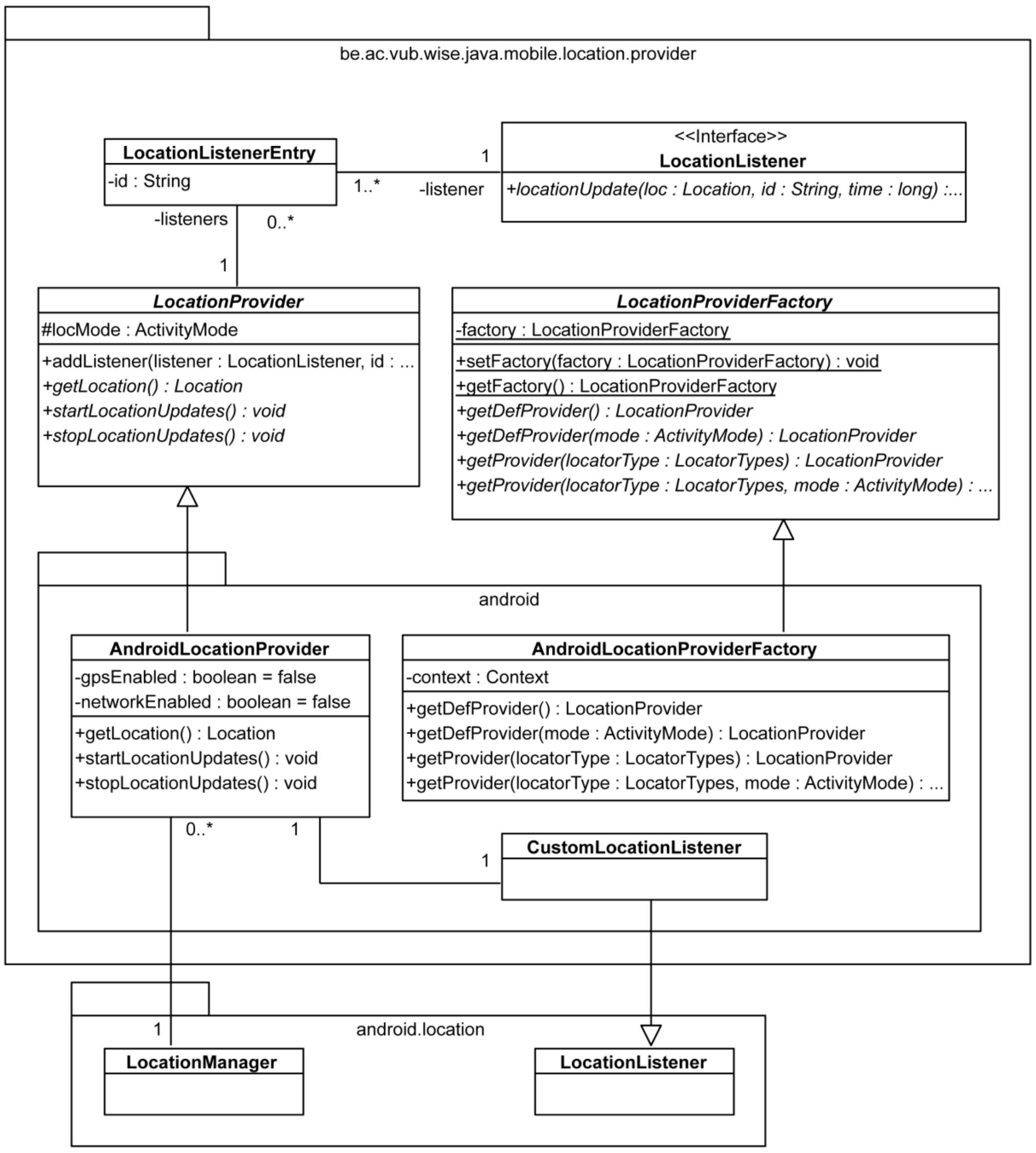

be.ac.vub.wise.java.mobile.location.provider
LocationListenerEntry
-id : String
<<Interface>>
LocationListener
+locationUpdate(loc : Location, id : String, time : long) :...
1
1..*
-listener
-listeners
0..*
1
LocationProvider
#locMode : ActivityMode
+addListener(listener : LocationListener, id : ...
+getLocation() : Location
+startLocationUpdates() : void
+stopLocationUpdates() : void
LocationProviderFactory
-factory : LocationProviderFactory
+setFactory(factory : LocationProviderFactory) : void
+getFactory() : LocationProviderFactory
+getDefProvider() : LocationProvider
+getDefProvider(mode : ActivityMode) : LocationProvider
+getProvider(locatorType : LocatorTypes) : LocationProvider
+getProvider(locatorType : LocatorTypes, mode : ActivityMode) : ...
android
AndroidLocationProvider
-gpsEnabled : boolean = false
-networkEnabled : boolean = false
+getLocation() : Location
+startLocationUpdates() : void
+stopLocationUpdates() : void
AndroidLocationProviderFactory
-context : Context
+getDefProvider() : LocationProvider
+getDefProvider(mode : ActivityMode) : LocationProvider
+getProvider(locatorType : LocatorTypes) : LocationProvider
+getProvider(locatorType : LocatorTypes, mode : ActivityMode) : ...
0..*
1
1
CustomLocationListener
android.location
1
LocationManager
LocationListener

Figure 3-6. *SCOUT Detection Layer*: LocationProvider package.

In Figure 3-6, we show the LocationProvider package. This package keeps classes supporting the positioning capabilities of mobile devices (e.g., GPS), and is thus useful for detection techniques relying on the positioning linking mechanism (e.g., indirect detection techniques). The LocationProvider class returns a Location object, which keeps a spatial shape representing the user

[48] https://code.google.com/p/zxing/ (access date: 13/06/2013)

(e.g., point) together with his absolute coordinates. For more information on Location and Position objects, we refer to Figure 3-12. LocationProvider supplies both push- and pull-based access to the user's location, whereby push-based access involves notifying the registered LocationListeners of the user's new location. LocationProviderFactory is an abstract factory for the LocationProvider class.

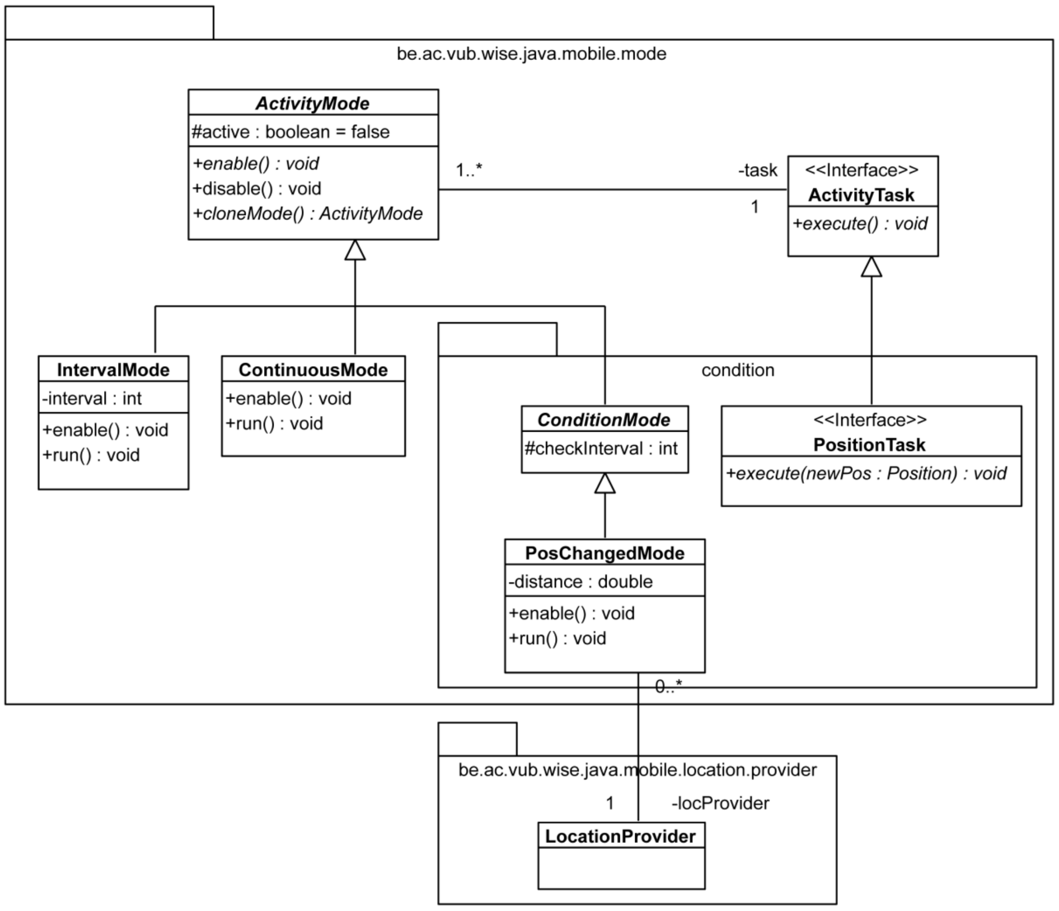


Figure 3-7. *SCOUT Detection Layer*: ActivityMode package.

Figure 3-7 shows the ActivityMode package. In order to automatically execute sensing actions at particular moments (e.g., RFID readings), the generic ActivityMode class is used. In general, an ActivityMode instance represents a certain task execution strategy, and is able to execute a given ActivityTask instance corresponding to the particular execution strategy. Subclasses include ContinuousMode, which stand for a continuous execution of tasks; IntervalMode, which denotes executing tasks after a given interval has passed; and ConditionMode, whereby tasks are executed each time an arbitrary condition is satisfied. PosChangedMode is a subclass of ConditionMode, and executes tasks each time the user's position has changed by a specified distance. This subclass relies on LocationProvider (see Figure 3-6) to obtain push-based access to the user's location. An ActivityMode can execute arbitrary tasks, represented as ActivityTasks; in our case, ActivityTask instances encapsulate sensor reading operations. It should be noted that certain sensing actions, such as QR code readings via the device camera, are initiated by the user and thus cannot be controlled by an ActivityMode strategy.

We observe that PosChangedMode is particularly useful in our mobile setting, as it allows automatically executing sensor readings (e.g., via RFID) to discover the user's new environment, each time he has moved a significant distance.

3.6.1.2 Detection techniques

A detection technique has several responsibilities. It needs to detect physical entities in the user's vicinity, and locate their associated online semantic data. Furthermore, a detection technique should return detection data, such as the detected entity's absolute position or maximum detection range, in order to allow the Spatial Layer to infer spatial relations (see section 3.3). Detection techniques follow the Observer design pattern, notifying other framework objects of detected entities. Below, we discuss the abstract detection technique classes, together with their implementing subclasses.

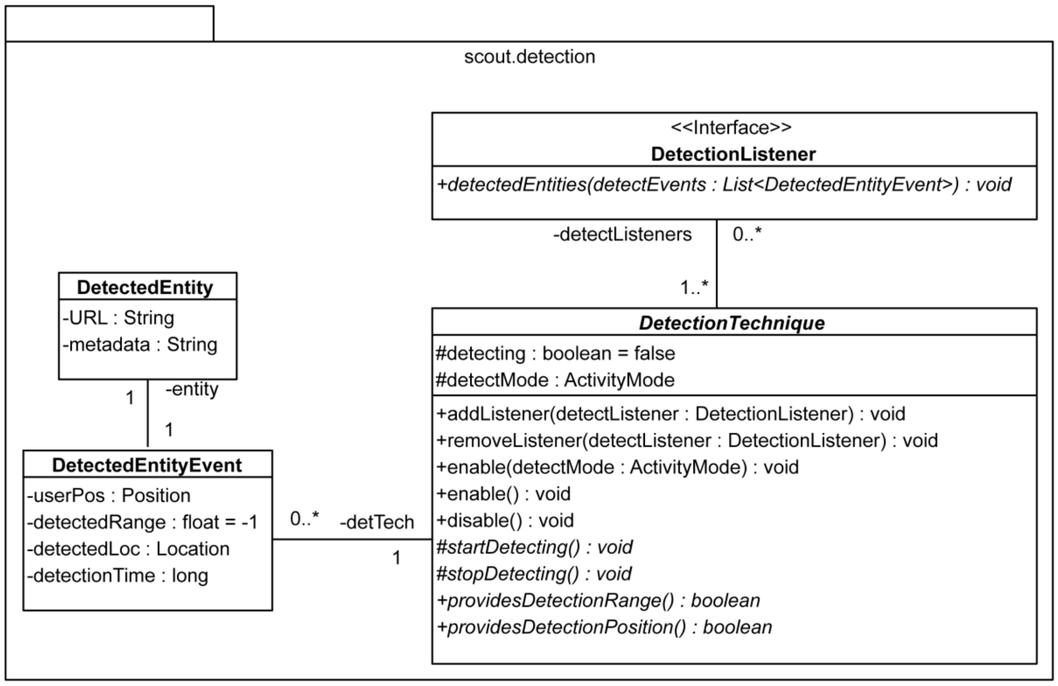

scout.detection
<<Interface>>
DetectionListener
+detectedEntities(detectEvents : List<DetectedEntityEvent>) : void
-detectListeners
0..*
1..*
DetectedEntity
-URL : String
-metadata : String
1
-entity
1
DetectionTechnique
#detecting : boolean = false
#detectMode : ActivityMode
+addListener(detectListener : DetectionListener) : void
+removeListener(detectListener : DetectionListener) : void
+enable(detectMode : ActivityMode) : void
+enable() : void
+disable() : void
#startDetecting() : void
#stopDetecting() : void
+providesDetectionRange() : boolean
+providesDetectionPosition() : boolean
DetectedEntityEvent
-userPos : Position
-detectedRange : float = -1
-detectedLoc : Location
-detectionTime : long
0..*
-detTech
1

Figure 3-8. *SCOUT Detection Layer:* DetectionTechnique package.

Figure 3-8 shows the DetectionTechnique package. Framework classes, which implement the DetectionListener interface, can register with a DetectionTechnique to be notified in case of entity detections. If relevant, a detection technique can be loaded with an ActivityMode (see Figure 3-7) to define when detection actions should be automatically performed (e.g., in case of RFID). This ActivityMode can then be passed to the TagReader / LocationProvider instances utilized by the DetectionTechnique subclass.

An individual detection is represented by a DetectedEntityEvent object, while the DetectedEntity class stands for detected entities themselves. In addition to a DetectedEntity object, a DetectedEntityEvent instance also keeps detection data, including the detection range and absolute location of the detected entity (if any), timestamp of the detection, and the related detection technique itself. A DetectedEntity object keeps the URL pointing to the associated online semantic data, while it can also keep descriptive metadata directly retrieved by the detection technique (e.g., read from a high-capacity RFID tag). In case this metadata encodes a spatial description of the detected entity, DetectionTechnique attempts to convert the metadata to objects representing the information (see Figure 3-12). Currently, the detection layer supports spatial descriptions represented using the GeoFeatures[49] and WGS84 [81] ontologies, as well as the Well-Known Text (WKT) format [82].

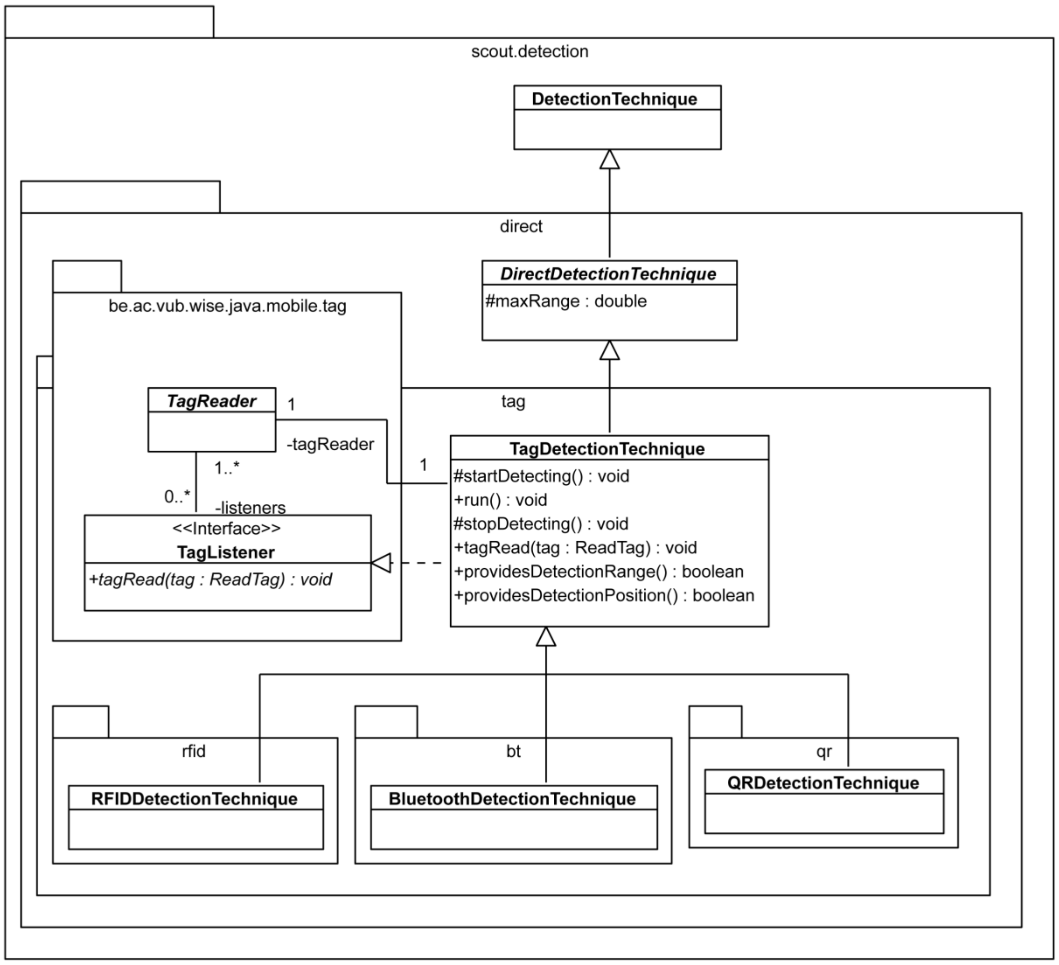


Figure 3-9. *SCOUT Detection Layer:* DirectDetectionTechnique package.

In Figure 3-9, we show the DirectDetectionTechnique package. A DirectDetectionTechnique directly detects physical entities using a particular tag sensing technology (e.g., RFID). The subclass

[49] http://www.mindswap.org/2004/geo/geoOntologies.shtml (access date: 08/03/2013)

TagDetectionTechnique represents detection techniques relying on tag-reading technology, as represented by TagReader (see Figure 3-5). The TagDetectionTechnique has three subclasses, based on the particular tagging technology; RFIDDetectionTechnique, QRDetectionTechnique and BluetoothDetectionTechnique. As detection data, each DirectDetectionTechnique subclass should minimally return the tag reader's maximum detection distance, while additional data can be provided as well, based on the capabilities of the utilized TagReader. The configured ActivityMode is passed on to the TagReader to realize the entity detection strategy, whereby the TagDetectionTechnique registers as a TagListener. To support the BluetoothDetectionTechnique, we implemented a simple BluetoothBeacon in Java (diagram not shown) that listens for incoming Bluetooth connections, as received from the BtBeaconWrapper (see Figure 3-5). This BluetoothBeacon is connected to a hardware Bluetooth receiver near a physical entity, and returns online semantic data locations as well as some metadata on the entity (in our case, encoding the entity's spatial data).

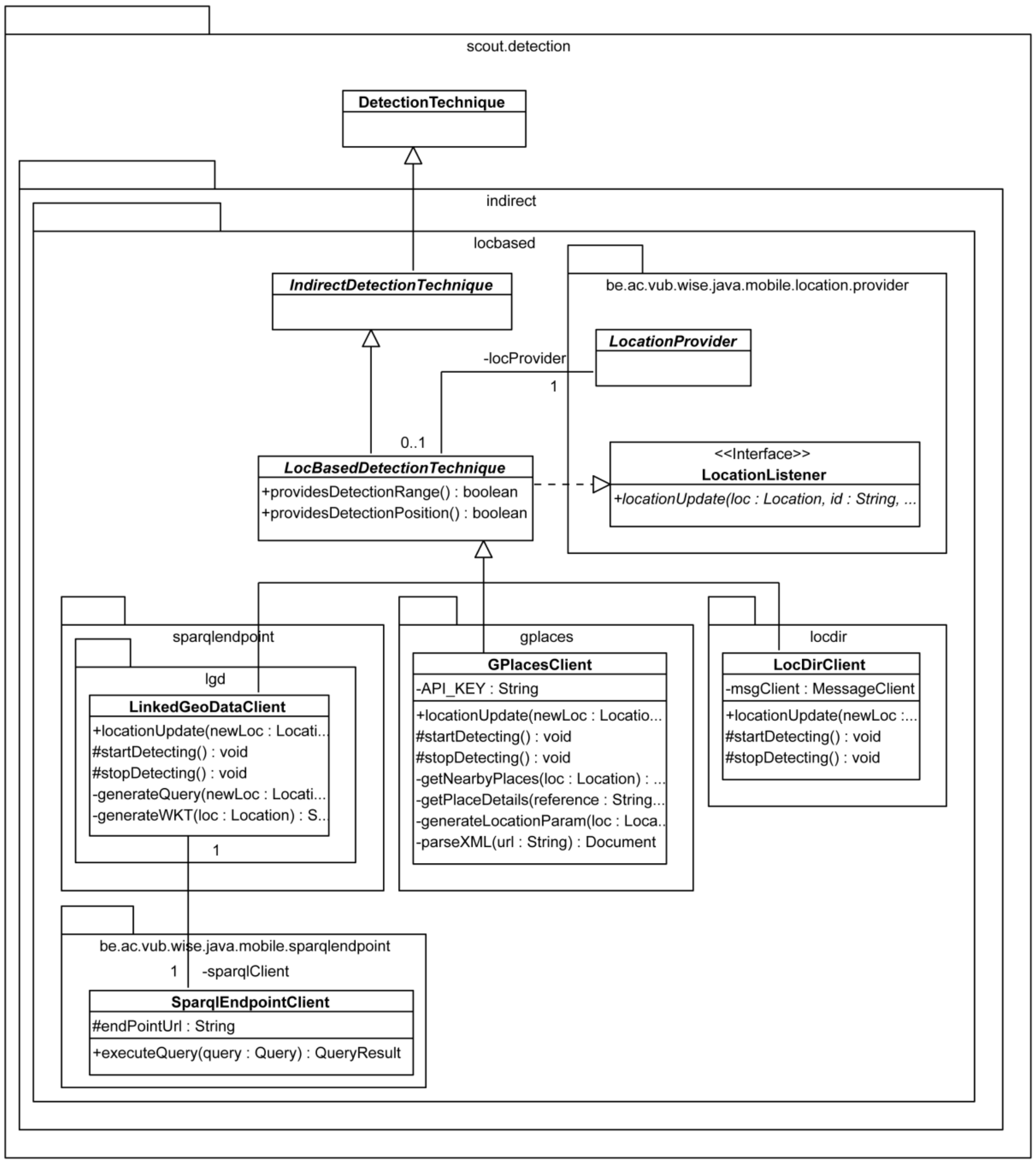

scout.detection
DetectionTechnique
indirect
locbased
IndirectDetectionTechnique
be.ac.vub.wise.java.mobile.location.provider
LocationProvider
-locProvider
1
0..1
<<Interface>>
LocationListener
+locationUpdate(loc : Location, id : String, ...
LocBasedDetectionTechnique
+providesDetectionRange() : boolean
+providesDetectionPosition() : boolean
sparqlendpoint
lgd
LinkedGeoDataClient
+locationUpdate(newLoc : Locati...
#startDetecting() : void
#stopDetecting() : void
-generateQuery(newLoc : Locati...
-generateWKT(loc : Location) : S...
1
gplaces
GPlacesClient
-API_KEY : String
+locationUpdate(newLoc : Locatio...
#startDetecting() : void
#stopDetecting() : void
-getNearbyPlaces(loc : Location) : ...
-getPlaceDetails(reference : String...
-generateLocationParam(loc : Loca...
-parseXML(url : String) : Document
locdir
LocDirClient
-msgClient : MessageClient
+locationUpdate(newLoc :...
#startDetecting() : void
#stopDetecting() : void
be.ac.vub.wise.java.mobile.sparqlendpoint
1
-sparqlClient
SparqlEndpointClient
#endPointUrl : String
+executeQuery(query : Query) : QueryResult

Figure 3-10. *SCOUT Detection Layer*: IndirectDetectionTechnique package.

Figure 3-10 shows the IndirectDetectionTechnique package. An indirect detection technique relies on an external service to perform detection. Specifically, the service is contacted with a piece of the mobile user's context (e.g., location), where the service responds with context-relevant physical entities (e.g., location-related) and references to their online metadata. Subclasses of IndirectDetectionTechnique differ based on the context required by the external service. Currently, the only supported subclass is the LocBasedDetectionTechnique, which contacts the service with the user's location obtained via a LocationProvider object (see Figure 3-6). Analogously to DirectDetectionTechnique, the LocBasedDetectionTechnique realizes the detection strategy by passing the configured ActivityMode to the LocationProvider and registering as a LocationListener.

LocBasedDetectionTechnique has three concrete subclasses, namely LinkedGeoDataClient, GPlacesClient and LocDirClient. The GPlacesClient leverages the Restful API of Google Places to obtain structured XML data on registered entities in the user's vicinity. This XML data includes a URL, potentially pointing to online semantic data describing the entity. In addition, the returned XML data specifies the geometry of the entity as well as its absolute coordinates. Based on the returned XML data, DetectedEntityEvent objects are populated and pushed to the registered DetectionListeners (see Figure 3-8). The LinkedGeoDataClient relies on the SparqlEndpointClient class to send a query to the LinkedGeoData online SPARQL query endpoint, parameterized with the user's current location. Using the returned RDF data, DetectedEntityEvent objects are likewise instantiated. Finally, the LocDirClient contacts an external server that deploys our LocationDirectory application (diagram not shown), which is built on top of a geographical database (PostgreSQL extended with PostGIS[50]). Given the user's current coordinates, the server returns data on registered entities in the user's vicinity, including URLs pointing to online semantic web data as well as associated coordinates and geometric shapes. We developed a simple JSP web application that allows users to easily add new physical entities to the database, by drawing rectangles representing their spatial shape on a Google Map and associating them with a URL.

3.6.2 Spatial Layer

This section reviews the implementation of the Spatial Layer. First, we show the SpatialManager class, and discuss how it communicates with classes from the Detection Layer to receive detection and location updates (section 3.6.2.1). Then, we show the Location package (which includes the SpatialShape class), and consider how approximate locations are represented (section 3.6.2.2). We present the SpatialEntityIndex package, which stands for the spatial index utilized to infer spatial information (section 3.6.2.3). Finally, we elaborate on the RectangleIndex package, which incorporates a rectangle-based implementation of the spatial index (using an R-tree data structure), and discuss how generic spatial shapes (e.g., circles) may be stored and retrieved using such an implementation.

[50] http://postgis.net/ (access date: 13/06/2013)

3.6.2.1 SpatialManager

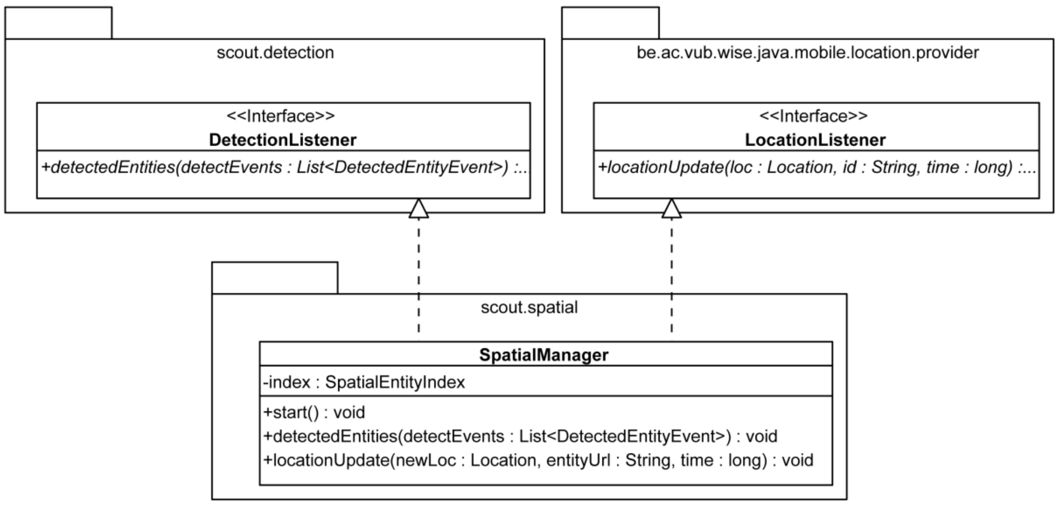

scout.detection
<<Interface>>
DetectionListener
+detectedEntities(detectEvents : List<DetectedEntityEvent>) :...
be.ac.vub.wise.java.mobile.location.provider
<<Interface>>
LocationListener
+locationUpdate(loc : Location, id : String, time : long) :...
scout.spatial
SpatialManager
-index : SpatialEntityIndex
+start() : void
+detectedEntities(detectEvents : List<DetectedEntityEvent>) : void
+locationUpdate(newLoc : Location, entityUrl : String, time : long) : void

Figure 3-11. *SCOUT Spatial Layer*: SpatialManager class.

Figure 3-11 shows the SpatialManager class, which is registered as a DetectionListener with the supported detection techniques. Whenever one of the detection techniques detects new physical entities, the SpatialManager object is notified, which starts the process for inferring spatial information (see section 3.3.1) that involves contacting the SpatialEntityIndex instance (see section 3.6.2.3). Similarly, the SpatialManager is registered as a LocationListener with the default LocationProvider. In case the user's location changes, the SpatialManager is notified, likewise causing the spatial information inferring process to ensue. The DetectionTechniques and LocationProvider are configured with specific ActivityModes (see Figure 3-7) to determine suitable detection and location-providing strategies. For instance, this enables the SpatialManager to be notified by a LocationProvider (see Figure 3-6) of the user's new location whenever it changes by a certain distance (e.g., 5m).

3.6.2.2 Locations and spatial shapes

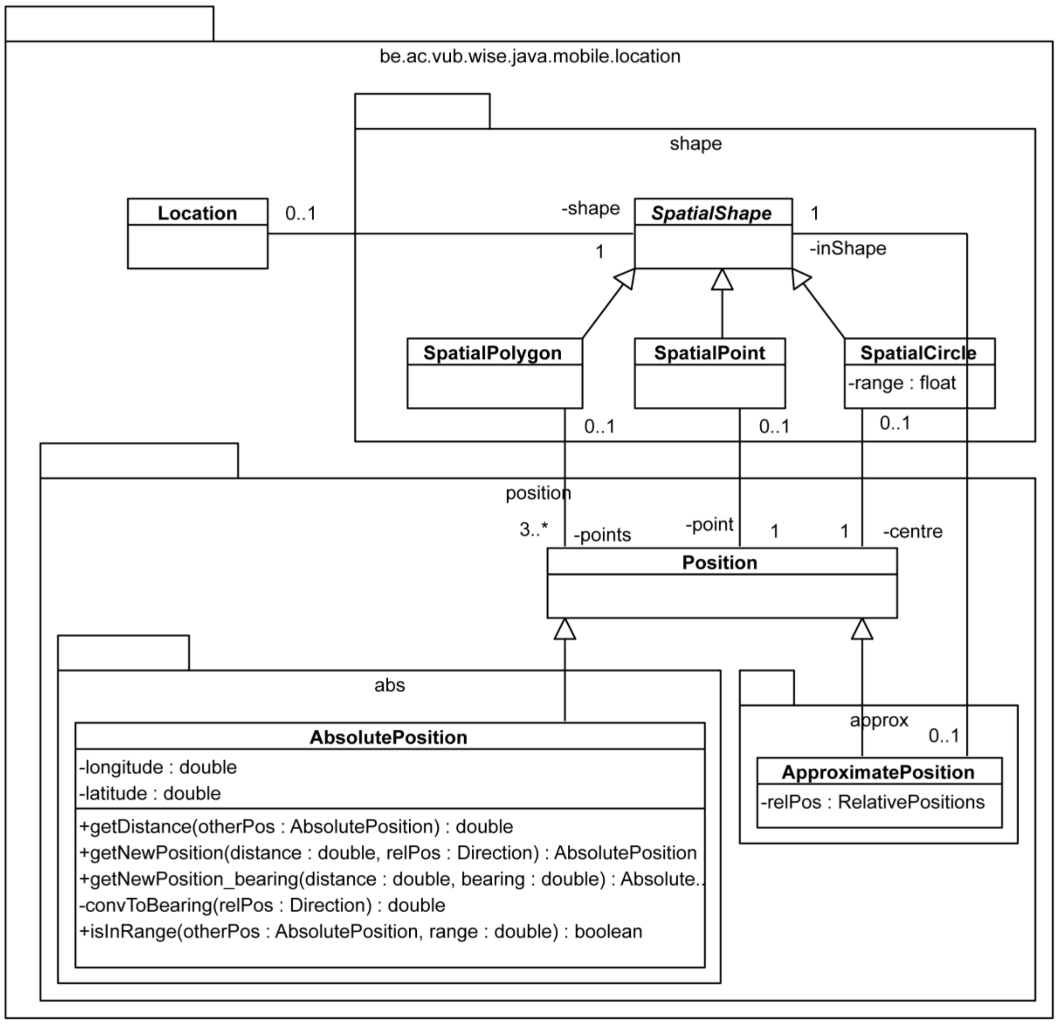

be.ac.vub.wise.java.mobile.location
shape
Location
0..1
-shape
1
SpatialShape
1
-inShape
SpatialPolygon
SpatialPoint
SpatialCircle
-range : float
0..1
0..1
0..1
position
3..*
-points
-point
1
1
-centre
Position
abs
AbsolutePosition
-longitude : double
-latitude : double
+getDistance(otherPos : AbsolutePosition) : double
+getNewPosition(distance : double, relPos : Direction) : AbsolutePosition
+getNewPosition_bearing(distance : double, bearing : double) : Absolute..
-convToBearing(relPos : Direction) : double
+isInRange(otherPos : AbsolutePosition, range : double) : boolean
approx
0..1
ApproximatePosition
-relPos : RelativePositions

Figure 3-12. *SCOUT Spatial Layer:* Location package, including SpatialShape.

In Figure 3-12, we show the package containing the Position, SpatialShape and Location classes. SpatialShape stores a geometric figure together with the positions of its vertex/vertices (e.g., polygon nodes, circle center). A Location keeps a physical entity's SpatialShape, and thus represents the entity's spatial presence in the world. As discussed in section 3.3.1, it is possible an entity's spatial shape is not accurately known. To represent absolute positions as well as approximate positions, the Position class has two subclasses: AbsolutePosition and ApproximatePosition. AbsolutePosition keeps the absolute longitude and latitude coordinates. ApproximatePosition keeps the SpatialShape in which the position is known to be located (e.g., circle with maximum detection range as radius), as well as the relative position in that shape (e.g., somewhere inside the circle or on the circle's edge) indicated by the RelativePositions enumeration. Currently, in case of approximation, an entity's location is represented by a SpatialPoint object, keeping as position a particular ApproximatePosition instance.

Below, we elaborate on the implementation of the spatial index.

3.6.2.3 Spatial entity index

The SpatialEntityIndex package is shown in Figure 3-13. SpatialEntityIndex stores the spatial shapes of detected physical entities as well as the user, and allows inferring spatial information. By treating the user as any other entity in the index, it can be easily checked whether the user and physical entities are spatially related, as well as physical entities themselves.

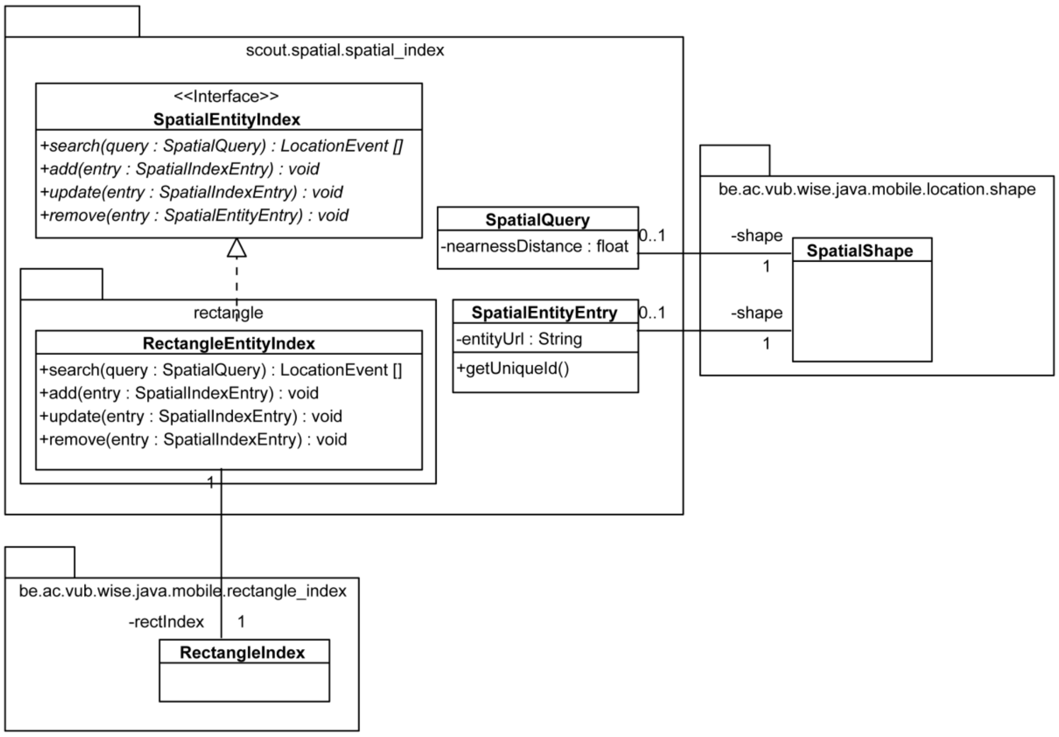

scout.spatial.spatial_index
<<Interface>>
SpatialEntityIndex
+search(query : SpatialQuery) : LocationEvent []
+add(entry : SpatialIndexEntry) : void
+update(entry : SpatialIndexEntry) : void
+remove(entry : SpatialEntityEntry) : void
SpatialQuery
-nearnessDistance : float
0..1
-shape
1
rectangle
RectangleEntityIndex
+search(query : SpatialQuery) : LocationEvent []
+add(entry : SpatialIndexEntry) : void
+update(entry : SpatialIndexEntry) : void
+remove(entry : SpatialIndexEntry) : void
SpatialEntityEntry
-entityUrl : String
+getUniqueId()
0..1
-shape
1
be.ac.vub.wise.java.mobile.location.shape
SpatialShape
1
be.ac.vub.wise.java.mobile.rectangle_index
-rectIndex
1
RectangleIndex

Figure 3-13. *SCOUT Spatial Layer:* SpatialEntityIndex package.

A SpatialEntityEntry object represents an entity to be stored in the spatial index, and keeps the entity's associated URL and spatial shape. A SpatialQuery represents a search into the SpatialEntityIndex. It requires the spatial shape of the search entity, as well as the maximum distance at which entities are still considered to be nearby (called nearness distance). In response, the spatial index returns indexed, spatially related entities in the form of LocationEvent subclass objects (subclasses not shown). Subclasses of SpatialEntityIndex leverage a specific spatial index implementation to store and retrieve spatial shapes. Currently, there is only one subclass called RectangleEntityIndex, which utilizes a rectangle-based implementation (represented by RectangleIndex). Below, we elaborate on this subclass.

3.6.2.4 Rectangle-based index

Figure 3-14 shows the RectangleIndex package, which stands for a rectangle-based spatial index. Currently, RectangleIndex has one concrete subclass, namely RTreeIndex, which relies on an R-tree data structure to index spatial shapes. Below, we elaborate on how the aforementioned SpatialEntityEntries, as well as SpatialQueries, are transformed to suit rectangle-based indices.

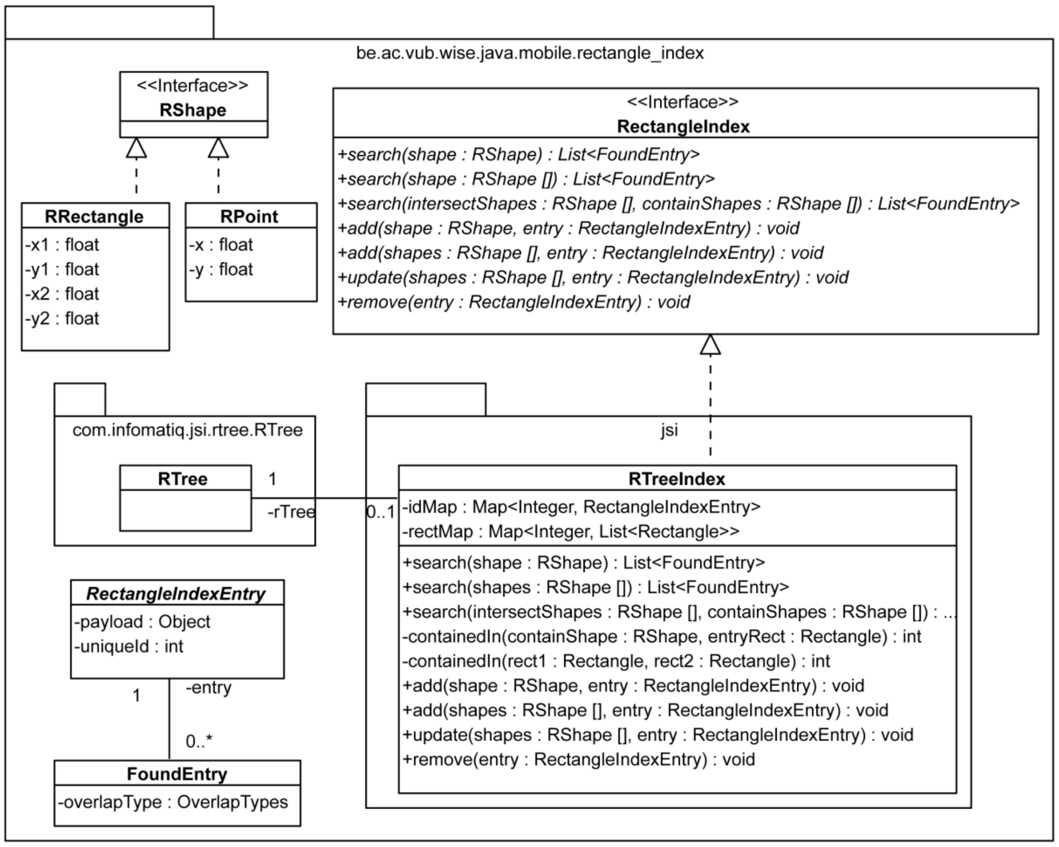

be.ac.vub.wise.java.mobile.rectangle_index
<<Interface>>
RShape
RRectangle
-x1 : float
-y1 : float
-x2 : float
-y2 : float
RPoint
-x : float
-y : float
<<Interface>>
RectangleIndex
+search(shape : RShape) : List<FoundEntry>
+search(shape : RShape []) : List<FoundEntry>
+search(intersectShapes : RShape [], containShapes : RShape []) : List<FoundEntry>
+add(shape : RShape, entry : RectangleIndexEntry) : void
+add(shapes : RShape [], entry : RectangleIndexEntry) : void
+update(shapes : RShape [], entry : RectangleIndexEntry) : void
+remove(entry : RectangleIndexEntry) : void
com.infomatiq.jsi.rtree.RTree
RTree
1
-rTree
0..1
jsi
RTreeIndex
-idMap : Map<Integer, RectangleIndexEntry>
-rectMap : Map<Integer, List<Rectangle>>
+search(shape : RShape) : List<FoundEntry>
+search(shapes : RShape []) : List<FoundEntry>
+search(intersectShapes : RShape [], containShapes : RShape []) : ...
-containedIn(containShape : RShape, entryRect : Rectangle) : int
-containedIn(rect1 : Rectangle, rect2 : Rectangle) : int
+add(shape : RShape, entry : RectangleIndexEntry) : void
+add(shapes : RShape [], entry : RectangleIndexEntry) : void
+update(shapes : RShape [], entry : RectangleIndexEntry) : void
+remove(entry : RectangleIndexEntry) : void
RectangleIndexEntry
-payload : Object
-uniqueId : int
1
-entry
0..*
FoundEntry
-overlapType : OverlapTypes

Figure 3-14. *SCOUT Spatial Layer*: RectangleIndex package.

The generic spatial shapes first need to be converted to rectangular shapes. In case absolute positions are available (see Figure 3-12), spatial circles and polygons are converted to their smallest encompassing rectangle, while a spatial point is converted to a rectangle point[51]. As mentioned in section 3.6.2.2, an entity's approximated shape is represented by a spatial point with an approximate position, which is somewhere inside a specific circle (case 1; in case only the maximum detection range is known) or on the edge of the circle (case 2; in case detection distance is known). The goal is to convert these approximate positions to the smallest rectangles encompassing *all potential positions*. In case 1, the associated circle will be converted to the smallest encompassing rectangle. In case 2, rectangles will be created to encompass the circle's edge, narrowing down the potential

[51] I.e., a rectangle with left-bottom coordinate x1, y1 and right-top coordinate x2, y2 where x1=x2 and y1=y2.

positions. This is illustrated in Figure 3-15, where 4 rectangles[52] are created to capture the circle's edge.

Comparable to SpatialEntityEntry, the RectangleIndexEntry class represents entries stored in a RectangleIndex instance. After generating the rectangles, a new RectangleIndexEntry object is created with the original SpatialEntityEntry as payload. This rectangle index entry is stored by the RectangleIndex, whereby it will be indexed on its generated rectangles.

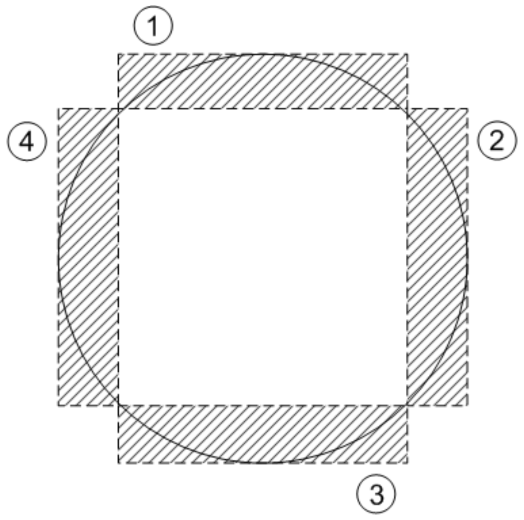


Figure 3-15. *SCOUT Spatial Layer:* An approximate position (case 2) converted to a set of rectangles.

A spatial query involves the search entity's spatial shape as well as the configured nearness distance. Typically, a rectangle-based index implementation, at the very least, supports checking whether a given rectangle overlaps with any of the indexed rectangles (e.g., this was the case for the R-tree implementations we investigated). Consequently, the search spatial shape is first converted to a set of rectangles, following the process mentioned above. Then, these rectangles are expanded with the nearness distance, by adding the distance to each of the resulting rectangles' width and height. This set of expanded rectangles now encompasses the spatial area considered to be nearby. In case these expanded rectangles intersect with any of the indexed rectangles, these are considered to be nearby. In case the original (unexpanded) search rectangles are completely contained by an indexed rectangle, or vice versa, we may infer containment. This is illustrated in Figure 3-16, which shows the user (search rectangle) inside a building (indexed rectangle) and nearby an object A (indexed rectangle). A rectangle-based search specifies both a set of expanded rectangles to check for nearness, as well as the set of original rectangles to check for containment. In response to a search, a list of FoundEntry objects is returned, each comprising the found RectangleIndexEntry and the type of overlap (i.e., intersect or contained in / by).

It should be noted that, in case the conversion of an approximate position leads to multiple rectangles (see above), overlapping with one of these rectangles is sufficient to infer spatial relations

[52] Technically, an arbitrary number of rectangles could be created to more precisely encompass the circle edge.

with the associated entity. Future work includes determining and returning the degree of uncertainty for each inferred spatial relation, depending on whether the search or indexed rectangles were approximated.

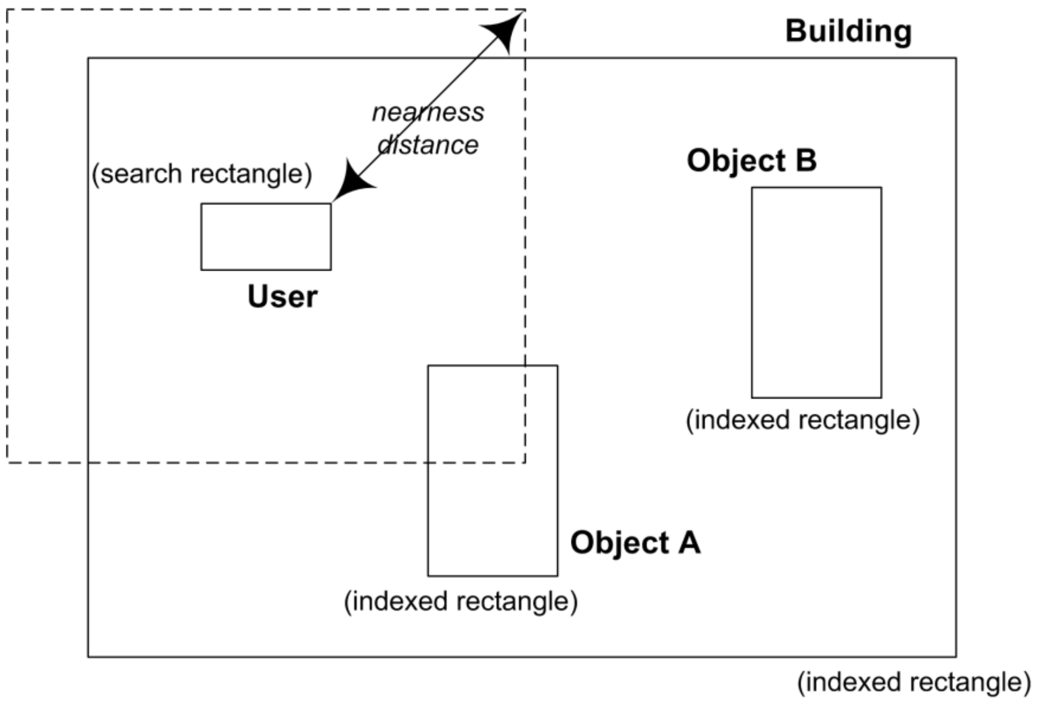


Figure 3-16. *SCOUT Spatial Layer:* An example search into a rectangle-based index.

The currently supported implementation of RectangleIndex is RTreeIndex (see Figure 3-14), which relies on an existing R-Tree library[53]. An R-Tree is a balanced tree structure much like a B+-tree, whereby the indexed rectangles are kept in the leaf nodes [83]. Each tree node has an associated bounding box, corresponding to the smallest rectangle still encompassing the bounding boxes of its child nodes or the rectangles stored by its leaf nodes. An R-Tree can be used to check whether a given search rectangle intersects with any of the indexed rectangles. For each of the nodes at the current tree level, it is checked whether their associated bounding boxes overlap with the search rectangle. If so, the tree is traversed down along that node to the next level, until the leaf nodes are reached. Since the bounding boxes may potentially overlap, it is possible that multiple paths need to be followed [83].

To cope with the limitations of the employed library, a number of extra steps had to be taken, which we shortly summarize here. In response to a spatial search, the RTree implementation is contacted to locate nearby rectangles as well as contained / containing rectangles. However, since the RTree does not directly support checking for containment, an additional containment check needs to occur after the nearby rectangles are returned. Secondly, the RTree is only able to associate integer payloads to

[53] http://jsi.sourceforge.net/ (access date: 13/06/2013)

the indexed rectangles. Therefore, in order to link indexed rectangles to their RectangleIndexEntry instances, an extra map (called idMap) needs to be kept, connecting the integer payloads to corresponding RectangleIndexEntry objects. Finally, in order to remove and update index entries, the RTree requires the integer payload as well as its associated rectangles. This necessitates a second map, linking the integer payloads to their corresponding indexed rectangles (rectMap).

3.6.3 Environment Layer

This section describes the Environment Layer implementation. We start by reviewing the implementation of the User Model, focusing on the mobile configuration tool (section 3.6.3.1). Then, we summarize the Environment Model implementation, discussing the Environment Model manager component and the Environment Notification service, whereby the latter supplies push-based access to the model (section 3.6.3.2).

3.6.3.1 User Model

The mobile User Model configuration tool was implemented using HTML 4, Javascript and CSS. In order to mimic the native Android look-and-feel, we employed the jQuery Mobile[54] framework. By using web technology, the tool can be easily integrated with web-based approaches such as client-side web augmentation methods, for instance COIN [29] (see Chapter 6, section 6.2) or Sticklets [84]. In order to enable communication between the web-based configuration tool and SCOUT, we implemented a Restful API in the Environment Layer. This API allows local web applications to query and update the Environment and User Model in a uniform way over the HTTP protocol, via the loopback interface (i.e., localhost).

Our web-based configuration tool sends HTTP requests via AJAX, which enables the tool to dynamically retrieve new User Model data when populating the form during navigation, and perform updates whenever the user selects the Save button. To bypass the AJAX cross-domain security restriction, we rely on a GreaseMonkey user script. Therefore, users need to use Mobile Firefox to run the mobile configuration tool.

3.6.3.2 Environment Model

Below, we elaborate on the Environment Model manager component. Then, we detail how push-based access to the Environment Model is realized, via the Environment Notification Service.

[54] http://jquerymobile.com/ (access date: 14/06/2013)

EnvironmentModelManager

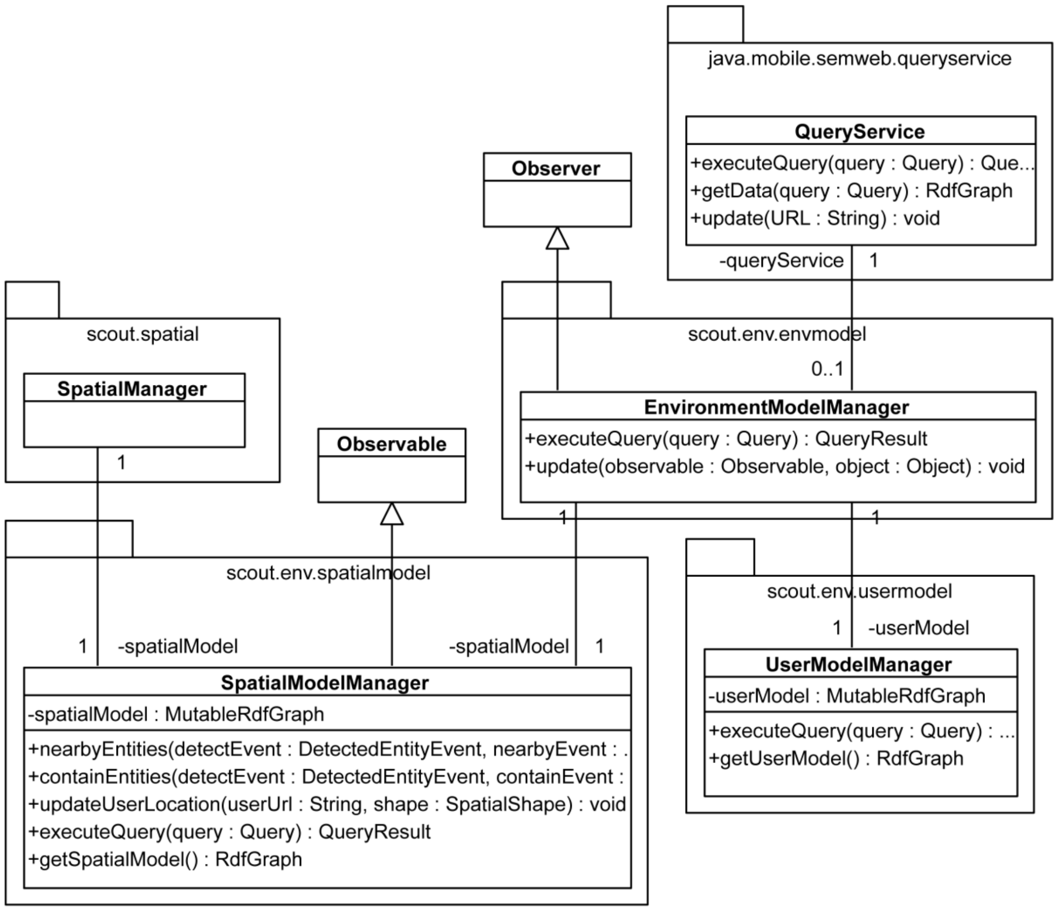

java.mobile.semweb.queryservice
QueryService
+executeQuery(query : Query) : Que...
+getData(query : Query) : RdfGraph
+update(URL : String) : void
-queryService 1
Observer
scout.env.envmodel
0..1
EnvironmentModelManager
+executeQuery(query : Query) : QueryResult
+update(observable : Observable, object : Object) : void
1
1
scout.spatial
SpatialManager
1
Observable
scout.env.spatialmodel
1 -spatialModel
-spatialModel 1
SpatialModelManager
-spatialModel : MutableRdfGraph
+nearbyEntities(detectEvent : DetectedEntityEvent, nearbyEvent : .
+containEntities(detectEvent : DetectedEntityEvent, containEvent :
+updateUserLocation(userUrl : String, shape : SpatialShape) : void
+executeQuery(query : Query) : QueryResult
+getSpatialModel() : RdfGraph
scout.env.usermodel
1 -userModel
UserModelManager
-userModel : MutableRdfGraph
+executeQuery(query : Query) : ...
+getUserModel() : RdfGraph

Figure 3-17. *SCOUT Environment Layer:* EnvironmentModelManager package.

Figure 3-17 shows the EnvironmentModelManager class, together with related classes. The SpatialModelManager class supports the Observer pattern, whereby registered observers are notified of the creation and invalidation of spatial relations. EnvironmentModelManager is registered as an observer with SpatialModelManager; for each new spatially related entity, the EnvironmentModelManager passes its URI resource as an online data source location to the query service (update method). This way, EnvironmentModelManager outlines the relevant online semantic dataset to the mobile query service (see section 3.4.3). Queries fired on the EnvironmentModelManager (executeQuery method) are executed on the User Model, Spatial Model and the online semantic dataset. The User Model and the Spatial Model are obtained by contacting their corresponding manager components (SpatialModelManager and UserModelManager classes). Furthermore, the mobile query service (QueryService class) is contacted to retrieve the semantic data selection related to the given query (getData method). The collected data is then assembled into an RDF graph and queried, and the results returned.

Push- and pull-based access to the Environment Model is respectively provided by the EnvironmentNotificationService and EnvironmentQueryService, whereby the latter simply delegates query execution to the EnvironmentModelManager. We discuss the EnvironmentNotificationService in more detail below.

Environment Notification Service

Mobile applications can obtain push-based query access to the Environment Model by registering queries with the EnvironmentNotificationService. These registered queries are re-evaluated whenever the contents of the Environment Model change; in case the query results differ from the previous results, the mobile application is notified. Figure 3-18 shows the diagram of the EnvironmentNotificationService package.

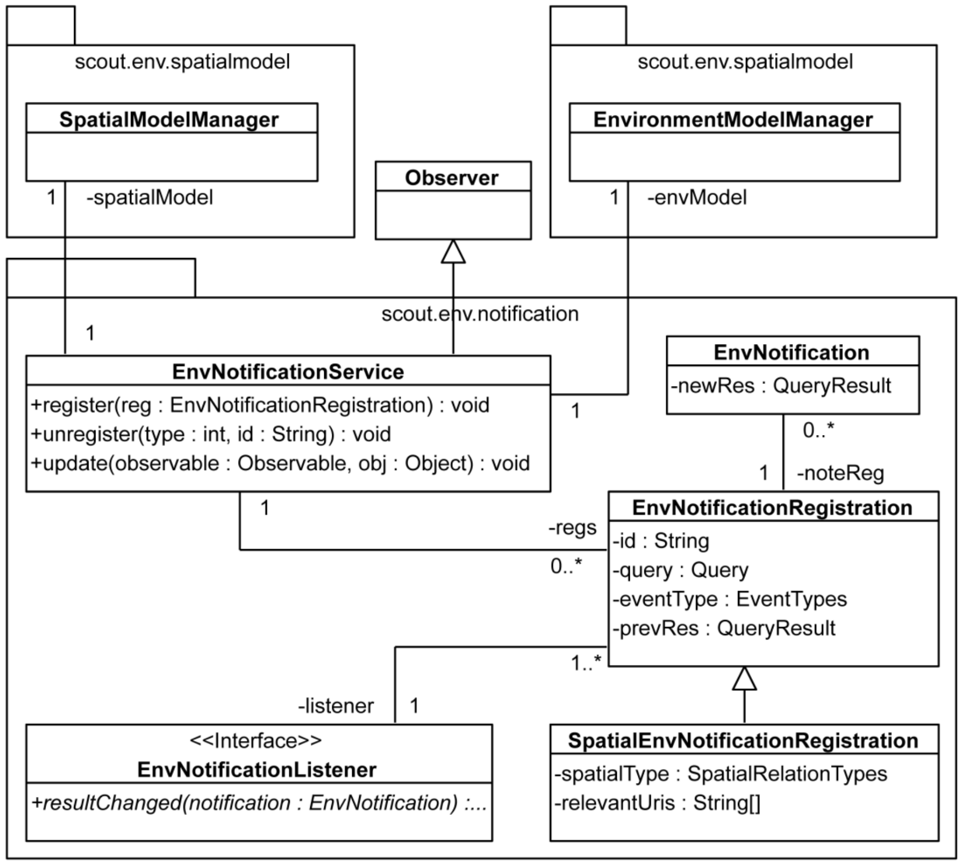


Figure 3-18. *SCOUT Environment Layer:* EnvironmentNotificationService package.

As the mobile user is walking around, the Environment Model is continuously extended, resulting from the detection of new physical entities and subsequent creation / invalidation of spatial relations. The EnvironmentNotificationService registers as an observer with the SpatialModelManager (see Figure 3-17), to be notified in case of such relation creations / invalidations. Mobile applications need to implement the EnvNotificationListener interface, while query registrations are represented by EnvNotificationRegistration instances. For optimization purposes, during registration, mobile applications can narrow down the times at which the query needs to be re-evaluated. This is done by specifying the type of event causing the change (EventTypes enumeration). In case of a spatial relation event, the type of spatial relation causing the change (i.e., current/previous nearness or containment; SpatialRelationTypes enumeration), together with the URIs identifying the physical entities participating in the spatial relation (relevantUris attribute), can be supplied as well. For instance, a mobile application may only be interested in performing a particular query in case new physical entities become nearby (spatial relation type) the user (participating entity URI).

When notified of new or invalidated spatial relations by the SpatialModelManager, the EnvNotificationService executes the registered queries on the EnvironmentModelmanager, taking into account the registration preferences. In case of new query results, the registered EnvNotificationListeners are notified via an EnvNotification instance.

3.6.4 Application Layer

This section illustrates how mobile applications, situated in the Application Layer, may utilize the SCOUT context services in their implementation. In particular, we detail how three developed applications access the SCOUT context services API. In Chapter 6, we discuss each of these mobile applications in more detail.

3.6.4.1 Person Matcher

The Person Matcher, which pro-actively identifies interesting people in the vicinity, registers a SPARQL query with the Environment Notification Service. This query returns nearby persons (type foaf:Person) currently nearby (sm:isNearby) the user (type um:User), together with their online FOAF profile location (indicated via rdfs:seeAlso or foaf:PersonalProfileDocument) (namespaces omitted for brevity):

```
SELECT ?person ?foaf_profile
WHERE {
    ?person rdf:type foaf:Person .
    ?stat rdf:subject ?person ;
          rdf:predicate sm:isNearby ;
          rdf:object ?user .
    ?user rdf:type um:User .
    {
         ?person rdfs:seeAlso ?foaf_profile .
    } UNION {
        ?foaf_profile rdf:type foaf:PersonalProfileDocument .
        ?foaf_profile foaf:maker ?person .
    }
 }
```

Code 3-1. *SCOUT Application Layer – Person Matcher*: Query registered with the Environment Notification Service.

In more detail, the mobile app instantiates a SpatialEnvNotificationRegistration object (see Figure 3-18), passing the query (see Code 3-1) together with a suitable configuration. In this case, the configuration states that the query should only be checked in case two physical entities are found to be nearby each other (SpatialRelationTypes enumeration), while one of these entities should be the user (indicated via the relevantUris array). After registering the registration object with EnvNotificationService (register method), the query will be re-evaluated each time a spatial relation is created adhering to the given conditions. A notification will be sent whenever the new query results differ from the previous results; in other words, when a new person becomes nearby the user.

3.6.4.2 COIN

The client-side COIN web augmentation approach injects context-aware features into existing websites. In a nutshell, the approach involves matching webpage content semantics to the mobile user's context, with the goal of locating context-relevant page content. Afterwards, suitable features are injected into the identified content.

In order to retrieve relevant context for matching, a query is executed on the Environment Model via the Environment Query Service. Below, we show a query that selects shops (type sumo:RetailShop) that are nearby (sm:isNearby) the user (type um:User) and the unique identifiers (gr:hasMPN) of their sold products (region:sells) (namespaces omitted for brevity):

```
SELECT ?shop ?shop_product_id
WHERE {
      ?user rdf:type um:User .
      ?stat rdf:subject ?user ;
            rdf:predicate sm:isNearby ;
            rdf:object ?shop .
      ?shop rdf:type sumo:RetailShop ;
            region:sells ?shop_product .
      ?shop_product gr:hasMPN ?shop_product_id .
}
```

Code 3-2. *SCOUT Application Layer – COIN*: Query passed to both context services.

Based on the returned context, the matching process locates page elements corresponding to products sold by nearby shops. In the final step, suitable features are injected; in this example, the aforementioned products are annotated on the page.

The query is also registered with the Environment Notification Service. In this case, the query needs to be evaluated whenever an entity becomes nearby, or is no longer nearby, the user. For instance, this allows injecting and altering/removing features that annotate sold products, depending on whether the shop becomes nearby or no longer nearby the user. For this purpose, a SpatialEnvNotificationRegistration object is instantiated, given the above query and suitable settings.

3.6.4.3 AdaptIO

The AdaptIO approach adapts the obtrusiveness of mobile service interactions, in order to suit the user's current situation. The mobile user is hereby put in charge of defining obtrusive situations (e.g., in a meeting, or in company of others) via a mobile user interface. Importantly, this interface allows "capturing" the mobile user's current situation, whereby current context data is re-used to define an obtrusive situation. This capturing is done by executing a query on the Environment Model via the Environment Query Service, which requests all physical entities that are currently spatially related (?spatialPred variable) to the user (type foaf:Person), together with their type (rdf:type) (namespaces omitted for brevity):

```
SELECT ?entity ?type
WHERE {
   ?user rdf:type um:User .
   ?stat rdf:subject ?user ;
         rdf:predicate ?spatialPred ;
         rdf:object ?entity .
   ?entity rdf:type ?type .
   FILTER (sameTerm(?spatialPred, sm:isNearby) ||
         (sameTerm(?spatialPred, sm:containedBy))
}
```

Code 3-3. *SCOUT Application Layer – AdaptIO*: Query executed via the Environment Query Service.

After an obtrusive situation has been defined by the mobile user, it is converted into a logical rule and passed to the Environment Layer, which was outfitted with an existing reasoning engine for this purpose[55]. Each time the Environment Model is changed or extended, this reasoning engine re-evaluates the registered rules. Any potentially inferred facts (e.g., the user's current situation) are hereby added to a separate concrete model, which is part of the Environment Model. In order to be notified whenever the user's current situation changes in the Environment Model, the AdaptIO system registers a query with the Environment Notification Service:

```
SELECT ?situation
WHERE {
   ?user rdf:type um:User ;
      usm:currentSituation ?situation .
}
```

Code 3-4. *SCOUT Application Layer – AdaptIO*: Query registered with the Environment Notification Service.

In particular, an EnvNotificationRegistration object (see Figure 3-18) is instantiated, given the above query and a configuration specifying that the query should be re-evaluated whenever new environment facts have been inferred (EventTypes enumeration). For more information on the AdaptIO system and related SCOUT extensions, as well as the two other mobile applications, we refer to Chapter 6.

[55] We utilized the generic Androjena rule engine (http://code.google.com/p/androjena/)

3.7 Summary

In this chapter, we presented SCOUT as a mobile, client-side context-provisioning framework. The framework consists of a layered architecture, where each task is neatly encapsulated in a distinct layer. This allows different technologies and mechanisms to be plugged in at each level, without requiring changes to the other layers.

The Detection Layer is responsible for dynamically detecting physical entities in the user's surroundings, and locating their associated online semantic data sources. For this purpose, the layer comprises a set of detection techniques, which leverage linking mechanisms (e.g., tagging, positioning) connecting the physical to the virtual world. Direct detection techniques employ a hardware component (e.g., RFID reader) to directly detect physical entities and identify their associated online data (e.g., by reading a URL from an RFID tag). Indirect detection techniques rely on external services for detection, whereby some contextual data (e.g., GPS position) is passed on to the service (e.g., online semantic dataset such as LinkedGeoData). This service then returns online locations of metadata describing context-relevant physical entities (e.g., in the user's vicinity). Suitable detection strategies can be defined in order to determine when detections should take place (e.g., after an interval, or in case of position changes). Multiple detection techniques can be utilized in parallel and interchangeably, allowing SCOUT to be deployed across heterogeneous environments.

The Spatial Layer abstracts low-level detection and location data, obtained from the Detection Layer, into high-level spatial information. Such spatial data indicates whether the user is nearby a physical entity or inside a place, or whether physical entities are nearby/ inside each other. By supplying mobile applications with such spatial data, they can easily obtain knowledge on the user's surroundings while still abstracting from low-level details (e.g., detection ranges). A flexible mechanism is utilized to keep the spatial information up-to-date, based on a spatial index. The spatial index stores the spatial shapes of detected physical entities, as well as the user, whereby a spatial shape denotes the entity's geometric shape and absolute location. Such spatial shapes are determined based on detection and location data; in case this data is inaccurate, the detected entity's spatial shape will be approximated. The following process is followed after a new physical entity is detected, or the user's location has changed. Firstly, the entity's spatial shape is approximated (if necessary), after which the spatial index is contacted to retrieve spatially related entities. Afterwards, two entity lists are created; 1/ keeping physical entities spatially related to the particular entity (i.e., detected entity or user), and 2/ keeping entities that are no longer spatially related. These two lists are then communicated to the Environment Layer, together with the entity in question. The related spatial shapes, which may be potentially relevant to mobile applications, are also passed to the layer. Currently, the spatial index is implemented as an R-tree.

The Environment Layer provides mobile applications with an abstract, integrated view of the user, his environment and the physical entities in it, called the Environment Model. Two concrete models supply essential data for this view. The *User Model* stores information useful for personalization (e.g., device properties, user characteristics and preferences), and can be updated by the mobile user via the mobile configuration tool. The *Spatial Model* keeps spatial relations between physical entities and the user, based on the spatial information from the Spatial Layer. The Environment Model comprises the User Model, Spatial Model, and detected online semantic data sources, thus supplying integrated access to the user's context and surroundings. In order to transparently query the detected semantic dataset, consisting of large amounts of small online sources, we utilize the mobile query service, which will be detailed in Chapter 4.

Mobile applications can access the Environment Model in a push- and pull-based way, respectively via the Environment Notification Service and Environment Query Service. We presented a short summary of mobile applications currently built on top of SCOUT, focusing on how the SCOUT context services are employed to realize the mobile applications' functionality.

Chapter 4

A Mobile Query Service for Online Semantic Data Sources

In the previous chapter, we presented SCOUT as a mobile, client-side context-provisioning framework. We detailed its layered architecture and elaborated on each of the layers, describing their components, data structures and processes. Based on detection data and spatial info from the below layers, the Environment Layer constructs and maintains the Environment Model, which comprises the mobile user's profile and environment context. An essential part of the Environment Model constitutes the online semantic dataset, consisting of small online sources associated with detected physical entities. In this chapter, we present a novel, general-purpose mobile query service utilized by SCOUT to transparently query this online dataset. Due to this general-purpose nature, other systems may also act as clients to the query service, in order to gain integrated query access to an online dataset comprising small semantic sources.

First, we motivate the necessity for such a general-purpose query service, targeted towards large amounts of small online sources (section 4.1). Subsequently, the chapter discusses the challenges and requirements that arise when querying such online semantic data on mobile devices (section 4.2). We then present an overview of the query service, discussing its components and phases (section 4.3). Afterwards, we elaborate on each major component (sections 4.4 and 4.5), and discuss configurable support for the Semantic Web's Open World Assumption (section 4.6). The chapter also reviews the core implementation of the query service (section 4.7). Throughout the chapter, we indicate how the identified challenges and requirements are met. Finally, we present a summary of this chapter (section 4.8).

4.1 Motivation

In order to access online semantic data, many mobile applications currently rely on online query endpoints, such as DBPedia Mobile[56], mSpace Mobile [65] and MobiSem [12]. Mobile applications can send queries to an online query endpoint and directly receive useful results, thus achieving easy and efficient query access. Such query endpoints can be set up using technologies such as OpenLink Virtuoso[57] and Sesame Server[58], while local RDF libraries (e.g., Androjena[59]) exist that provide support for accessing online query endpoints. Examples of currently online, freely accessible query endpoints are LinkedGeoData[60], which is a semantic version of OpenStreetMap, and DBPedia[61], which keeps Wikipedia information in semantic format.

Regrettably, the tools and technologies are currently lacking to access other parts of the Semantic Web, which are not available behind query endpoints. This includes online RDF files (e.g., FOAF profiles) and semantically annotated websites (e.g., using RDFa[62]). As structured RDF data can be directly extracted from many types of semantic annotations (e.g., RDFa, microformats [3]), these kinds of websites are online semantic sources in their own right. This Semantic Web segment is far from negligible. Sindice[63], a Semantic Web search engine, indexes around 708 million online RDF files. At the same time, the Web Data Commons[64] initiative found that close to 13% of all crawled webpages contain semantic annotations. With major search engines (Google, Bing) exploiting semantic annotations to improve search results, this amount can be expected to increase.

Our mobile, client-side query service enables mobile applications to transparently access this currently untapped Semantic Web segment, consisting of large amounts of relatively small online sources. In doing so, we unlock this online semantic dataset for consumption by mobile clients. Below, we elaborate on the challenges and requirements for this mobile query service.

[56] http://wiki.dbpedia.org/DBpediaMobile (access date: 24/05/2013)
[57] http://virtuoso.openlinksw.com/ (access date: 24/05/2013)
[58] http://www.openrdf.org/doc/sesame/users/ch01.html#d0e98 (access date: 24/05/2013)
[59] http://code.google.com/p/androjena/ (access date: 24/05/2013)
[60] http://linkedgeodata.org/ (access date: 24/05/2013)
[61] http://dbpedia.org/ (access date: 24/05/2013)
[62] http://www.w3.org/TR/xhtml-rdfa-primer/ (access date: 24/05/2013)
[63] http://sindice.com/ (access date: 24/05/2013)
[64] http://webdatacommons.org/2012-02/stats/stats.html (access date: 24/05/2013)

4.2 Challenges and requirements

Our mobile query service provides transparent, integrated query access to a currently inaccessible segment of the Semantic Web, consisting of small online semantic sources. In general, mobile applications require access to a specific selection of Semantic Web data. For instance, the SCOUT context-provisioning framework (see Chapter 3) needs to access online sources describing the user's surroundings, while recommender systems require semantic descriptions of items to be recommended [85]. To accommodate this, our query service enables mobile applications to delineate their relevant Semantic Web subset (e.g., as in [15]), as well as dynamically expand this subset when new useful information is discovered (e.g., for SCOUT, when new physical entities are detected).

In this particular querying scenario, a number of issues, and corresponding challenges, arise. We discuss these below.

Large query dataset: Even a delineated subset of the Semantic Web is likely to contain a large number of sources. However, querying the entire relevant dataset would not be feasible (e.g., the entire query dataset should be kept in-memory to enable fast querying). Therefore, a first challenge is to **reduce the total query dataset**, making it feasible for querying. This challenge of reducing the amount of query data is also tackled in other related approaches (see Chapter 2, sections 2.2.6 and 0) [12, 62, 67, 68].

Data captured in online files: We note that query-relevant data items are captured in online sources, which need to be fully downloaded and usually contain other (irrelevant) data as well. This makes the data-retrieval overhead relatively large. Furthermore, connectivity loss, not uncommon in mobile scenarios, results in the online dataset becoming inaccessible. As a result, a second challenge is to **minimize the number of source downloads**, to reduce download times and reliance on connectivity.

To meet these challenges, two solutions naturally present themselves:

- **Identify relevant online sources:** By identifying data relevant to posed queries, the total query dataset can be reduced (see first challenge). Moreover, the system only needs to download online sources containing query-relevant data, tackling our second challenge. Such identification may occur pro-actively (i.e., before any queries have been posed) or re-actively (i.e., after the application posed a query). For instance, some approaches pro-actively locate useful Semantic Web data, by correlating the information to the user's context. Afterwards, any posed queries are executed on the pro-actively selected dataset (see Chapter 2, section 2.2.6) [12, 62]. However, our general-purpose query service should not be limited to any type of scenario (e.g., context-awareness). As an alternative, the system can automatically identify relevant information per individually posed query (e.g., as in query distribution approaches [67, 68]). This re-active approach supports any scenario, as encapsulated by application queries (e.g., location-awareness, recommendation). On the other hand, this approach requires downloading online sources during query resolution, increasing query resolving times. This extra cost can be reduced by applying the second solution, locally cache data.

- **Locally cache data:** By locally caching online data, fewer sources need to be downloaded to serve a posed query (see second challenge, minimize source downloads). In doing so, query resolution time can thus be reduced (see above). When performing caching, care should be taken not to monopolize the volatile and persistent storage of mobile devices. The need for caching data in a mobile setting, for instance to cope with connectivity issues, is reflected in related work (see Chapter 2, section 2.2.9) [73, 76].

To further meet the presented challenges, any software component realizing these solutions should meet two key requirements:

Fine-grained data identification and retrieval: Query-relevant data should be retrieved at a high level of granularity; this means relevant online sources are identified with a high selectivity (see first solution, identify relevant online sources), while locally cached, query-relevant data is also retrieved in a fine-grained way (see second solution, locally cache data). This way, the query dataset is further minimized (see first challenge, reduce the total query dataset); at the same time, ruling out more irrelevant online sources also avoids unnecessary downloads (see second challenge, minimizing source downloads).

Reduce memory usage and processing effort: When realizing our challenges and requirements, software components should not strain the memory and processing capacity of mobile devices. Although these capabilities have increased greatly, there is still a significant gap with larger devices (e.g., PC's, laptops), whereby factors such as battery consumption are currently keeping manufacturers from closing the divide. Firstly, a relatively limited amount of volatile memory is available (e.g., 64MB on the Android platform), meaning the additional memory required by the components (e.g., to store supporting data structures such as B+-trees and hash tables) needs to be minimal. Ideally, the data should fit in volatile memory to avoid frequent swapping with persistent storage, which unavoidably causes performance loss. Secondly, as mentioned, mobile applications are able to dynamically extend their relevant Semantic Web selection as new data is discovered. Therefore, software components (i.e., their internal data structures) need to be updateable on-the-fly, with minimal computational effort, to avoid straining the processing capacity of mobile devices.

4.3 General approach

In order to meet the identified challenges, the mobile query service realizes the two proposed solutions, namely identifying relevant online sources and locally caching data. Implementing these solutions, the *Source Index Model* (SIM) component identifies query-relevant online sources, while the *Source Cache* and *Meta Cache* components locally cache downloaded source data. In more detail, the SIM performs the identification task by indexing source metadata (i.e., predicates, subject/object types) found in online sources. The cache components organize the cached data in a particular way, in order to balance the fine-graininess of cached data retrieval with processing and memory effort (see requirements). In particular, the Source Cache organizes cached source data according to origin source, while the Meta Cache arranges the cached data based on shared source metadata. We developed these two separate cache components to investigate the impact of source metadata on data selectivity and memory/computational overhead. For the same purpose, three variants of the SIM were developed, each keeping an increased amount of metadata. Below, we discuss the rationale behind this focus on source metadata.

Importantly, source metadata is efficiently obtainable and requires only minimal memory space, adhering to req. 2, *Reduce memory usage and processing effort.* In our related work chapter, we discuss the increased overhead of retrieving and storing other kinds of metadata (see Chapter 2, section 2.2.8). Nevertheless, this metadata still enables fine-grained data selection, a hypothesis that is confirmed by our evaluation (see Chapter 5). As a result, req. 1, *Fine-grained data identification and retrieval*, is fulfilled. Clearly, before such source metadata is usable, it needs to be available in both online sources and posed queries. First, we observe that online sources typically specify subject/object types to detail their contained RDF resources. This is substantiated by the real-world dataset used in our experiments, which was extracted from a variety of existing online sources (see Chapter 5, section 5.1.2). At the other end, semantic queries often specify subject and object types to restrict triple patterns. These two observations are also reflected by other approaches in the query distribution field, which focus on indexing predicates [67, 70] and found types [68] to identify query-relevant sources.

Although this focus on source metadata yields a balance between selectivity and overhead, it also incurs a drawback related to the Semantic Web's Open World Assumption (OWA). More specifically, the OWA implies that online sources may describe any existing RDF resource. For the mobile query service, this means new sources may specify additional types for already processed source data; potentially leading to indexed source metadata to become out-of-date. To update components on-the-fly, a resource-intensive *type mediation* process needs to be performed. We note that other approaches, integrating semantic data from multiple sources, also experience this problem but do not consider it [12, 67, 68]. Furthermore, our evaluation (see Chapter 5, section 5.5.2.2) indicates that such typing issues occur only very rarely in our collected real-world dataset. Section 4.6 discusses support for the Semantic Web's OWA.

An overview of the query service components and phases can be found in Figure 4-1. To locally query RDF(S)/OWL data, the query service utilizes an existing mobile query engine (e.g., Androjena, RDF On The Go [28]). Below, we give a detailed overview of the query service phases, and elaborate on how components cooperate in each phase.

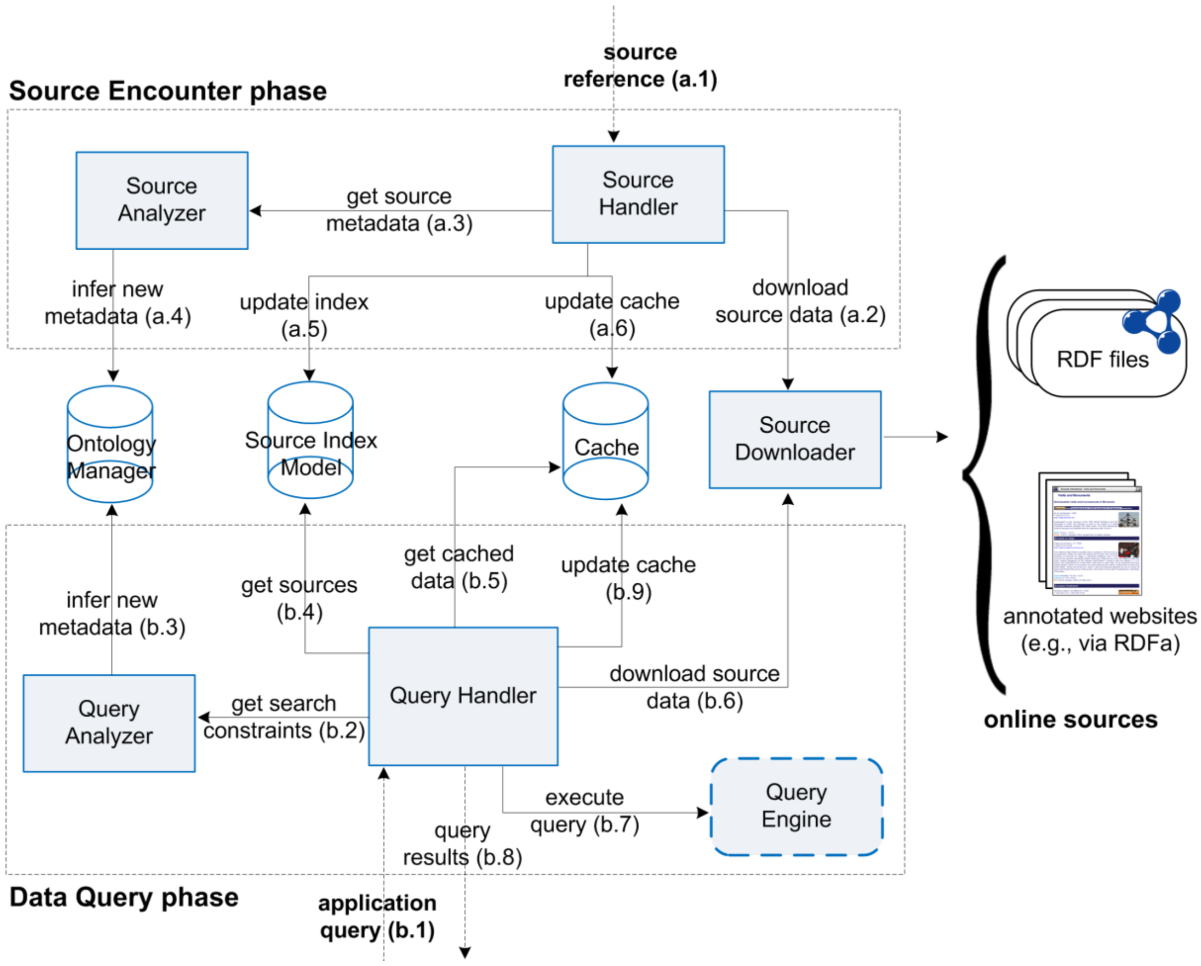


Figure 4-1. Overview of the components and phases of the query service.

During the *Source Encounter* phase, the client (i.e., mobile application utilizing the query service) passes the location of a newly discovered online source to the query service (a.1), extending its particular selection of the Semantic Web. In our evaluation (see Chapter 5), the context-provisioning SCOUT framework acts as client, which passes source references describing the user's physical surroundings. After receiving a source reference (a.1), the Source Handler employs the Source Downloader to download the source data (a.2). Aside from online RDF files, the Source Downloader also supports semantically annotated websites as semantic data sources, by automatically extracting their semantic annotations as RDF triples (currently, RDFa-annotated websites are supported). Next, the semantic source data is passed to the Source Analyzer (a.3), which extracts the required *source metadata*, including predicates and subject/object types. Optionally, the Source Analyzer employs the Ontology Manager component to infer additional source metadata, based on axioms from well-known ontologies (a.4). The extracted source metadata is then passed to the Source Index Model (SIM) (a.5) for indexing, as well as to the cache together with the downloaded source data (a.6).

The *Data Query* phase is triggered when the client poses a query (b.1). Firstly, the Query Handler hands the query to the Query Analyzer (b.2). This component analyzes the query and extracts query metadata as *search constraints*, which reflect the extracted source metadata and include concrete predicates and subject/object types. The Query Analyzer may also utilize the Ontology Manager, leveraging its inferencing support to enrich the search constraints (b.3). The search constraints are then passed to the SIM, yielding references to online sources containing relevant data (b.4). Afterwards, the cache component is contacted (b.5). Based on the identified source references and extracted search constraints, locally cached source data is returned. Subsequently, sources not found in the cache, i.e., because the cached source data was previously removed (due to full storage), are re-downloaded by the Source Downloader (b.6). An existing mobile query engine (e.g., Androjena) is then employed to execute the query on the collected dataset (b.7), after which the query results are returned to the client (b.8). Finally, the cache is updated with the re-downloaded source data (b.9).

In the sections below, we elaborate on the two solutions realized by the mobile query service, namely identifying online sources and caching source data, and discuss how the implementing components adhere to our requirements (see section 4.2).

4.4 Identifying online sources

By indexing newly encountered online sources, relevant sources can be identified at query time. The Source Index Model (SIM) performs this indexing and identification step. To comply with our requirements (see section 4.2), the SIM is specifically designed to maintain a very compact index, and to be quick to update and maintain (see req. 2, *Reduce memory usage and processing effort*). At the same time, the SIM should still guarantee high source selectivity (see req. 1, *Fine-grained data identification and retrieval*). In order to reconcile these two goals, the SIM focuses on indexing source metadata (i.e., predicates and their subject/object types).

Below, we discuss the multi-level index employed by the SIM component, and how it is utilized to identify query-relevant sources (section 4.4.1). In the related work chapter (Chapter 2, section 2.2.8), we discuss indexing structures employed by other related approaches (e.g., RDF stores and query distribution approaches) and compare them to the SIM. Section 4.7.1 elaborates on the core SIM implementation.

4.4.1 Multi-level index

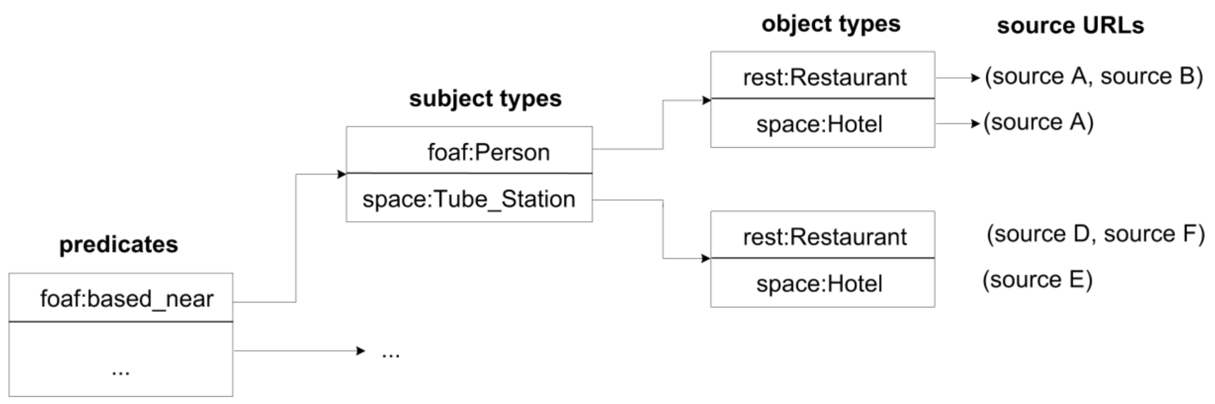


Figure 4-2. Example of the SIM multi-level index.

To enable fast identification of query-relevant sources, the SIM employs a multi-level index. Each level indexes on a particular metadata part (i.e., predicates, subject/object types), and keeps map data structures that connect metadata parts occuring together in source triples. Figure 4-2 shows an example multi-level index, keeping source metadata extracted from sources A – F. These sources encode location-related information using the FOAF[65], space[66] and restaurant[67] ontologies, and contain triples with predicate "foaf:based_near"; subject type "foaf:Person" and/or "space:Tube_Station"; and object type "rest:Restaurant" and/or "space:Hotel". Each found combination of metadata parts corresponds to a path through the multi-level index, where the last level points to a list of sources in which the combination was found. For instance, in Figure 4-2, the metadata combination *foaf:based_near – foaf:Person – space:Hotel* was extracted from one (or more) source triple(s) in source A.

```
SELECT ?place
WHERE {
    ?person rdf:type foaf:Person .
    ?person foaf:based_near ?place .
    ?place rdf:type rest:Restaurant .  (1)
    ?place rdf:type space:Hotel .      (2)
}
```

Code 4-1. Example query, whereby the SIM is employed to identify relevant sources.

For a given SPARQL query, the SIM identifies relevant sources per individual triple pattern[68]. This enables the query service to resolve queries not solvable by any single source. Each triple pattern in

[65] http://xmlns.com/foaf/0.1/ (access date: 24/05/2013)
[66] See http://wise.vub.ac.be/ontologies/space.rdf (access date: 24/05/2013)
[67] See http://wise.vub.ac.be/ontologies/restaurant.owl (access date: 24/05/2013)
[68] This does not include triple patterns specifying subject/object types; in our solution, these provide type constraints for the other triple patterns.

WHERE, OPTIONAL and UNION clauses are investigated, while FILTER clauses are scanned for functions specifying a predicate, subject or object variable type (using sameTerm()).

In order to illustrate how the SIM performs the identification task, we consider the example multi-level index discussed above, together with the query triple patterns shown in Code 4-1. The triple patterns in italic provide type information for the pattern in bold. When only considering the first object type (indicated by (1)), the following query metadata combination (or search constraint) is extracted from the triple pattern: *foaf:based_near – foaf:Person – rest:Restaurant*. The SIM attempts to follow the path, indicated by the query metadata combination, through the multi-level index. Specifically, given the predicate key in the first index, the SIM returns a second index with subject type keys. The final index maps object type keys to lists of source URLs containing the given combination of predicate, subject and object type. This search leads to a list of URLs containing the given query metadata combination (if any). This process is illustrated in Figure 4-3 (1), where sources A, B are returned for our example.

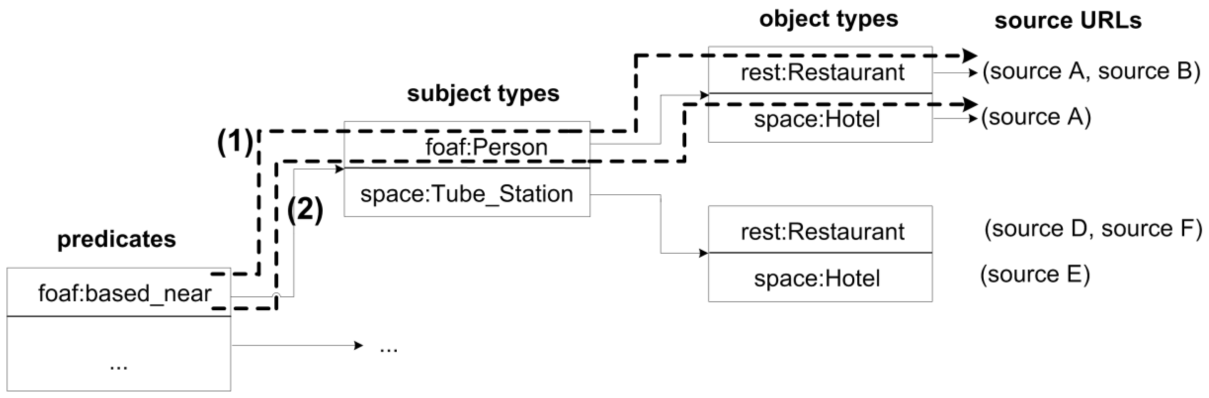


Figure 4-3. Example navigation of the SIM multi-level index.

In case the subject/object variable of a triple pattern has multiple types, sources need to be identified containing all specified subject/object types. Specifically, separate metadata combinations are extracted for each type, whereby identified sources should contain each metadata combination. For instance, when also considering the second object type in Code 4-1 (indicated by (2)), our example gives rise to the following two metadata combinations: *foaf:based_near – foaf:Person – rest:Restaurant* and *foaf:based_near – foaf:Person – space:Hotel*. Each individual combination is followed through the multi-level index, returning sets of sources in which the particular combination occurs. This is illustrated in Figure 4-3 (1,2), where our example returns sources A, B. Since we are looking for sources comprising *both* metadata combinations, the intersection of these sets is taken, returning source A in our example.

It should be noted that sources, identified via the above process, do not necessarily comprise single triples adhering to all search metadata combinations. Instead, these sources may contain multiple triples, each of which adhering to an individual metadata combination. For instance, two source triples could exist with subject type foaf:Person and predicate foaf:based_near, whereby the first triple has an object of type rest:Restaurant, and the second triple an object of type space:Hotel. However, these source triples do not yield query-relevant data, since their object resource should adhere to all related type constraints in the query. As a result, the SIM may identify online sources that do not actually contain data relevant to the query; thus conflicting with req. 1, *Fine-grained data identification and retrieval.* On the other hand, resolving this issue would require keeping separate sets of metadata combinations occurring in individual triples. This would incur much more memory overhead and thus break req. 2, *Reduce memory usage and processing effort*.

The SIM also supports online sources comprising triples without subject/object types, as well as query triple patterns without concrete predicates or subject/object types. For this purpose, the map data structures contain a special *<empty>* entry. For instance, if an extra source G keeps a triple with predicate foaf:based_nearby, subject type foaf:Person, and no object types, the source metadata combination *foaf:based_nearby – foaf:Person – <empty>* would be extracted. Figure 4-4 shows the SIM extended with this source triple. The <empty> entry is utilized when query triple patterns have missing metadata parts, as elaborated below.

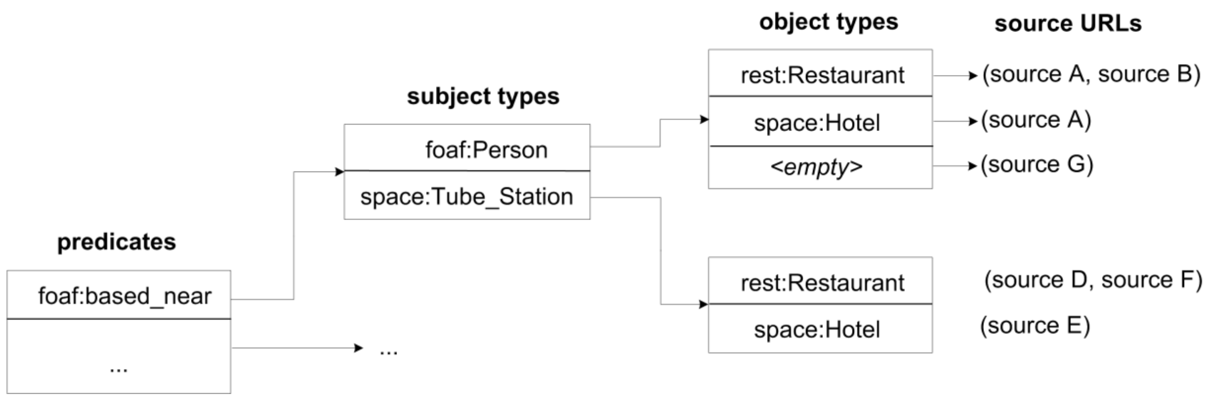


Figure 4-4. The SIM multi-level index extended with an empty entry.

In case a query triple pattern does not have a concrete predicate, or no subject/object types were provided, the SIM will follow all paths starting at the levels standing for the missing metadata. This includes the paths starting from concrete entries (e.g., rest:Restaurant, space:Hotel) as well as the *<empty>* entry. This strategy reflects the fact that, as no constraint (i.e., predicate, subject/object types) is provided for the particular level, all paths starting at that level will eventually point to relevant data sources. After the sources are identified for each path, their union is returned. For instance, in case the object types (1) and (2) are left out from the query specified in Code 4-1, the query metadata combination *foaf:based_nearby – foaf:Person – <empty>* is extracted. In Figure 4-5, we illustrate how the SIM identifies relevant sources for this query metadata combination. Since no object type restrictions were provided, sources are identified for all paths starting at each of the level's entries, namely rest:Restaurant, space:Hotel, and <empty>, yielding the union of A, B and G.

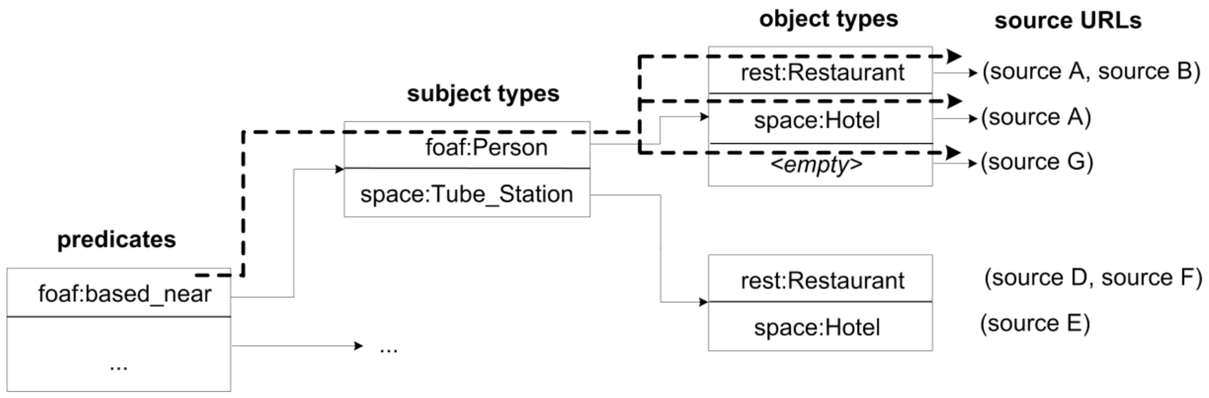


Figure 4-5. Example navigation of the SIM multi-level index whereby no object types are given.

In order to reduce SIM size, we apply dictionary encoding (see section 4.7.1.1). An overview of the SIM implementation can be found in section 4.7.1.

4.5 Caching source data

By locally caching source data, the number of online source downloads, required to serve posed queries, is reduced. To meet our requirement of *Fine-grained data identification and retrieval* (see section 4.2), any cache should identify query-relevant cached data with high selectivity, thus also reducing the final query dataset. Secondly, only a small amount of additional data (e.g., indices such as B+-trees) should be stored, ideally fitting in volatile memory (currently, 64 Mb on the Android platform). Moreover, creating and updating the cache should be quick and efficient. That way, req. 2, *Reduce memory usage and processing effort*, is met.

In this section, we consider two caching systems: Meta Cache, organizing cached data according to shared metadata; and Source Cache, which organizes the cache via origin source. In order to save memory and storage space, certain cached data can be moved to persistent storage or removed entirely. For this purpose, we present a replacement policy (or removal strategy) called Least-Popular-Sources (LPS), tailored to our particular setting. Finally, we ensure the freshness of the cache by applying a validity strategy.

Below, we discuss the Meta Cache and Source Cache organizations, and compare them to weigh their advantages and drawbacks (section 4.5.1). Note that we also elaborate on other caching mechanisms in the related work chapter (Chapter 2, section 2.2.9.1). Then, we discuss useful removal strategies and detail the Least-Popular-Sources strategy (section 4.5.2). Other removal strategies, specifically developed for mobile scenarios, are also detailed in related work (Chapter 2, section 2.2.9.2). Section 4.5.3 discusses the cache validity strategy. Finally, an implementation overview is given in section 4.7.2.

4.5.1 Cache organizations

Cached data can be organized in different ways, influencing the fine-graininess of data retrieval, as well as the cache maintenance cost and in-memory size requirements. Below, we discuss the Source Cache organization, and compare it to Meta Cache.

4.5.1.1 Source Cache

The Source Cache organizes cached triples via their origin source. This is a natural choice in our setting, where data originates from individual online sources. In a caching system, data is indexed, stored and retrieved per unit of data (called cache unit), whereby the particular unit depends on the cache organization. In case of Source Cache, an individual cache unit contains all data from a certain online source. A search index (implemented using a hash table) is kept on cached source URLs, whereby each URL uniquely identifies a cache unit. In order to identify the URLs of cached sources that are relevant to posed queries, the Source Cache is deployed in combination with the SIM.

The Source Cache organization leads to only minimal memory overhead, since only one index is kept with a relatively small amount of index entries; while the SIM is specially designed to have a low-memory impact. Updating the cache on-the-fly is fast, since each source is added as a single cache unit. As such, Source Cache meets req. 2, *Reduce memory usage and processing effort*. However, it does not support fine-grained data selection, as a retrieved data unit comprises a complete query-relevant source, instead of only their relevant triples. Therefore, it does not adhere to req. 1, *Fine-grained data identification and retrieval*. As shown by our experimental validation, this leads to huge overall query resolution overheads. Although such coarse-grained retrieval is unavoidable when dealing with online sources (as indicated by our second challenge, *Data captured in online files*), we can improve on the fine-graininess of data retrieval when dealing with local data.

4.5.1.2 Meta Cache

Meta Cache organizes source triples according to their shared metadata. In this organization, a cache unit contains triples sharing the same metadata combination, i.e., predicate, subject/object types. Search indices are kept on predicates, subject and object types; allowing useful cached data to be quickly returned, given a particular metadata combination. This organization allows data to be obtained in a more fine-grained way, as only triples matching the query's search constraints (i.e., query metadata) are returned. As such, it adheres to req. 1, *Fine-grained data identification and retrieval*. However, this comes with an additional computational and storage overhead, contradicting req. 2, *Reduce memory usage and processing effort*. Firstly, the cache update time is increased, since the metadata of each source triple needs to be extracted. Furthermore, caching triples from a new source likely requires creating / updating multiple cache units (i.e., all units matching the metadata combinations of the source's triples). Regarding memory space, this organization requires three indices with considerable more index entries than Source Cache, since the number of distinct predicates and types will likely exceed the number of source URLs. Moreover, each cached triple needs to keep its origin source URL, since a cache unit may group data from multiple sources. For instance, this enables checking the validity and freshness of cached data (see section 4.5.3).

On the other hand, we note that the Meta Cache still has a much lower memory and update overhead compared to other indexing approaches (see Chapter 2, section 2.2.8), due to its focus on source metadata (i.e., schema-level information). Moreover, our experimental validation shows that the computational and memory overhead is still reasonable, especially compared to the resulting increase in overall query execution time (see Chapter 5, section 5.3.2). However, another issue resulting from the Meta Cache organization is related to cache removal (see next section).

It should be noted that Meta Cache also keeps information on cached data that was previously removed to clear storage space ("missing" data). More specifically, it stores the metadata combination associated with the removed data, as well as the locations of the online sources from which the data originated. This information on missing data is indexed in the same way as the cached data, via the aforementioned search indices. In this way, a cache lookup may return cached data, as well as references to online sources to be re-downloaded. This mechanism represents an optimization, since it rules out the need for a separate source identification component (i.e., SIM) and its accompanying overhead; which is already relatively high for the Meta Cache component. As a result, the Meta Cache implements both the *online source identification* and *local caching* solutions. In case Meta Cache is utilized, the online source identification task (see Figure 4-1/b.4) is thus also performed by the cache component.

4.5.2 Cache removal strategies

After some time, a caching system can aggregate huge amounts of data. While modern mobile devices have increased volatile and persistent storage, users are probably not keen on spending a large portion to store cached data. A replacement policy (or removal strategy) identifies data be moved to persistent storage or removed entirely. To achieve this, removal strategies typically identify data not likely to be referenced in the future, by assuming a particular locality of reference. For instance, temporal locality indicates that items that have been recently referenced will likely be referenced again, and is employed by the Least-Recently-Used policy (LRU). As mentioned in related work (see Chapter 2, section 2.2.9.2), much work has been done concerning removal strategies meant for location-aware scenarios. However, since our mobile query service does not target one particular type of client application, we focus on general-purpose removal strategies re-usable across different application scenarios.

Below, we discuss "regular" strategies such as Least-Recently-Used and Furthest-Away-Replacement (FAR) [76], and how they can pose problems in Meta Cache. Then, we introduce the Least-Popular-Sources removal strategy, which is fine-tuned to our particular querying scenario.

4.5.2.1 Regular removal strategies

The Least-Recently-Used (LRU) replacement policy assumes temporal locality, identifying cache units least recently referenced by the client application to be stored persistently or removed. The FAR [76] replacement policy assumes a certain semantic locality (i.e., based on general properties and relations of data [73]). More specifically, it assumes that cached data, associated with a nearby location in the user's current movement direction, will likely be required in the future. In general, the suitability of a particular removal strategy thus depends on the observed locality of reference exhibited by cache access. For instance, in location-aware scenarios, client applications will frequently access location-related data (FAR); in more general scenarios, applications may often access recently referenced data (LRU).

For Source Cache, regular strategies (e.g., LRU) perform as expected. However, we found that such removal strategies can cause major problems for Meta Cache, because of the particular cache organization [86]. In Meta Cache, a stored cache unit contains triples likely originating from multiple sources. This means that, whenever a cache unit is removed and later referenced during query resolution (i.e., cache miss), all sources containing the associated metadata need to be re-downloaded. As a result, query execution overhead becomes exceedingly high in case of cache misses. We note that this issue results from our particular setting, where data is captured in online files that need to be fully downloaded (see second challenge in section 4.2, *Data captured in online files*). Experimental results, both in our experimental validation (see Chapter 5) and a previous evaluation [86], show that removal strategies not considering this issue may cause a huge download overhead during the Data Query phase.

Below, we present a removal strategy that takes this issue into account.

4.5.2.2 Least-Popular-Sources (LPS)

In this section, we present a removal strategy called Least-Popular-Sources (LPS), a so-called source-based removal strategy we tailored to our setting where data is captured in online files. In Figure 4-6, LPS is applied on a cache where data from sources A, B and C are spread across multiple cache units.

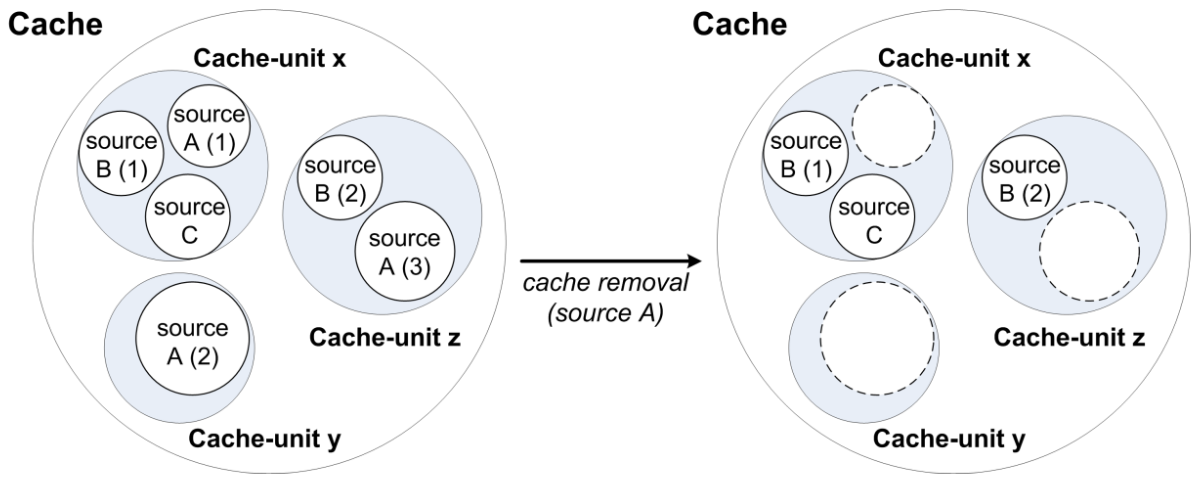


Figure 4-6. An example application of LPS (Least-Popular-Sources).

Importantly, the source-based LPS strategy identifies origin sources (e.g., source A) to be removed, instead of cache units (e.g., cache-unit x). By removing data on a per-source level, a cache miss resulting from a single removal only triggers a single source download, instead of a multitude of downloads. This is illustrated in Figure 4-6. After source A has been removed, the referencing of *cache unit x* only results in one source download, instead of three[69]. As such, this strategy aims to avoid the aforementioned issue with multiple downloads (see section 4.5.2.1), and thus increase adherence to our second challenge of minimizing source downloads (see section 4.2). A drawback is that the probability of a cache miss (i.e., accessing a cache unit with missing data) increases as well, since one removal may influence multiple cache units (e.g., units x, y and z in Figure 4-6). Also, a single removal becomes more complex, since multiple cache units need to be updated. In our evaluation, we investigate how the incurred overheads weigh against the overall increase in efficiency. While this dissertation considers the application of LPS to Meta Cache, we note that LPS may be applied to any cache where the data originates from online sources and is not internally organized via origin source.

In order to reduce query-time overheads, LPS considers the "popularity" of the source in the cache, which is reflected by 1/ the popularity of its source data, i.e., the number of stored cache units containing the data, and 2/ the popularity of its own metadata, i.e., how often the contained metadata is found in other sources. By considering factor 1, we reduce the impact of removing the source (since less cache units are influenced), thus decreasing the probability of a cache miss. For instance, removing source C (see Figure 4-6) has a relatively low impact on the cache (as only one cache unit is affected), reducing the probability of a cache miss later on. As such, this factor mediates the increased likelihood of cache misses when utilizing a source-based removal strategy.

[69] Note that this is a simplified example; in a real-world setting, a cache unit will typically contain data from many more sources.

During experiments however, we observed that while factor 1 indeed reduces cache misses, it also increases the overall amount of source re-downloads; i.e., the amount of sources that needs to be downloaded in response to a cache miss. In Figure 4-7, we illustrate a situation where only factor 1 is considered.

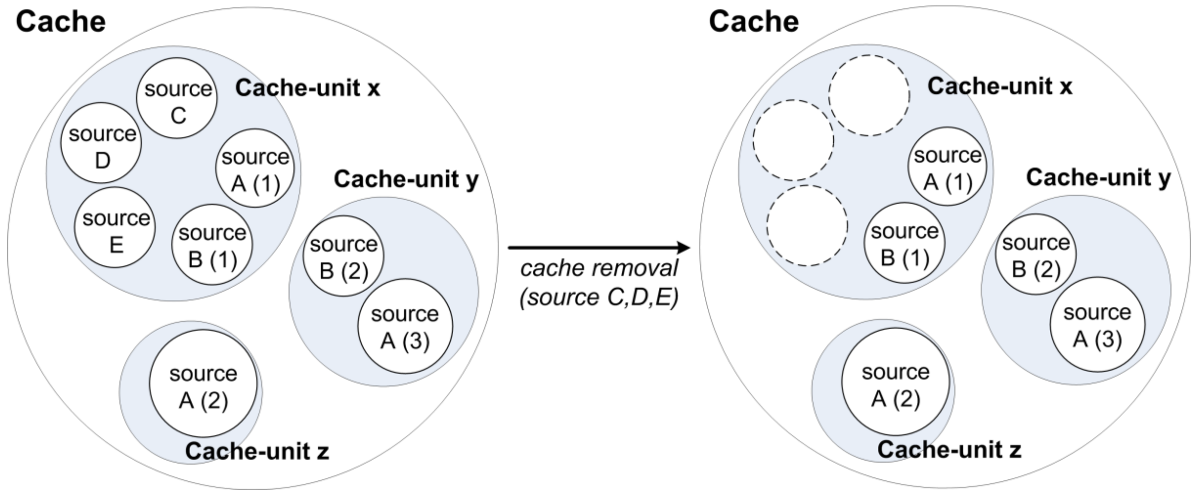


Figure 4-7. Example application of LPS when only considering factor 1.

Small cached sources typically contain less distinct metadata combinations, reducing their value for factor 1. For instance, the relatively small sources C, D and E are only involved in one cache unit (i.e., cache unit x), reducing their value for factor 1 and thus making them candidates for removal. In addition, we note that an increased amount of small sources need to be removed to clear the same amount of memory space. As illustrated in Figure 4-7, this leads to a cache composition with a comparably large number of removed sources for cache unit x; which increases the potential number of source re-downloads per cache miss. For instance, in case cache unit x is referenced, three sources need to be downloaded instead of just one.

Factor 2 indicates the popularity of the comprised metadata in the online semantic dataset, delineated by the mobile application. Importantly, the second factor helps to keep the number of source re-downloads per cache miss in check. As the number of associated sources increases for a certain metadata combination, the second factor's value becomes higher for these associated sources, as they contain increasingly "popular" source metadata. Therefore, it becomes less likely that many of these associated sources will be removed. For instance, sources C, D and E have an increased value for factor 2, since cache unit x involves a relatively large number of sources (5), making the associated metadata combination popular. As such, the likelihood that many of them are removed is decreased. In this respect, the second factor serves as a counter-weight to the first factor, allowing for a balance between cache misses and source re-downloads. This is confirmed by our experimental validation (see Chapter 5, section 5.4). As an additional advantage, it could be assumed that the more popular a particular combination of metadata, the more likely it will be referenced by a query fired by the mobile application. As such, sources containing these kinds of popular metadata combinations should not be removed from the cache.

As a final consideration, LPS can also take into account the source download cost, which leads to sources that are expensive to download (i.e., have long download times) being less likely to be removed. Different weights may be set for each of the three factors, to accommodate different scenarios (e.g., particular data distributions that make factor 1 or 2 more suitable). Below, we show the formula for calculating the removal value for source s, where f1 stands for source data popularity, f2 for source metadata popularity, and f3 for download cost; and α, β and γ represent the respective factor weights:

$$lps(s) = \alpha f1 + \beta f2 + \gamma f3$$

4.5.3 Cache validity

Various invalidation strategies exist to detect invalid, no longer up-to-date information in client-server architectures and mobile scenarios. For instance, a cache invalidation strategy called Selective Adaptive Sorted (SAS) is presented in [87], where updates on data items on the server are reflected on the mobile device. In [88, 89], location-dependent cache invalidation approaches are presented, which ensure validity of location-specific cached data, retrieved from information services. However, these strategies are not suitable in our setting, where cached data originates from online files stored on general-purpose web servers instead of dedicated servers. Therefore, we rely on the cache support of HTTP (typically also used by proxy caches).

To support validity management, we keep information on each retrieved source, potentially including last download time and expiration time (e.g., indicated via the “Expires” header field). After a predefined amount of time, a process checks the validity of each source. Importantly, by leveraging HTTP caching support, validity checks can be performed without having to download all the source data and checking for changes (e.g., via checksum comparison). In particular, either the server-specified expiration time is checked; or a conditional GET request is sent, whereby the last download time is filled into the “Last-Modified-Since” header field. In case this conditional request returns (updated) source data, or the expiration time has been exceeded, the outdated source triples are removed from the cache and replaced by the new source data.

4.6 Semantic Web Open World Assumption

The vision of the Semantic Web is that of an open, interlinked web of semantic data, where data providers publish information on any person, place, or thing identifiable by a resource URI. To support this vision, Semantic Web technologies implement the Open World Assumption (OWA). In contrast to the Closed World Assumption (CWA), the OWA states that the absence of a statement does not imply its negation. In essence, this means no single data source is assumed to be comprehensive, and any other data source can extend existing resource descriptions. Additionally, the OWA allows dealing with new RDF descriptions in a flexible way, via logical axioms that enable inferring new statements. Below, we discuss the OWA support provided by the mobile query service.

In the Semantic Web, RDF resources may be described by multiple online sources. For instance, a person's FOAF profile can list people the person knows, who are likely described in more detail in their own FOAF profiles. Each data source may hereby specify additional information for these resources (e.g., extra person details), including new resource types (e.g., foaf:Person). For the mobile query service, this means newly encountered sources may specify extra types for already processed source data; possibly leading to indexed source metadata to become outdated. In order to keep the query service metadata up-to-date, resource types should be tracked across sources; whereby component data structures are updated[70] when necessary. We call this updating process *type mediation*. In case multiple online sources specify different types for identical RDF resources, type mediation needs to be enabled to guarantee that all relevant query results are returned. In case the same resource types are specified across online sources, the process may be disabled to reduce overhead; for instance, typing issues occurred very rarely in our real-world experiment dataset (see Chapter 5, section 5.5.2.2). We also note that other approaches, which integrate Semantic Web data from multiple sources, suffer this problem but do not consider it in their work [12, 67, 68].

Secondly, ontologies can be defined using the Web Ontology Language (OWL), containing logical axioms that place restrictions on RDF models. For instance, an ontology may contain a property domain restriction, which constrains the types of related subject resources; e.g., the domain restriction of the foaf:familyName property implies the type of subject resources is foaf:Person (reflecting the fact only people have a family name). Importantly, any RDF model not stating any types (or different types) for these subject resources is still considered valid. Typically, such RDF models do not explicitly state the constrained type is *not* applicable; and since the absence of type statements does not imply their negation, it can also not be assumed the type is not applicable. Such cases essentially imply a state of affairs where the domain restriction holds [90]. Therefore, Semantic Web reasoners use these kinds of axioms to infer new type statements (e.g., foaf:Person in the previous example), and make them explicit in the original RDF data. Likewise, the query service exploits these property domain and range restrictions to infer new resource types, using the inferred types to enrich extracted source and query metadata. This process is called type inferencing. Most RDF stores support type inferencing (e.g., Androjena), and typically allow enabling/disabling inferencing to suit application needs and improve performance. Analogously, the query service allows disabling type inferencing.

Below, we elaborate on type inferencing and type mediation, and discuss how they impact the query service and its components.

[70] Only components utilizing source metadata, i.e., the SIM and Meta Cache, need to be updated.

4.6.1 Type inferencing

As explained, the Semantic Web's Open-World Assumption (OWA) allows the inferencing of new type information based on logical axioms specified in OWL ontologies. With regards to our query service, type inferencing may be applied during the Source Encounter phase, in order to enrich extracted source metadata; and on posed queries during the Data Query phase, to extend extracted search constraints. Below, we elaborate on how the two phases are extended to support type inferencing.

The Source Encounter phase (see Figure 4-1) is extended with the Ontology Manager component, which provides inferencing support based on axioms from online ontologies. Via the Ontology Manager, the Source Analyzer component retrieves each predicate's domain/range types (possibly including their subtypes), and adds them to the extracted source metadata. By making more source metadata available, search constraints can identify increased amounts of query-relevant semantic data.

For instance, consider the following small RDF snippet in Code 4-2 (namespaces omitted for brevity):

```
vub:denker_in_alle_staten rdfs:label "Denker in alle staten" .
vub:denker_in_alle_staten geo:xyCoordinates
                          "50.82242202758789,4.393936634063721" .
```

Code 4-2. Example RDF snippet illustrating type inferencing at the source side.

In this case, the Source Analyzer obtains the domain type restriction of the geo:xyCoordinates predicate (specified in the GeoFeatures[71] ontology) via the Ontology Manager, namely geo:SpatialEntity. In case a query was invoked, requesting for instance labels of all encountered resources of type geo:SpatialEntity, the vub:denker_in_alle_staten resource would be correctly returned as a result.

During the Data Query phase, the Query Analyzer component, responsible for extracting search constraints from posed queries, leverages the same ontological knowledge to enhance the search constraints. Utilizing the Ontology Manager's inferencing support, the Query Analyzer obtains each concrete predicate's domain and range types (possibly accompanied by their subtypes) and adds them to the search constraints. This way, data identification becomes more selective, ruling out more irrelevant source data.

For instance, consider a query containing the following two triple patterns (Code 4-3) (namespaces omitted for brevity):

```
?restaurant lgd:cuisine ?cuisine .
?restaurant rdfs:label ?label .
```

Code 4-3. Example query to illustrate type inferencing at the query side.

[71] http://www.mindswap.org/2003/owl/geo/geoFeatures20040307.owl# (access date: 03/04/2013)

Here, the Query Analyzer obtains the domain type restriction of the `lgd:cuisine` predicate (as specified in the LGD[72] ontology) from the Ontology Manager, namely `lgd:Restaurant`. As query-relevant sources are identified per triple pattern, such inferred types can greatly improve data selectivity. For instance, since no type constraints are given and `rdfs:label` is a much-occurring predicate, a great deal of source data will be identified for the second triple pattern. However, since both triple patterns share the same subject variable (i.e., `?restaurant`), the inferred `lgd:Restaurant` can also be added to the second search constraint, making it much more selective.

We note that, during query resolving, type inferencing needs to be re-applied to retrieved online sources, which were for instance downloaded due to a cache miss. Although the Source Analyzer previously enriched the source metadata with inferred types, the online sources cannot be updated with the newly inferred types. This is a direct result from our specific setting, where data is captured in online files not under our control (see second challenge, *Data captured in online files*; section 4.2).

The Ontology Manager component dynamically downloads ontologies when required, by checking the namespaces of passed terms (e.g., predicates, in case domain/range types are requested) and resolving them to their online locations. For this purpose, the Ontology Manager is initialized with a configuration file, which connects the namespaces of well-known ontologies to concrete URLs where the ontology may be found (together with its format; e.g., RDF/XML). If a particular namespace URI is not found in the configuration file, an attempt is made to resolve the namespace URI itself to an online location. After downloading an ontology, the Ontology Manager stores it locally for later use. To reduce memory usage, retrieved ontologies are cleared from memory after analyzing a source or query.

4.6.2 Type mediation

In the Semantic Web, data sources may reference RDF resources already described elsewhere. In order for the same resource to be easily identified across different sources, an important Linked Data guideline is to re-use the same URI for identical resources, and using constructs (e.g., owl:sameAs) to indicate resource equivalence [91].

In case resources occur across multiple online sources, whereby sources may specify different types for these resources, the type mediation process needs to be applied. This process requires keeping track of resource types across encountered sources, and updates components to reflect new types for already processed source data. In case type mediation is not applied, some query-relevant resources would not be returned in response to posed queries. The type mediation process is resource-intensive regarding memory and processing effort, since it requires keeping types for each individual resource, and updating component data structures when necessary. As such, it contradicts req. 2, *Reduce memory usage and processing effort*. On the other hand, our index structure keeping

[72] http://downloads.linkedgeodata.org/releases/110406/LGD-Dump-110406-Ontology.nt.bz2 (access date: 03/04/2013)

resource types (among other information) still consumes less memory space than the ones used by for instance RDF stores (see section 4.7.3).

In case such typing issues do not occur in the online semantic dataset (i.e., no sources exist where the same RDF resources are typed differently), type mediation can be disabled. For instance, this is the case for scenarios where there is complete control over the contents of online sources, and the sources can be supplemented with the missing resource types. In other cases, knowledge on the online dataset can be utilized to rule out the existence of typing issues. For instance, the real-world dataset used in our experimental validation (see Chapter 5, section 5.1.2) comprises FOAF profiles, where a person's profile typically lists RDF resources standing for other persons (e.g., acquaintances indicated via foaf:knows); each of which are described in more detail in their own FOAF profile, including additional types (e.g., foaf:Person). In case type mediation was not enabled, the experiment query referencing these FOAF profiles would return less query-relevant results. On the other hand, in this experimental dataset, only a very limited number of typing issues occurred, compared to the total dataset size (see Chapter 5, section 5.5.2.2). In cases where the amount of typing issues is negligible, or do not occur at all, type mediation can be disabled to reduce overhead.

We note that other approaches integrating Semantic Web data also suffer from this problem, but do not make an effort to tackle it. Consequently, the current state of the art corresponds to the case where type mediation is disabled in the mobile query service. For instance, the SemWIQ [68] query distribution approach keeps a catalog for each data source, listing the classes and their number of instances. However, other data sources may specify additional types for the contained RDF resources, resulting in an incomplete catalog. The DARQ [67] query distribution system keeps so-called service descriptions for each data source, including constraints on subject and object resources. In case other data sources specify extra types for contained resources, these types should also occur in the subject and object constraints. MobiSem [12] replicates RDF data on the local device from multiple online sources, depending on the user's context. However, the online sources may supply different types for the same RDF resources, possibly resulting in less results being returned when querying each dataset separately.

An overview of the realization of the type mediation process, including the supporting implementation and process logic, can be found in section 4.7.3.

4.7 Implementation

In this section, we summarize the core implementation of the two major query service components, namely the Source Index Model (section 4.7.1) and Meta Cache (section 4.7.2). Also, we review the implementation of the OWA type mediation process (section 4.7.3). The implementation sections include UML diagrams showing the core classes[73].

[73] As before, items such as private members, constructors and getter/setter methods are left out.

4.7.1 Source Index Model

Figure 4-8 shows the SIMStrategy package. A SIMStrategy instance represents a SIM component, and keeps references to QueryAnalyzer, SIMSourceAnalyzer and SourceIndexModel instances, as well as a SourceDownloader object. Below, we correlate the package structure to the components and phases illustrated in the overview figure (see Figure 4-1). Note that we only consider individual use of the SIM here[74]. For an overview of the cache package, where the SIM is utilized in combination with a cache component, we refer to section 4.7.2. The OntologyManager, which is used to perform type inferencing, is not elaborated here and is discussed in section 4.6.1.

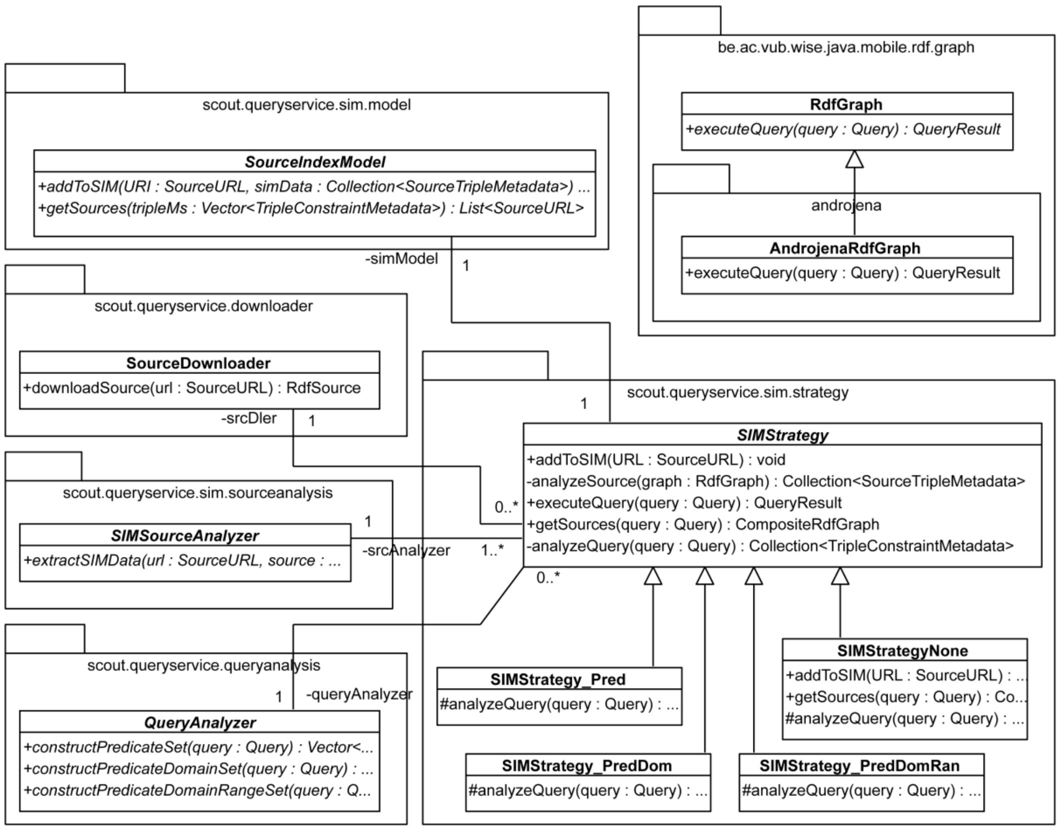

be.ac.vub.wise.java.mobile.rdf.graph
RdfGraph
+executeQuery(query : Query) : QueryResult
androjena
AndrojenaRdfGraph
+executeQuery(query : Query) : QueryResult
scout.queryservice.sim.model
SourceIndexModel
+addToSIM(URI : SourceURL, simData : Collection<SourceTripleMetadata>) ...
+getSources(tripleMs : Vector<TripleConstraintMetadata>) : List<SourceURL>
-simModel
1
scout.queryservice.downloader
SourceDownloader
+downloadSource(url : SourceURL) : RdfSource
-srcDler
1
scout.queryservice.sim.sourceanalysis
SIMSourceAnalyzer
+extractSIMData(url : SourceURL, source : ...
1
-srcAnalyzer
1..*
0..*
scout.queryservice.sim.strategy
1
SIMStrategy
+addToSIM(URL : SourceURL) : void
-analyzeSource(graph : RdfGraph) : Collection<SourceTripleMetadata>
+executeQuery(query : Query) : QueryResult
+getSources(query : Query) : CompositeRdfGraph
-analyzeQuery(query : Query) : Collection<TripleConstraintMetadata>
0..*
scout.queryservice.queryanalysis
1
-queryAnalyzer
QueryAnalyzer
+constructPredicateSet(query : Query) : Vector<...
+constructPredicateDomainSet(query : Query) : ...
+constructPredicateDomainRangeSet(query : Q...
SIMStrategy_Pred
#analyzeQuery(query : Query) : ...
SIMStrategyNone
+addToSIM(URL : SourceURL) : ...
+getSources(query : Query) : Co...
#analyzeQuery(query : Query) : ...
SIMStrategy_PredDom
#analyzeQuery(query : Query) : ...
SIMStrategy_PredDomRan
#analyzeQuery(query : Query) : ...

Figure 4-8. *Query Service - SIM:* SIMStrategy package.

During the Source Encounter phase, the SIMStrategy assumes the role of Source Handler. The SIMStrategy is contacted to index new sources (see Figure 4-1/a.1) by invoking its addToSIM method (see Figure 4-8). Firstly, the SIMStrategy employs the SourceDownloader class to download the online semantic source (see Figure 4-1/a.2), which returns an RdfSource encapsulating the downloaded RDF graph. The SourceDownloader automatically detects the type of the online semantic source: in case of an RDF file, the contained RDF triples are simply downloaded; in case of a

[74] In our experimental validation (see chapter 5), we evaluate a setup where only the SIM component is utilized, to investigate the utility of keeping an extra local cache.

website, the component attempts to extract its RDFa annotations as RDF triples, using an Android-ported version of java-rdfa[75]. Afterwards, the SIMSourceAnalyzer, fulfilling the role of Source Analyzer, extracts the required source metadata from the downloaded source (see Figure 4-1/a.3). Finally, the SIMStrategy passes the source metadata on to the SourceIndexModel instance for indexing (see Figure 4-1/a.5).

In the Data Query phase, the SIMStrategy acts as the Query Handler. In this case, the SIMStrategy to contacted (see Figure 4-1/b.1) by invoking its executeQuery method (see Figure 4-8). To extract search constraints from the posed query (see Figure 4-1/b.2), SIMStrategy relies on the QueryAnalyzer class, acting as the Query Analyzer component. The SIMStrategy then contacts the SourceIndexModel instance with the extracted search constraints, obtaining a list of query-relevant source URLs (see Figure 4-1/b.4). Afterwards, the SIMStrategy employs the aforementioned SourceDownloader to download each identified source (see Figure 4-1/b.6). Finally, SIMStrategy relies on an existing mobile query engine to execute the query on the collected source data (see Figure 4-1/b.7). After invoking the executeQuery method, the results are returned to the query service (see Figure 4-1/b.8).

To locally query and work with RDF data, the query service utilizes a custom RDF library (not shown in detail), containing an abstract RdfGraph class that can be subclassed to support any Android RDF library. Currently, we provide support for Androjena[76], which is a ported version of the well-known Apache Jena framework[77]. We extended this library with support for efficiently combining RDF graphs, which significantly optimized the collection of identified sources into a single dataset during querying (see above).

The SIMStrategy subclasses represent different SIM variants. In our evaluation (see Chapter 5), we compare three SIM variants that keep varying amounts of metadata, with the goal of evaluating memory overhead and performance. The first variant keeps only predicates (SIMStrategy_Pred, where Pred stands for Predicate); the second variant keeps predicates and subject types (SIMStrategy_PredDom, where Dom stands for domain); and the third variant keeps predicates and subject/object types (SIMStrategy_PredDomRan, where Ran stands for Range). SIMStrategyNone is utilized in case the query service does not employ the SIM (e.g., when the Meta Cache is employed; see section 4.5).

Below, we first discuss wrapper classes encapsulate information in the query service, enabling dictionary encoding. Then, we elaborate on implementing packages of the major SIM components.

4.7.1.1 Dictionary encoding

In order to reduce memory consumption on resource-restricted devices, the query service performs dictionary encoding, both in the SIM and Meta Cache (see section 4.7.2). Dictionary encoding is also

[75] https://github.com/shellac/java-rdfa (access date: 24/05/2013)
[76] https://code.google.com/p/androjena/ (access date: 15/05/2013)
[77] http://jena.apache.org/ (access date: 15/05/2013)

applied by existing RDF stores to reduce memory usage [71, 72]. Each piece of stored data (e.g., URLs, metadata parts) is wrapped in an abstract class, whereby one of the subclasses performs dictionary encoding. For instance, the TripleMetadataPart class represents a metadata part (i.e., type or predicate, extracted from an online source or query), whereby the DictEncTriplePart subclass keeps a dictionary-encoded version of the term URI. Figure 4-9 shows TripleMetadataPart, together with its subclasses and related abstract factory classes.

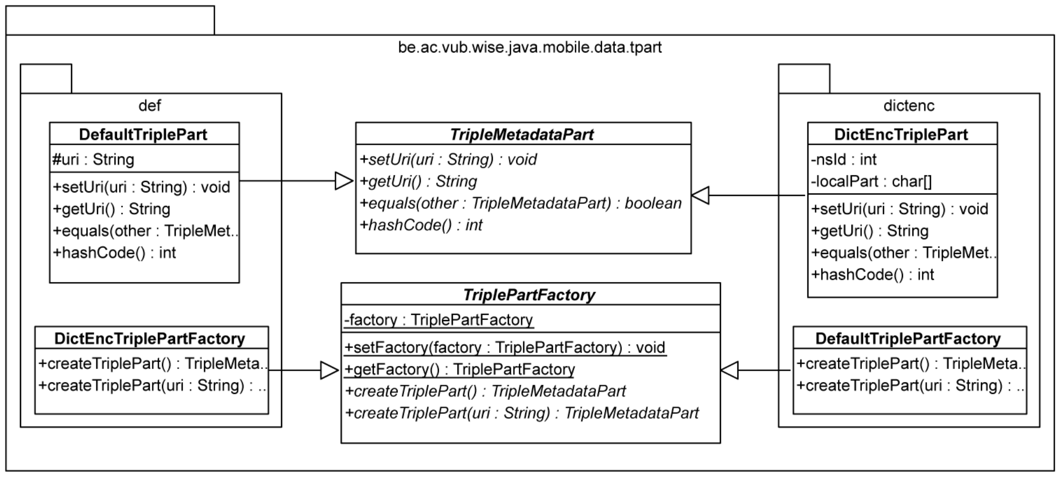

be.ac.vub.wise.java.mobile.data.tpart
def
DefaultTriplePart
#uri : String
+setUri(uri : String) : void
+getUri() : String
+equals(other : TripleMet..
+hashCode() : int
DictEncTriplePartFactory
+createTriplePart() : TripleMeta..
+createTriplePart(uri : String) : ..
TripleMetadataPart
+setUri(uri : String) : void
+getUri() : String
+equals(other : TripleMetadataPart) : boolean
+hashCode() : int
TriplePartFactory
-factory : TriplePartFactory
+setFactory(factory : TriplePartFactory) : void
+getFactory() : TriplePartFactory
+createTriplePart() : TripleMetadataPart
+createTriplePart(uri : String) : TripleMetadataPart
dictenc
DictEncTriplePart
-nsId : int
-localPart : char[]
+setUri(uri : String) : void
+getUri() : String
+equals(other : TripleMet..
+hashCode() : int
DefaultTriplePartFactory
+createTriplePart() : TripleMeta..
+createTriplePart(uri : String) : ..

Figure 4-9. *Query Service:* TripleMetadataPart package

The DictionaryEncoder class (not shown) is responsible for performing dictionary encoding. For instance, RDF terms (i.e., predicates, types and resources) are encoded by mapping their namespace to an integer identifier, and keeping the local part as a character array. We found that this resulted in the largest reduction in memory usage, as namespaces are repeated across data sources much more often.

4.7.1.2 SIMSourceAnalyzer

The SIMSourceAnalyzer is responsible for extracting the required source metadata from a downloaded RDF source. We show the SIMSourceAnalyzer package in Figure 4-10.

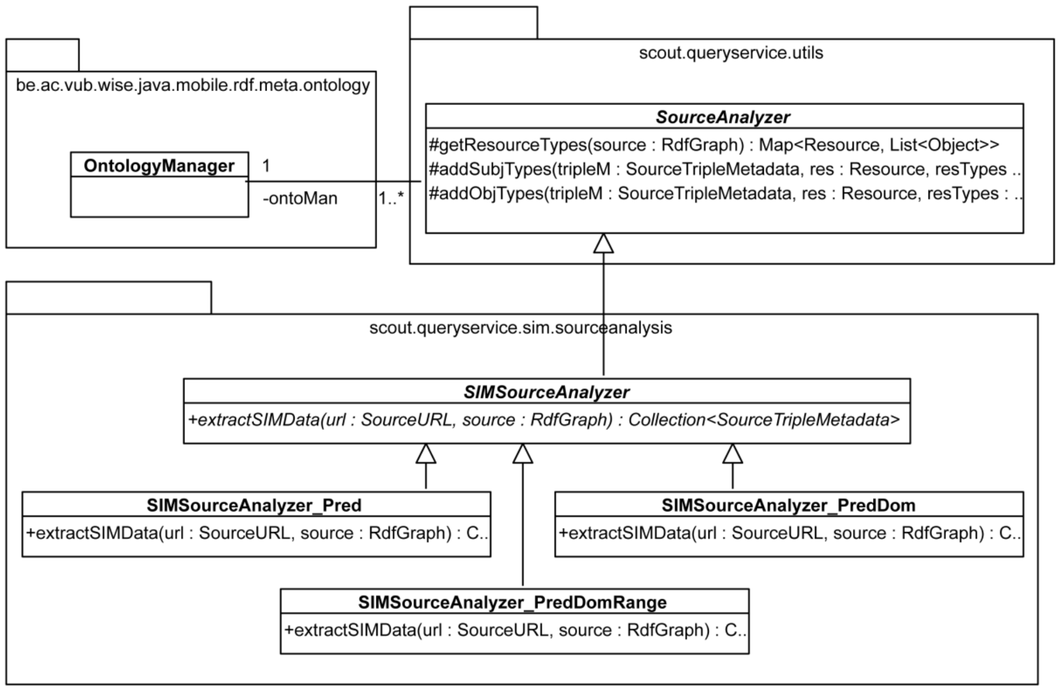

be.ac.vub.wise.java.mobile.rdf.meta.ontology
OntologyManager
1
-ontoMan
1..*
scout.queryservice.utils
SourceAnalyzer
#getResourceTypes(source : RdfGraph) : Map<Resource, List<Object>>
#addSubjTypes(tripleM : SourceTripleMetadata, res : Resource, resTypes ..
#addObjTypes(tripleM : SourceTripleMetadata, res : Resource, resTypes : ..
scout.queryservice.sim.sourceanalysis
SIMSourceAnalyzer
+extractSIMData(url : SourceURL, source : RdfGraph) : Collection<SourceTripleMetadata>
SIMSourceAnalyzer_Pred
+extractSIMData(url : SourceURL, source : RdfGraph) : C..
SIMSourceAnalyzer_PredDom
+extractSIMData(url : SourceURL, source : RdfGraph) : C..
SIMSourceAnalyzer_PredDomRange
+extractSIMData(url : SourceURL, source : RdfGraph) : C..

Figure 4-10. *Query Service - SIM:* SIMSourceAnalyzer package.

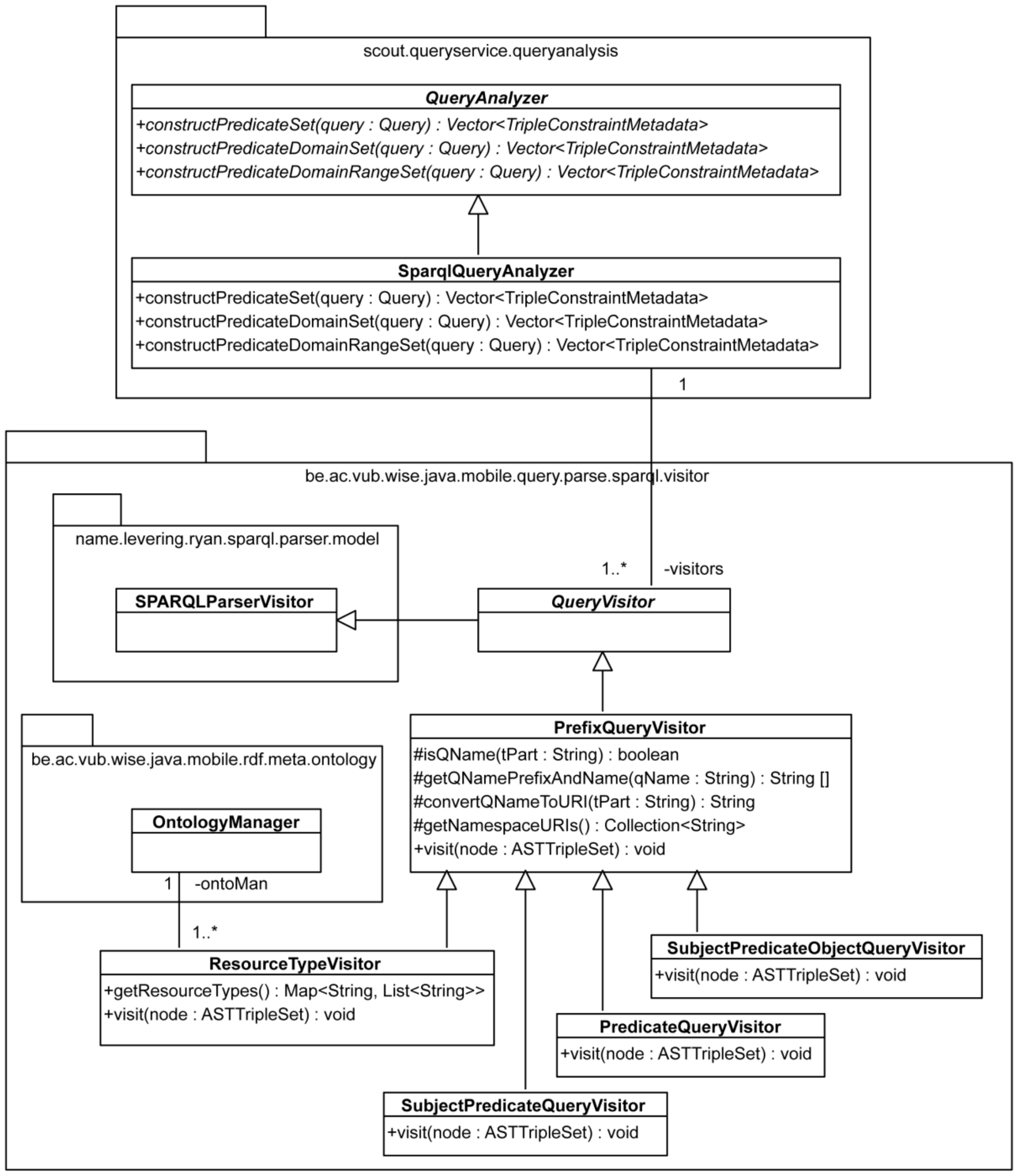
The SIMSourceAnalyzer class is contacted (see extractSIMData method) to extract source metadata from a given RdfGraph. The metadata is returned as a set of SourceTripleMetadata instances (class not shown), each of which aggregates metadata (i.e., predicates, subjects and object types) for an individual source triple. As such, these objects collectively present the extracted *source metadata.* A SIMSourceAnalyzer subclass retrieves source metadata for one of the three SIM variants (see Figure 4-8). The SIMSourceAnalyzer class itself is a subclass of the SourceAnalyzer class, which allows its subclasses to retrieve a map from resources to their types found in an RDF graph. This superclass is also used by the Meta Cache (see section 4.7.2.2).

Initially, the necessary source metadata was retrieved by executing predefined SPARQL extraction queries on the source RDF graph [86, 92]. However, when dealing with relatively large sources with substantial amounts of type information, this led to significant processing overhead on mobile devices (ca. 31s for 3,5Mb of source data). To resolve this issue, we extended our RDF library (diagram not shown) with an abstract RdfStatementIterator class, which enables client code to efficiently iterate over individual RDF statements. Its NTripleStatementIterator subclass dynamically parses RDF files in N-Triple format, extracting new RDF statements as they are requested via the iterator interface. By relying on this subclass to extract source metadata, we managed to speed up the extraction process by a factor 10 on average. RdfStatementIterator subclasses for other RDF formats (e.g., N3) can be easily plugged in later on.

To perform type inferencing, SourceAnalyzer utilizes the OntologyManager class (see section 4.6.1).

4.7.1.3 QueryAnalyzer

The QueryAnalyzer is utilized by both the SIM and Meta Cache (see section 4.7.2) to extract search constraints from queries. Figure 4-11 shows the QueryAnalyzer package.



scout.queryservice.queryanalysis
QueryAnalyzer
+constructPredicateSet(query : Query) : Vector<TripleConstraintMetadata>
+constructPredicateDomainSet(query : Query) : Vector<TripleConstraintMetadata>
+constructPredicateDomainRangeSet(query : Query) : Vector<TripleConstraintMetadata>
SparqlQueryAnalyzer
+constructPredicateSet(query : Query) : Vector<TripleConstraintMetadata>
+constructPredicateDomainSet(query : Query) : Vector<TripleConstraintMetadata>
+constructPredicateDomainRangeSet(query : Query) : Vector<TripleConstraintMetadata>
1
be.ac.vub.wise.java.mobile.query.parse.sparql.visitor
name.levering.ryan.sparql.parser.model
SPARQLParserVisitor
1..*
-visitors
QueryVisitor
PrefixQueryVisitor
#isQName(tPart : String) : boolean
#getQNamePrefixAndName(qName : String) : String []
#convertQNameToURI(tPart : String) : String
#getNamespaceURIs() : Collection<String>
+visit(node : ASTTripleSet) : void
be.ac.vub.wise.java.mobile.rdf.meta.ontology
OntologyManager
1
-ontoMan
1..*
ResourceTypeVisitor
+getResourceTypes() : Map<String, List<String>>
+visit(node : ASTTripleSet) : void
SubjectPredicateObjectQueryVisitor
+visit(node : ASTTripleSet) : void
PredicateQueryVisitor
+visit(node : ASTTripleSet) : void
SubjectPredicateQueryVisitor
+visit(node : ASTTripleSet) : void

Figure 4-11. *Query Service:* QueryAnalyzer package.

The QueryAnalyzer extracts query predicates (see constructPredicateSet method); predicates and subject types (constructPredicateDomainSet method); and predicates and subject/object types (constructPredicateDomainRangeSet method), thus accommodating each of the three SIM variants we studied. These methods return a list of TripleConstraintMetadata objects, each of which encapsulates metadata extracted from an individual query triple pattern (cfr. SourceTripleMetadata; see section 4.7.1.2). Together, these objects thus represent the extracted *search constraints*. The SparqlQueryAnalyzer subclass implements support for SPARQL queries and utilizes one or more QueryVisitors (not all visit methods shown), which realize the Visitor design pattern. For instance, SparqlQueryAnalyzer employs the SubjectPredicateQueryVisitor to extract the concrete predicate and subject URI term from each triple pattern, and the ResourceTypeVisitor to retrieve a mapping from URI terms to their specified types. In order to perform type inferencing, the ResourceTypeVisitor relies on the OntologyManager class (see section 4.6.1). We utilize the SPARQL Parser library[78] to parse SPARQL queries and subsequently visit the parsed Abstract Syntax Tree (AST).

4.7.1.4 SourceIndexModel

The SourceIndexModel class indexes source metadata and identifies query-relevant sources. The SourceIndexModel package is shown in Figure 4-12.

[78] http://sparql.sourceforge.net/ (access date: 19/03/2013)

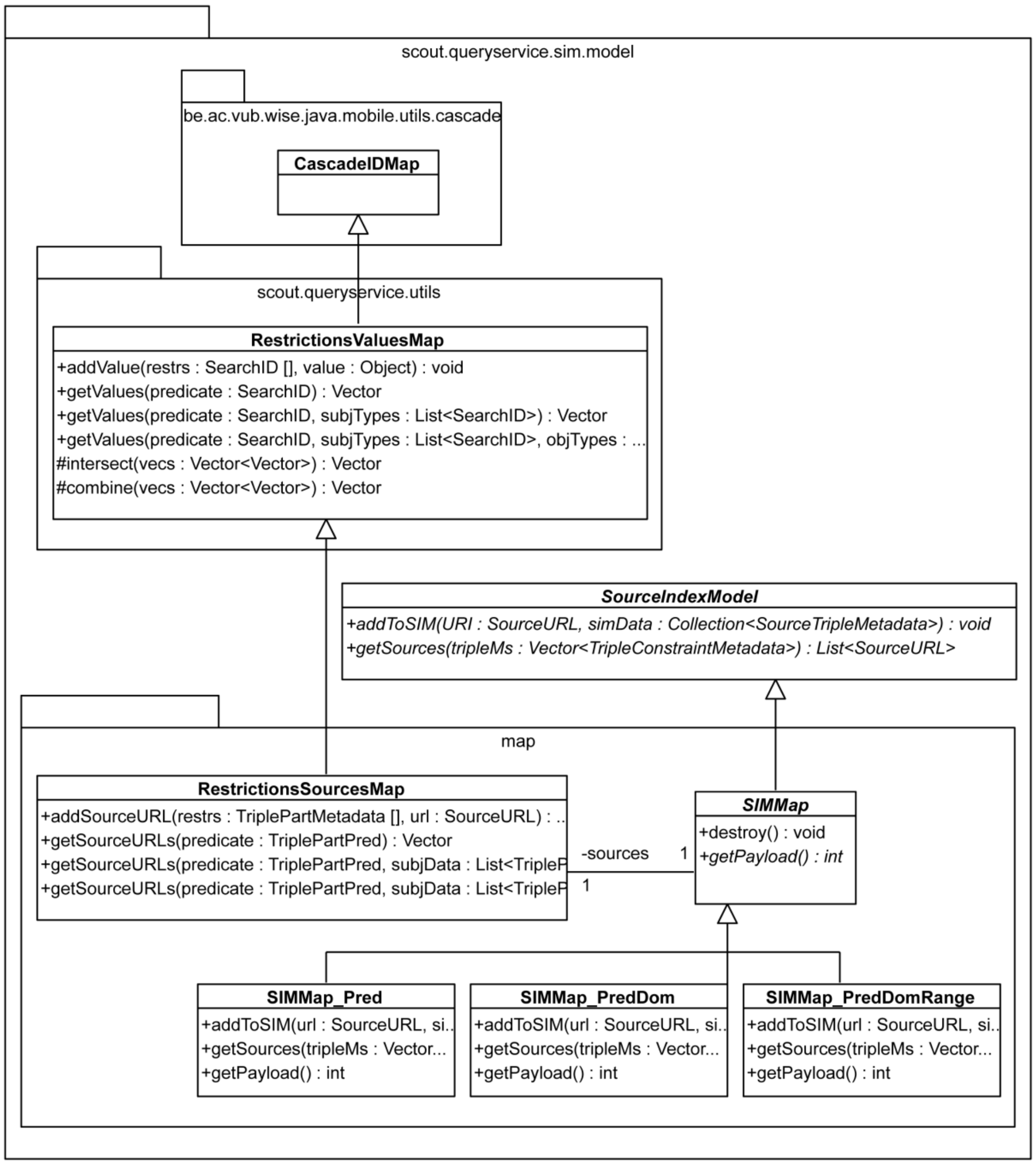
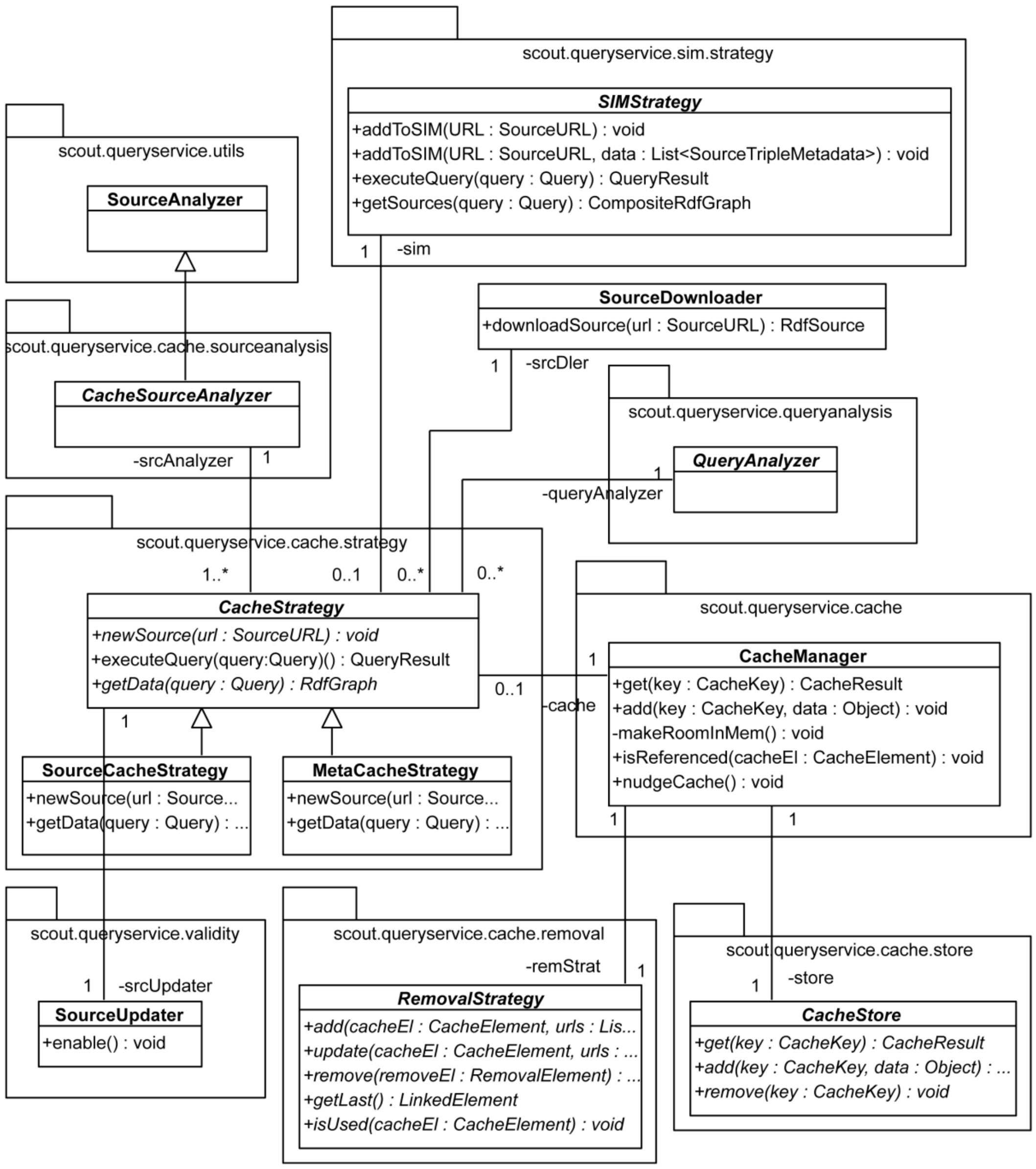

scout.queryservice.sim.model
be.ac.vub.wise.java.mobile.utils.cascade
CascadeIDMap
scout.queryservice.utils
RestrictionsValuesMap
+addValue(restrs : SearchID [], value : Object) : void
+getValues(predicate : SearchID) : Vector
+getValues(predicate : SearchID, subjTypes : List<SearchID>) : Vector
+getValues(predicate : SearchID, subjTypes : List<SearchID>, objTypes : ...
#intersect(vecs : Vector<Vector>) : Vector
#combine(vecs : Vector<Vector>) : Vector
SourceIndexModel
+addToSIM(URI : SourceURL, simData : Collection<SourceTripleMetadata>) : void
+getSources(tripleMs : Vector<TripleConstraintMetadata>) : List<SourceURL>
map
RestrictionsSourcesMap
+addSourceURL(restrs : TriplePartMetadata [], url : SourceURL) : ..
+getSourceURLs(predicate : TriplePartPred) : Vector
+getSourceURLs(predicate : TriplePartPred, subjData : List<TripleP
+getSourceURLs(predicate : TriplePartPred, subjData : List<TripleP
-sources
1
1
SIMMap
+destroy() : void
+getPayload() : int
SIMMap_Pred
+addToSIM(url : SourceURL, si..
+getSources(tripleMs : Vector...
+getPayload() : int
SIMMap_PredDom
+addToSIM(url : SourceURL, si..
+getSources(tripleMs : Vector...
+getPayload() : int
SIMMap_PredDomRange
+addToSIM(url : SourceURL, si..
+getSources(tripleMs : Vector...
+getPayload() : int

Figure 4-12. *Query Service - SIM:* SourceIndexModel package.

After the QueryAnalyzer has extracted the search constraints (see Figure 4-11), SIMStrategy passes them on to the SourceIndexModel in the form of TripleConstraintMetadata objects (see getSources method), after which a list of query-relevant SourceURLs is returned. The SIMMap subclass represents the multi-level index solution discussed in section 4.4.1. A generic version of this multi-level index is implemented by the CascadeIDMap class. Its RestrictionsValuesMap subclass realizes behavior specific for the SIM, such as performing an intersection of the results in case search constraints specify multiple subject/object types (see section 4.4.1). The RestrictionsSourcesMap is a subclass of RestrictionsValuesMap, and provides an interface for keeping SourceURLs as payload in the multi-level index. The SIMMap class relies on the RestrictionsSourcesMap class to act as multi-level index. Finally, SIMMap has three subclasses, to support the three different SIM variants.

4.7.2 Cache

Figure 4-13 shows the CacheStrategy package. A CacheStrategy instance represents a cache component, keeping references to QueryAnalyzer, CacheSourceAnalyzer and SourceDownloader, as well as CacheManager and (potentially) SIMStrategy. The CacheManager further keeps a reference to CacheStore and RemovalStrategy instances. Below, we correlate the package structure to the phases and components from the overview figure (see Figure 4-1). The OntologyManager is discussed in section 4.6.1.



scout.queryservice.sim.strategy
SIMStrategy
+addToSIM(URL : SourceURL) : void
+addToSIM(URL : SourceURL, data : List<SourceTripleMetadata>) : void
+executeQuery(query : Query) : QueryResult
+getSources(query : Query) : CompositeRdfGraph
1 -sim
scout.queryservice.utils
SourceAnalyzer
scout.queryservice.cache.sourceanalysis
CacheSourceAnalyzer
-srcAnalyzer 1
SourceDownloader
+downloadSource(url : SourceURL) : RdfSource
1 -srcDler
scout.queryservice.queryanalysis
QueryAnalyzer
1
-queryAnalyzer
scout.queryservice.cache.strategy
1..* 0..1 0..* 0..*
CacheStrategy
+newSource(url : SourceURL) : void
+executeQuery(query:Query)() : QueryResult
+getData(query : Query) : RdfGraph
0..1
-cache
1
SourceCacheStrategy
+newSource(url : Source...
+getData(query : Query) : ...
MetaCacheStrategy
+newSource(url : Source...
+getData(query : Query) : ...
scout.queryservice.cache
1
CacheManager
+get(key : CacheKey) : CacheResult
+add(key : CacheKey, data : Object) : void
-makeRoomInMem() : void
+isReferenced(cacheEl : CacheElement) : void
+nudgeCache() : void
1 1
scout.queryservice.validity
1 -srcUpdater
SourceUpdater
+enable() : void
scout.queryservice.cache.removal
-remStrat 1
RemovalStrategy
+add(cacheEl : CacheElement, urls : Lis...
+update(cacheEl : CacheElement, urls : ...
+remove(removeEl : RemovalElement) : ...
+getLast() : LinkedElement
+isUsed(cacheEl : CacheElement) : void
scout.queryservice.cache.store
1 -store
CacheStore
+get(key : CacheKey) : CacheResult
+add(key : CacheKey, data : Object) : ...
+remove(key : CacheKey) : void

Figure 4-13. *Query Service - Cache:* CacheStrategy package.

In the Source Encounter phase, the CacheStrategy acts as Source Handler. In case of a newly encountered source, the CacheStrategy is contacted (see Figure 4-1/a.1) by calling its newSource method (see Figure 4-13). First, the SourceDownloader is employed to download the online source (see Figure 4-1/a.2), returning an RdfSource wrapping the downloaded RDF graph. As mentioned (see section 4.7.1), the SourceDownloader supports both online RDF files as well as RDFa-annotated websites. Afterwards, the CacheSourceAnalyzer, representing the Source Analyzer component, is contacted, returning the source data accompanied by their associated metadata (see Figure 4-1/a3). In case the chosen cache type requires a SIM component (i.e., Source Cache), the extracted source metadata is first communicated towards the SIMStrategy instance for indexing (see extra addToSIM method). Then, the downloaded source data, together with its associated metadata, is passed to the CacheManager for local caching (see add method). The CacheManager utilizes a CacheStore instance for actual data storage, and relies on a RemovalStrategy object to manage memory and persistent storage space.

During the Data Query phase, the CacheStrategy takes up the role of Query Handler. In order to retrieve query-relevant data, the CacheStrategy is contacted (see Figure 4-1/b.1) by calling its executeQuery method[79] (see Figure 4-13). The CacheStrategy first contacts the QueryAnalyzer to extract search constraints from the posed query (see Figure 4-1/b.2). Then, two scenarios may ensue; one where the cache component relies on a SIM (i.e., Source Cache) and one where the cache component is standalone (i.e., Meta Cache). In the first case, the CacheStrategy communicates the search constraints to the SIM, with the goal of identifying query-relevant sources (see Figure 4-1/4). Then, the CacheStrategy contacts the CacheManager, to retrieve the identified cached sources and determine which sources need to be re-downloaded (see Figure 4-1/b.5). Each non-cached source is then downloaded using the SourceDownloader object (see Figure 4-1/b.6). In the second case, the CacheStrategy directly passes on the search constraints to the CacheManager, which returns query-relevant cached data as well as missing online source locations to be re-downloaded (see Figure 4-1/b.5). Each missing source is then downloaded using the SourceDownloader (see Figure 4-1/b.6).

To execute the query on the collected data (see Figure 4-1/b.7), the CacheStrategy employs an existing query engine, encapsulated by the abstract RdfGraph class (not shown here). Currently, we support the Androjena query engine. After executing the executeQuery method, the query results are returned to the mobile application (see Figure 4-1/b.8). Finally, the re-downloaded source data is communicated to the CacheManager for local caching (see Figure 4-1/b.9). The SourceUpdater class implements our cache validity strategy (see section 4.5.3), and runs in the background to keep the cache up-to-date with the online origin sources.

The QueryAnalyzer class is elaborated in section 4.7.1.3. Below, we discuss how the cached data is identified and represented, respectively via CacheKey and CacheElement classes (section 4.7.2.1). Afterwards, section 4.7.2.2 discusses the CacheSourceAnalyzer class. We elaborate on the CacheManager in section 4.7.2.3, while the CacheStore and RemovalStrategy classes, employed by the CacheManager class, are elaborated in sections 4.7.2.4 - 4.7.2.6 and 4.7.2.7 - 4.7.2.10, respectively. Finally, section 4.7.2.11 elaborates on how cached data is stored persistently, via the PersistentStorage class.

4.7.2.1 CacheKey and CacheElement

In this section, we discuss the representation and identification of cached data. Figure 4-14 shows the related class diagram. Below, we elaborate on the main classes.

[79] The getData method returns the query-relevant set of source data; e.g., this allows the SCOUT framework to add this query-relevant data to the final query dataset (chapter 3, section 3.4.3).

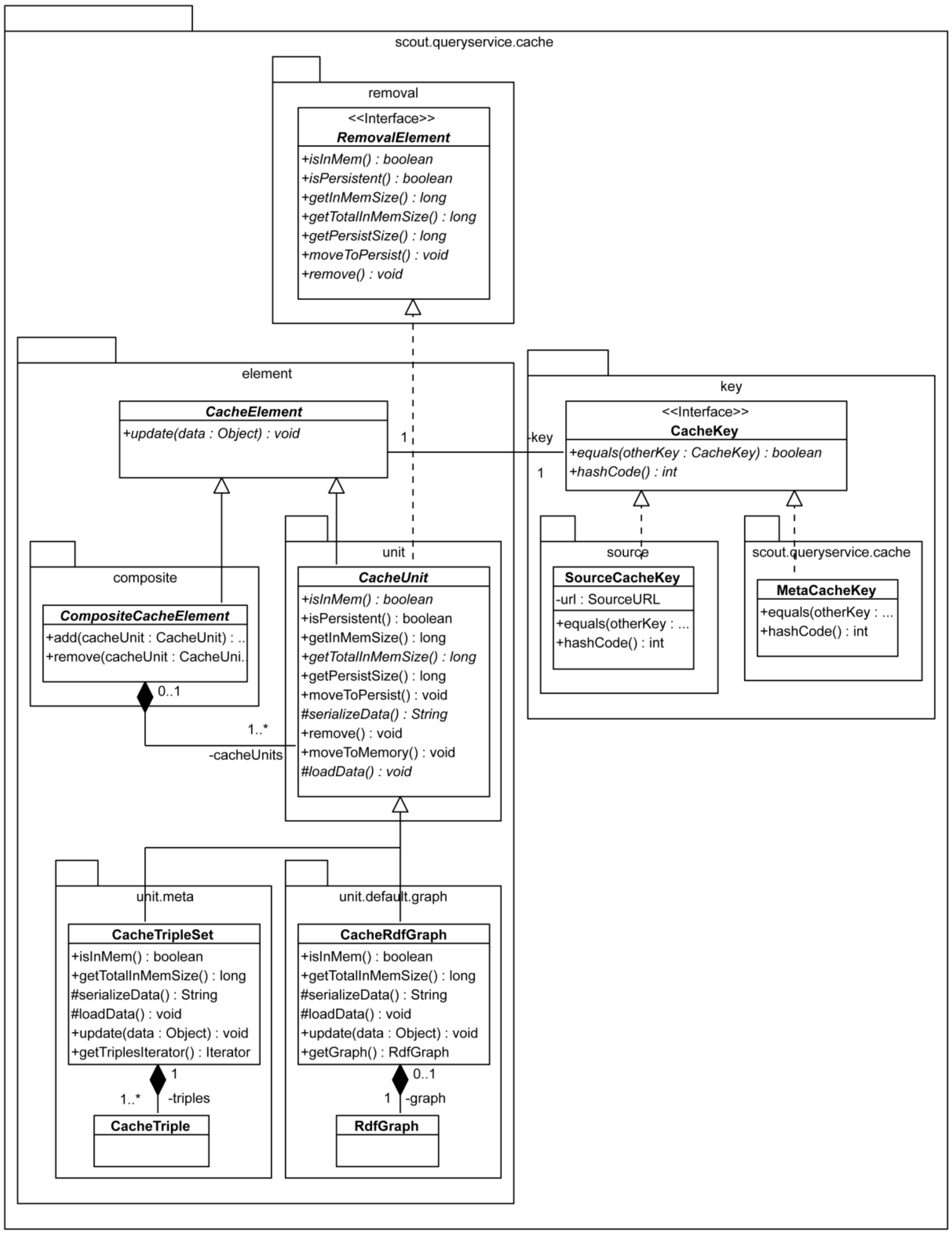


Figure 4-14. *Query Service - Cache:* CacheKey and CacheElement classes.

To increase flexibility, our cache explicitly separates the units of *storage*, *retrieval* and *removal*. Below, we discuss these different units and their implementing classes.

Firstly, the *unit of storage* is represented by the CacheUnit class, which stores the data actually being cached (called the payload) either in-memory or persistently. In Source Cache, the CacheRdfGraph subclass keeps an RDF graph containing all RDF data from a particular source; in Meta Cache, the CacheTripleSet subclass collects all RDF triples sharing the same metadata.

The CacheElement class stands for the *unit of retrieval*, meaning a cache store retrieves a CacheElement instance in response to a lookup. The CacheKey class is responsible for uniquely identifying CacheElement objects in the cache. For Meta Cache, a MetaCacheKey object stands for a particular metadata combination, and identifies a CacheTripleSet containing RDF triples adhering to the metadata. For Source Cache, a SourceCacheKey instance identifies a CacheRdfGraph object containing data from a particular origin source.

Finally, the RemovalElement class serves as the *unit of removal*. In case memory is full, the memory management process (see section 4.7.2.3) stores RemovalElement objects persistently or removes them entirely. The RemovalElement interface supports such memory management operations; e.g., via methods to move data to persistent storage (moveToPersist method) and removing the data (remove method).

The flexibility brought by this separation is required by some setups (i.e., when using certain removal strategies); see section 4.7.2.10 for an example. For most cases however, it is not necessary to differentiate between these units. To facilitate such cases, the CacheUnit class can act as unit of storage, retrieval and removal at the same time, as it is a subclass of CacheElement and implements the RemovalElement interface. The CompositeCacheElement class, subclass of CacheElement, aggregates multiple CacheUnits and also serves as unit of retrieval (see section 4.7.2.10).

As the type of payload depends on the particular cache (e.g., RDF graph vs. set of RDF triples; see below), CacheUnit subclasses define separate methods for returning their payload (e.g., see getGraph and getTriplesIterator methods). In case the CacheUnit had been stored persistently, requesting its payload results in automatically loading the unit back into memory (see moveToMemory method). To fulfill its responsibilities as a removal unit, the CacheUnit class implements various memory management methods (e.g., moveToPersist, remove methods) and delegates other tasks to its subclasses (e.g., serializeData method). Subclasses may also override these implemented methods (e.g., moveToPersist method) for optimization reasons.

Meta Cache relies on the CacheTripleSet subclass, which stores an arbitrary set of RDF triples. These triples are represented by CacheTriple instances, which additionally keep the triple's origin source URL (therefore, they actually stand for quads). The YARS RDF store [71] notes that storing provenance is one of the fundamental necessities in the open, distributed Web environments. The CacheRdfGraph subclass keeps an RDF graph as payload, and is employed by Source Cache. It should be noted that, by letting the CacheTripleSet class keep cached triples individually, we avoid keeping separate RDF graphs for each metadata combination. This would incur a large memory overhead, since there are typically many more metadata combinations than origin sources, and Androjena graphs keep their own internal indices to speed up query access (increasing memory overhead). Furthermore, this enables dictionary encoding to be applied to reduce the memory footprint, while the individual triples can also be easily moved between different units (e.g., necessary in case of type mediation; see section 4.7.3.3). On the other hand, Androjena RDF graphs can be more efficiently combined into the final query dataset, as compared to individual RDF triples. We revisit this issue in our evaluation chapter (see Chapter 5, section 5.3.2).

As mentioned, CacheKey instances uniquely identify CacheElement objects in the cache. Subclasses of CacheKey are responsible for overriding the equals method, to ensure that the method returns true for two identical CacheKey objects, and false otherwise. For instance, in case of Meta Cache, the equals method should only return true in case both MetaCacheKey instances represent the same metadata combination. To guarantee correct hashing, CacheUnit subclasses should also override the hashCode method, whereby two identical CacheKey instances should return the same hash code[80].

4.7.2.2 CacheSourceAnalyzer

The CacheSourceAnalyzer class (see Figure 4-13) is a subclass of the SourceAnalyzer class (similar to the SIMSourceAnalyzer class; see section 4.7.1.2). It has one concrete subclass, namely MetaCacheSourceAnalyzer (class not shown), which extracts the source data and metadata required by Meta Cache. Specifically, it extracts individual triples from the RDF source, accompanied by their associated metadata (i.e., predicates, subject/object types).

The MetaCacheSourceAnalyzer subclass performs this extraction task on a given source RDF graph, and returns the extracted data as a set of MetaSourceData instances. Each MetaSourceData object encapsulates a set of CacheTriple objects, together with their associated metadata combination represented as a MetaCacheKey (see section 4.7.2.6). As elaborated in section 4.7.1.2, utilizing SPARQL extraction queries to retrieve the necessary source metadata proved unfeasible. Therefore, the MetaCacheSourceAnalyzer relies on the NTripleStatementIterator class to efficiently iterate over individual source RDF statements. As was the case for the SIM, this means the component currently only supports RDF data in N-Triple format.

4.7.2.3 CacheManager

The CacheManager class represents the access point for the cache, and is contacted to add and retrieve cached data. Behind the scenes, the CacheManager automatically performs memory management tasks whenever the cache exceeds the maximum defined space, in which case removal elements are persistently stored or removed. Different CacheManager instances can be created to manage multiple caching systems (e.g., see section 4.7.3). Figure 4-15 shows the diagram for the CacheManager class.

[80] The inverse is not guaranteed; namely, that two different CacheKeys return different hash codes (this is in line with the contract of the Java Object hashCode method).

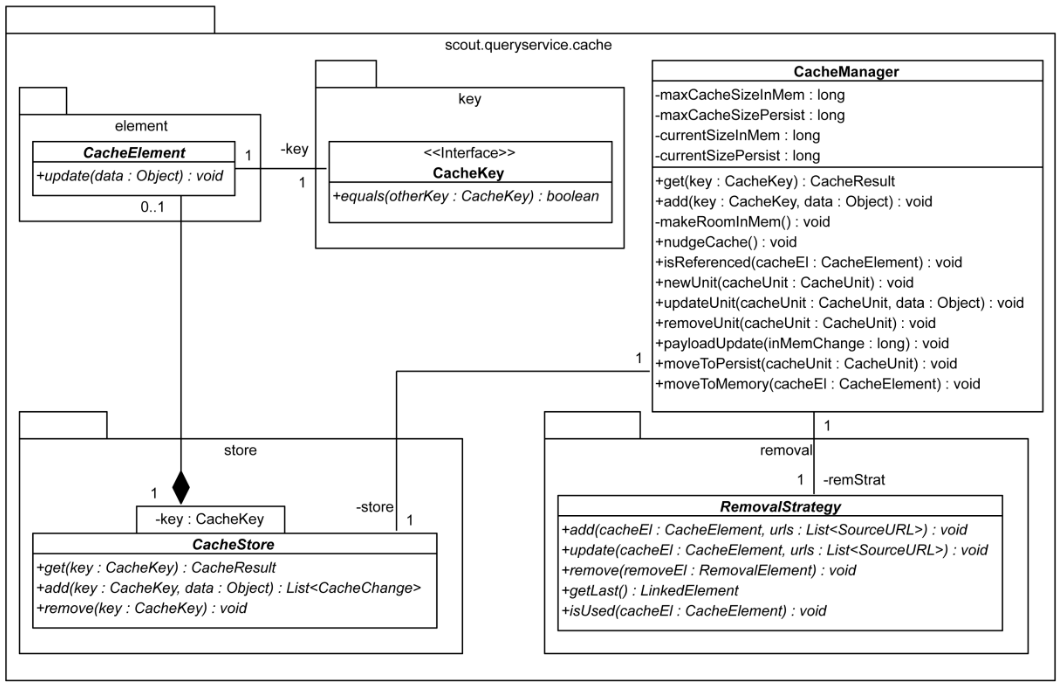

scout.queryservice.cache
element
CacheElement
+update(data : Object) : void
0..1
1
-key
1
key
<<Interface>>
CacheKey
+equals(otherKey : CacheKey) : boolean
CacheManager
-maxCacheSizeInMem : long
-maxCacheSizePersist : long
-currentSizeInMem : long
-currentSizePersist : long
+get(key : CacheKey) : CacheResult
+add(key : CacheKey, data : Object) : void
-makeRoomInMem() : void
+nudgeCache() : void
+isReferenced(cacheEl : CacheElement) : void
+newUnit(cacheUnit : CacheUnit) : void
+updateUnit(cacheUnit : CacheUnit, data : Object) : void
+removeUnit(cacheUnit : CacheUnit) : void
+payloadUpdate(inMemChange : long) : void
+moveToPersist(cacheUnit : CacheUnit) : void
+moveToMemory(cacheEl : CacheElement) : void
1
1
store
1
-key : CacheKey
-store
1
CacheStore
+get(key : CacheKey) : CacheResult
+add(key : CacheKey, data : Object) : List<CacheChange>
+remove(key : CacheKey) : void
removal
1
-remStrat
RemovalStrategy
+add(cacheEl : CacheElement, urls : List<SourceURL>) : void
+update(cacheEl : CacheElement, urls : List<SourceURL>) : void
+remove(removeEl : RemovalElement) : void
+getLast() : LinkedElement
+isUsed(cacheEl : CacheElement) : void

Figure 4-15. *Query Service - Cache*: CacheManager class.

CacheManager relies on the CacheStore class (see section 4.7.2.4) for storing and retrieving cached data. In case storage space becomes full, the CacheManager utilizes a particular RemovalStrategy subclass (see section 4.7.2.7) to identify suitable cache elements to be stored persistently or removed. The maximum memory and persistent storage space is configurable, to suit the capabilities of the mobile device. Below, we elaborate on how the add and retrieve operations are realized, and how memory management is performed during these two operations. Then, we discuss how the sizes of currently used storage (in-memory/persistent) are estimated and kept up-to-date.

When adding data (see add method), the new payload is accompanied by a CacheKey, uniquely identifying the data. Firstly, CacheManager passes the new data to the CacheStore for storage. Any newly created or updated CacheElements are then communicated to the configured RemovalStrategy (see add method), allowing the strategy to update its internal structures (see section 4.7.2.7). Afterwards, the CacheManager checks whether the new in-memory payload caused the in-memory storage limit to be exceeded. If so, CacheManager contacts the configured RemovalStrategy to obtain the “least” removal element (i.e., having the lowest removal value; see getLast method). To facilitate memory management, this removal element is returned as part of a linked list, sorted on removal value. In this linked list, each RemovalElement object is wrapped by a LinkedElement instance, keeping references to the next (i.e., “better”) and previous (i.e., “lesser”) RemovalElement object. This list allows CacheManager to easily iterate over the removal elements, storing them persistently or removing them until enough space has been cleared. CacheManager relies on the RemovalElement interface to perform these storage (moveToPersist method) and removal (remove method) operations.

In order to retrieve data (see get method), CacheManager is contacted with a CacheKey, identifying the requested data. Subsequently, the CacheStore is contacted to retrieve the related CacheElement (if any) and return its concrete payload. In addition, CacheManager indicates to the RemovalStrategy that the retrieved CacheElement was referenced (e.g., LRU utilizes this knowledge to calculate removal values; see isUsed method). It should be noted that retrieved cache elements are either in-memory or stored persistently, depending on previous memory management operations. In case a persistent CacheElement is retrieved, any stored data is automatically loaded into memory (see section 4.7.2.1). Consequently, to keep the memory space below the configured limit, memory management should also be performed after each retrieval. However, this may have undesired consequences. In our case, cached data is fetched per query triple pattern, in order to resolve queries not solvable by any single source. After all triple patterns have been handled, the retrieved payloads are added to the query dataset. However, if memory management was applied after individual retrievals, previously retrieved payloads may have already been stored persistently or removed (which involves destroying the in-memory payload) before they could be used by client code (e.g., added to the query dataset). For this purpose, CacheManager allows client code to indicate when retrieved data is no longer needed (e.g., after query execution; see nudgeCache method). After invoking this method, memory management will be performed. While this mechanism increases flexibility, it also allows the in-memory storage limit to be exceeded during data retrieval, possibly leading to out-of-memory errors. Responsibility to utilize this mechanism correctly lies with the client code.

The sizes of the currently used in-memory and persistent storage are dynamically kept up-to-date. CacheUnit subclasses (e.g., CacheTripleSet for Meta Cache; see section 4.7.2.6) are responsible for estimating their payload size in bytes. Each time an operation is performed influencing memory / storage space, these subclasses contact CacheManager (e.g., see newUnit, updateUnit methods), which then requests their new payload size to update the currently used storage sizes. Since the Android API does not supply tools to efficiently and accurately measure object memory usage at runtime (such as e.g., the Java Instrumentation API), the actual payload size is estimated. This estimated size includes the size of concrete payload, such as character arrays and integers, but not the object overheads of the CacheUnit objects (e.g., CacheTripleSet) or payload objects (e.g., CacheTriple). As we discuss in the evaluation chapter (see Chapter 5, section 5.3.2), this estimation corresponds only roughly to the actual memory usage. Devising a better way to accurately estimate memory usage is future work.

4.7.2.4 CacheStore

The CacheStore class is responsible for storing and retrieving cached data, and implements a particular cache organization for efficient and fine-grained retrieval. In Figure 4-16, we show the CacheStore class, together with its subclasses.

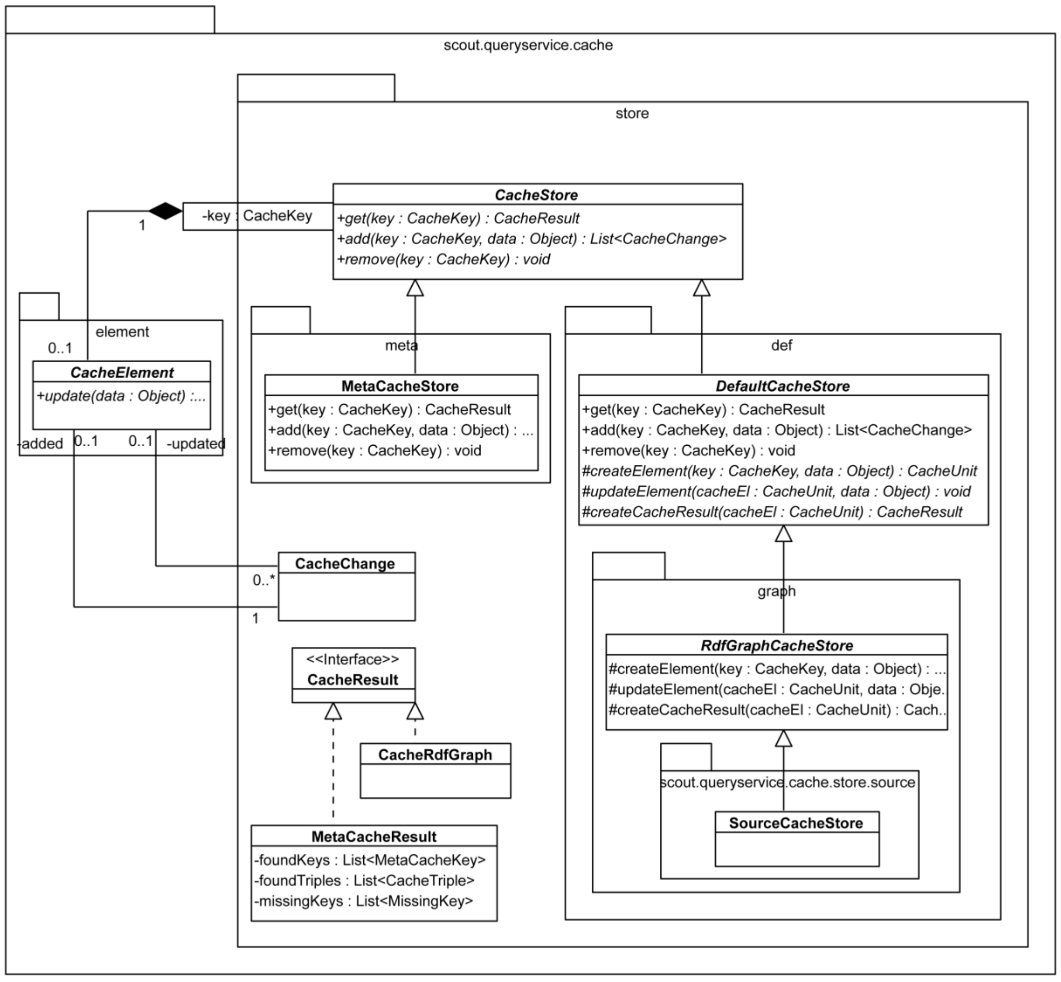

scout.queryservice.cache
store
CacheStore
+get(key : CacheKey) : CacheResult
+add(key : CacheKey, data : Object) : List<CacheChange>
+remove(key : CacheKey) : void
-key : CacheKey
1
element
0..1
CacheElement
+update(data : Object) :...
-added 0..1 0..1 -updated
meta
MetaCacheStore
+get(key : CacheKey) : CacheResult
+add(key : CacheKey, data : Object) : ...
+remove(key : CacheKey) : void
def
DefaultCacheStore
+get(key : CacheKey) : CacheResult
+add(key : CacheKey, data : Object) : List<CacheChange>
+remove(key : CacheKey) : void
#createElement(key : CacheKey, data : Object) : CacheUnit
#updateElement(cacheEl : CacheUnit, data : Object) : void
#createCacheResult(cacheEl : CacheUnit) : CacheResult
CacheChange
0..*
1
<<Interface>>
CacheResult
CacheRdfGraph
MetaCacheResult
-foundKeys : List<MetaCacheKey>
-foundTriples : List<CacheTriple>
-missingKeys : List<MissingKey>
graph
RdfGraphCacheStore
#createElement(key : CacheKey, data : Object) : ...
#updateElement(cacheEl : CacheUnit, data : Obje...
#createCacheResult(cacheEl : CacheUnit) : Cach...
scout.queryservice.cache.store.source
SourceCacheStore

Figure 4-16. *Query Service - Cache*: CacheStore package.

A CacheKey identifies data to be added, retrieved or removed from the CacheStore. In case the key to be added already exists, the associated cache element is updated with the new payload. Else, a new cache element will be created. A CacheChange object identifies the changes in the cache resulting from an add operation, and includes either the updated cache element or the newly created element. After a retrieval operation, the found payload (if any) is returned in the form of a CacheResult object. In case of Meta Cache, this result also includes sources to be re-downloaded, in the form of MissingKey objects (see section 4.7.2.6). Finally, it should be noted that, for performance reasons, none of the cache stores currently check for duplicate triples.

4.7.2.5 SourceCacheStore

The SourceCache class implements the Source Cache organization (see section 4.5.1.1) and organizes the cached data via its origin source. Most of its functionality is implemented via generic super classes. Figure 4-16 (see previous section) shows the inheritance hierarchy for SourceCacheStore.

The DefaultCacheStore class is a straightforward implementation of CacheStore, utilizing a hashmap to map cache keys to their corresponding cache elements. For certain operations, such as creating and updating the kept cache elements, it relies on its subclasses (see abstract methods). The RdfGraphCacheStore class inherits from DefaultCacheStore, and utilizes the CacheRdfGraph class (see Figure 4-14) to serve as cache element. Finally, the SourceCacheStore class is a concrete subclass of RdfGraphCacheStore, where each CacheRdfGraph element will keep RDF data originating from a specific source. A SourceCacheKey object uniquely identifies a CacheRdfGraph instance with data originating from a specific online source.

In order to store the RDF graph, a CacheRdfGraph keeps an RdfGraph subclass instance (see Figure 4-8) as payload. In particular, it keeps an AndrojenaRdfGraph object, which utilizes the Androjena library to locally handle and query RDF data. This library was extended to optimize the combining of RDF graphs, significantly reducing the overhead of assembling individual, cached sources into a single query dataset. We shortly revisit this issue in the evaluation chapter (see Chapter 5, section 5.3.2).

4.7.2.6 MetaCacheStore

The MetaCacheStore organizes the source data via its shared metadata (i.e., predicate, subject types, object types), and implements the Meta Cache organization (see section 4.5.1.2).
Figure **4-17.** *Query Service - Cache*: MetaCache package.
 shows the MetaCacheStore package. Below, we first elaborate on the keys and different types of elements in the MetaCacheStore. Then, we discuss the retrieval and add operations.

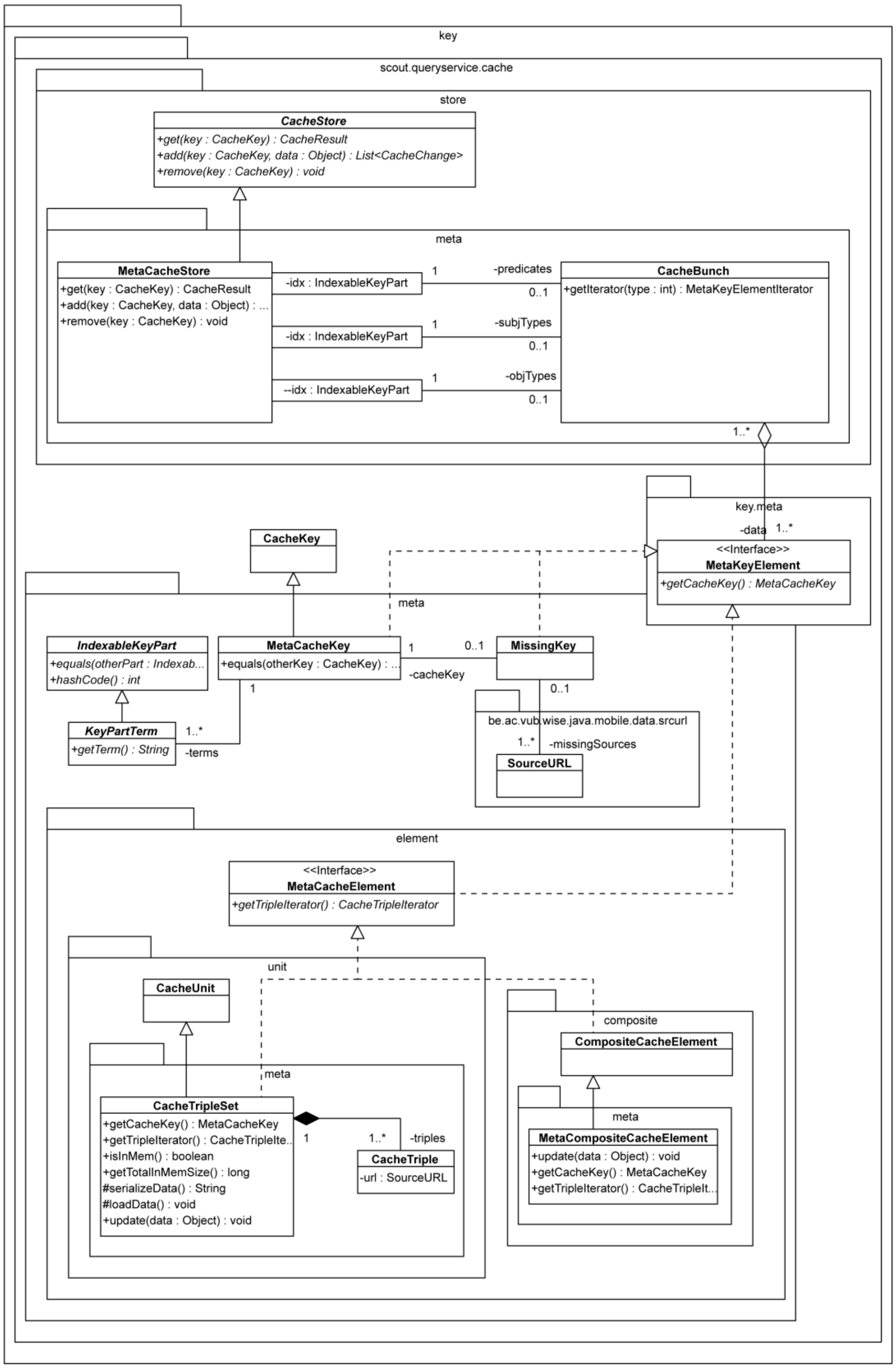

key
scout.queryservice.cache
store
CacheStore
+get(key : CacheKey) : CacheResult
+add(key : CacheKey, data : Object) : List<CacheChange>
+remove(key : CacheKey) : void
meta
MetaCacheStore
+get(key : CacheKey) : CacheResult
+add(key : CacheKey, data : Object) : ...
+remove(key : CacheKey) : void
-idx : IndexableKeyPart
-idx : IndexableKeyPart
--idx : IndexableKeyPart
1
-predicates
0..1
1
-subjTypes
0..1
1
-objTypes
0..1
CacheBunch
+getIterator(type : int) : MetaKeyElementIterator
1..*
key.meta
-data 1..*
<<Interface>>
MetaKeyElement
+getCacheKey() : MetaCacheKey
CacheKey
meta
IndexableKeyPart
+equals(otherPart : Indexab...
+hashCode() : int
MetaCacheKey
+equals(otherKey : CacheKey) : ...
1
0..1
-cacheKey
MissingKey
0..1
1
KeyPartTerm
+getTerm() : String
1..*
-terms
be.ac.vub.wise.java.mobile.data.srcurl
1..*
-missingSources
SourceURL
element
<<Interface>>
MetaCacheElement
+getTripleIterator() : CacheTripleIterator
unit
CacheUnit
meta
CacheTripleSet
+getCacheKey() : MetaCacheKey
+getTripleIterator() : CacheTripleIte...
+isInMem() : boolean
+getTotalInMemSize() : long
#serializeData() : String
#loadData() : void
+update(data : Object) : void
1
1..*
-triples
CacheTriple
-url : SourceURL
composite
CompositeCacheElement
meta
MetaCompositeCacheElement
+update(data : Object) : void
+getCacheKey() : MetaCacheKey
+getTripleIterator() : CacheTripleIt...

Figure 4-17. *Query Service - Cache*: MetaCache package.

Keys and elements

A MetaCacheKey object stands for a particular metadata combination, and is used to uniquely identify cached data. In more detail, a MetaCacheKey object keeps a set of IndexableKeyPart instances, each representing an individual piece of metadata. Via these IndexableKeyPart instances, internal indices perform a mapping from metadata parts to associated cached data (see below). KeyPartTerm, subclass of IndexableKeyPart, encapsulates RDF terms (i.e., predicates and types) and overrides the hashCode method to ensure unique hashing (i.e., two identical metadata parts return the same hash code), and the equals method for correct comparison. As the KeyPartTerm class wraps data in our mobile query service, it has a concrete subclass performing dictionary encoding (see section 4.7.1.1).

The MetaKeyElement interface stands for any element identified by a MetaCacheKey. This includes cache elements (MetaCacheElement class), as well as elements identifying previously removed source data (MissingKey class) and MetaCacheKey objects themselves. MetaCacheElement objects keep RDF triples sharing the same metadata, which is reflected by its declared methods to access RDF triples (e.g., getTriplesIterator). The CacheTripleSet class implements the MetaCacheElement interface and is a concrete subclass of CacheUnit (see section 4.7.2.1), making it the default unit of storage, removal and retrieval in Meta Cache. A CacheTripleSet object keeps a set of cached RDF triples in the form of CacheTriple instances. To reduce the in-memory storage space, the CacheTriple class (not all fields shown) also applies dictionary encoding. The MissingKey class stands for previously removed cached data, keeping the MetaCacheKey identifying the removed data, as well as the SourceURLs from which the data originated. Below, we discuss the rationale behind the MissingKey class.

Missing keys are an accurate and efficient way of determining which sources need to be re-downloaded to serve an information request. As an alternative, the Source Index Model could be employed to identify sources previously removed from the cache. Specifically, given a metadata combination, missing sources could be identified by subtracting the relevant sources found in Meta Cache from the total set of found SIM sources. However, relying on missing keys avoids the overhead of keeping an extra index data structure. Furthermore, missing keys stand for a more accurate way of identifying sources that contain removed cached data. As previously mentioned, it is possible that in specific cases, sources identified by the SIM do not actually contain query-relevant data; due to the fact metadata combinations are indexed on a per-source level (for more details, see section 4.4.1). Consequently, missing keys enable us to further reduce the number of source downloads (see second challenge, minimizing source downloads).

In Figure 4-18, we show an example Meta Cache structure. To enable fast retrieval of cached data, the MetaCacheStore keeps internal indices on each metadata part (i.e., predicate, subject type and object type). Compared to related work on RDF stores, this schema-level indexing achieves a balance between retrieval speed and memory overhead; see Chapter 2, section 2.2.8.3 for a discussion. Specifically, these indices map from an IndexableKeyPart to a CacheBunch instance, which collects

MetaKeyElements (i.e., MetaCacheElements or MissingKeys) of which the associated MetaCacheKey keeps a specific indexed metadata part. For instance, in Figure 4-18, the foaf:knows metadata term is connected to a CacheBunch keeping a MetaCacheElement and MissingKey, each having the foaf:knows term in their MetaCacheKey.

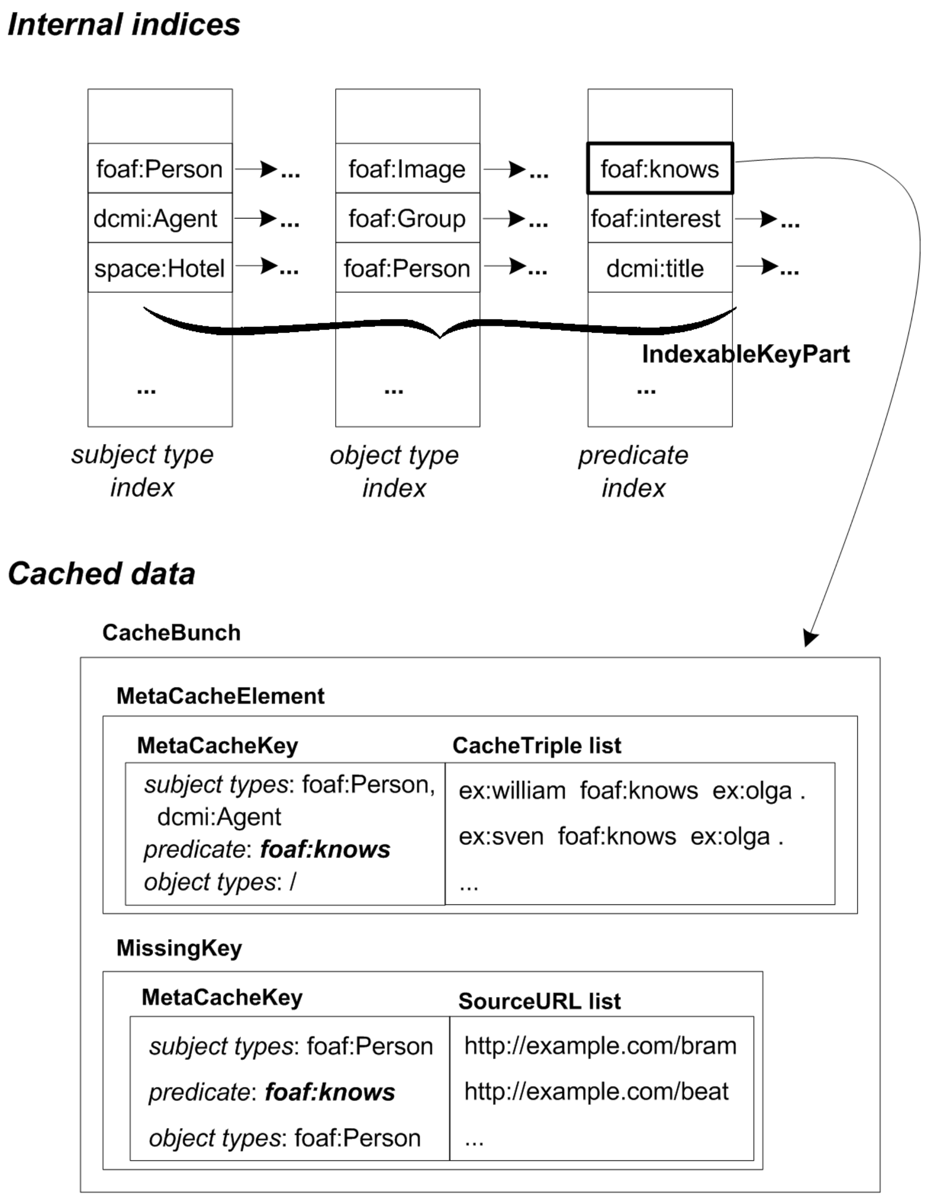


Figure 4-18. *Query Service – Cache*: Meta Cache structure.

Retrieve operation

When performing the search operation, one of the aforementioned indices is employed to retrieve a set of matching MetaKeyElement instances, whereby the used index depends on the supplied metadata parts (i.e., specified by the query). Each of the returned MetaKeyElement objects hereby adheres to a single metadata part in the search key (i.e., predicate, subject/object type). Afterwards, the MetaCacheKey associated with each found MetaKeyElement is compared to the entire search key. In this comparison, the cached MetaCacheKeys need not exactly match the search key; while the predicates need to be identical, the subject/object types of the search key need to be a (non-strict) subset of the cached key types. For instance, in case a search key looks for RDF triples with subject type foaf:Person, cached triples with subject types foaf:Person and dcmi:Agent are also returned (as their subject is also a foaf:Person). Consequently, a search may return multiple matching MetaKeyElement instances. After the search operation, we obtain a set of MetaCacheElement and MissingKey instances, whereby the found MetaCacheKeys and MissingKeys, together with the wrapped CacheTriples, are finally returned as a result.

It should be noted that the retrieved triples do not include the original source triples declaring types for the RDF resources. However, for completeness, these type triples should also be returned; for instance, in our case, they are required to successfully resolve the original query. Storing these type triples in the cache) would significantly increase storage space. Initially, these type triples were automatically generated and returned as extra CacheTriples, based on the retrieved triples and types found in associated MetaCacheKeys. However, this resulted in a significant performance overhead, since this required iterating over each returned cache triple and generating corresponding type triples. Consequently, the result was simply extended to include related MetaCacheKeys for the returned triples (see MetaCacheResult class, Figure 4-16), thus supplying the types associated with each returned RDF resource. Client code can then utilize these resource types as they see fit. In our case, we extended the Androjena library to efficiently insert resource types into the query graph.

Add operation

In order to add new source data to the cache, the payload and its identifying MetaCacheKey are passed to the MetaCacheStore. The add operation either leads to new cache elements being created or existing ones being updated, depending on whether the particular MetaCacheKey already exists in the cache. To perform this check, the search key is exactly matched to cached keys, whereby their predicates as well as subject/object type sets need to be identical (in contrast to regular search, see above). In case no match is found, a new cache element is created and loaded with the new payload. Else, the matching cache element is updated with the new payload.

Finally, it is possible source data from the new payload's origin source(s) had previously been added to the cache, and removed in order to clear storage space. Any missing keys, matching the new MetaCacheKey and the new payload's origin source(s), are removed from the cache, since the indicated data is no longer missing from the cache.

4.7.2.7 RemovalStrategy

The RemovalStrategy class identifies certain cached data (e.g., unlikely to be referenced again), which will be stored persistently or removed to clear space. Figure 4-19 shows the RemovalStrategy package. Below, we elaborate on the main classes.

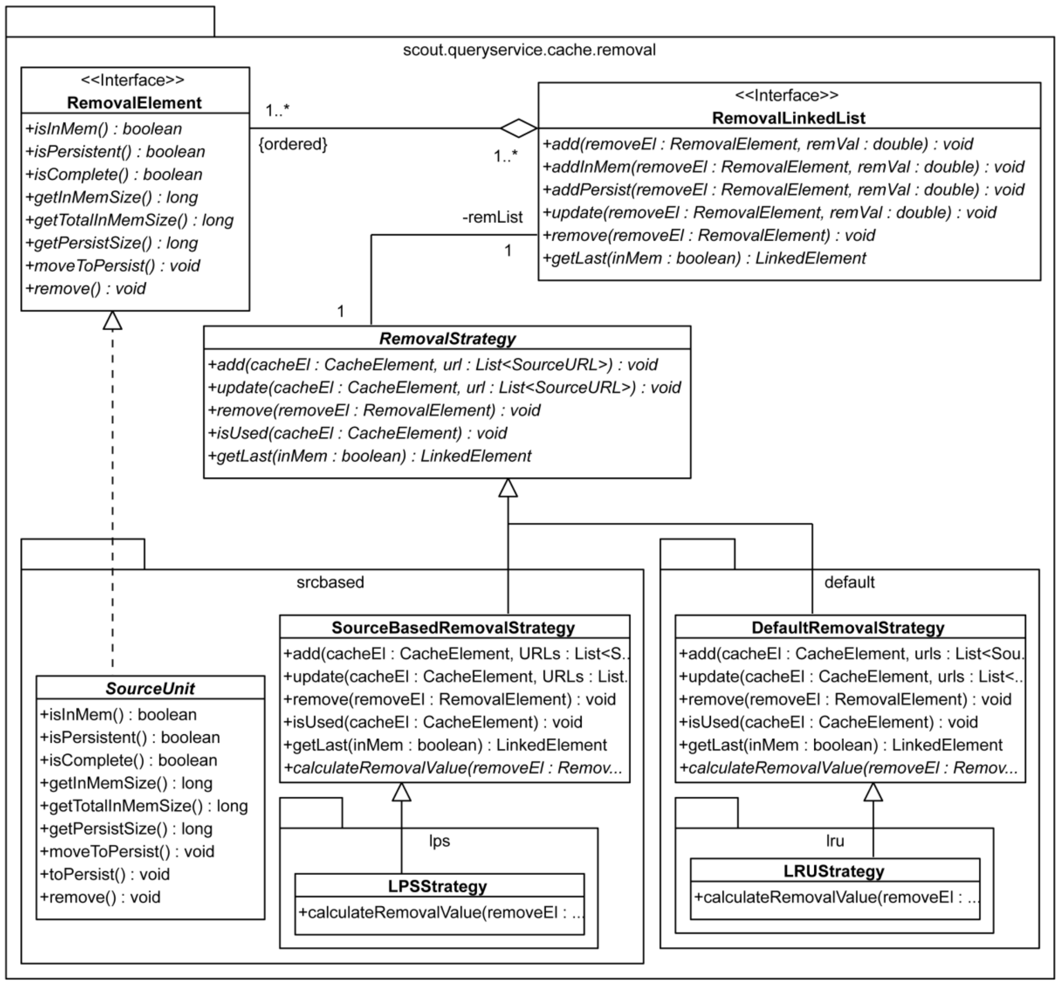

scout.queryservice.cache.removal
<<Interface>>
RemovalElement
+isInMem() : boolean
+isPersistent() : boolean
+isComplete() : boolean
+getInMemSize() : long
+getTotalInMemSize() : long
+getPersistSize() : long
+moveToPersist() : void
+remove() : void
1..*
{ordered}
1..*
<<Interface>>
RemovalLinkedList
+add(removeEl : RemovalElement, remVal : double) : void
+addInMem(removeEl : RemovalElement, remVal : double) : void
+addPersist(removeEl : RemovalElement, remVal : double) : void
+update(removeEl : RemovalElement, remVal : double) : void
+remove(removeEl : RemovalElement) : void
+getLast(inMem : boolean) : LinkedElement
-remList
1
1
RemovalStrategy
+add(cacheEl : CacheElement, url : List<SourceURL>) : void
+update(cacheEl : CacheElement, url : List<SourceURL>) : void
+remove(removeEl : RemovalElement) : void
+isUsed(cacheEl : CacheElement) : void
+getLast(inMem : boolean) : LinkedElement
srcbased
SourceBasedRemovalStrategy
+add(cacheEl : CacheElement, URLs : List<S...
+update(cacheEl : CacheElement, URLs : List...
+remove(removeEl : RemovalElement) : void
+isUsed(cacheEl : CacheElement) : void
+getLast(inMem : boolean) : LinkedElement
+calculateRemovalValue(removeEl : Remov...
SourceUnit
+isInMem() : boolean
+isPersistent() : boolean
+isComplete() : boolean
+getInMemSize() : long
+getTotalInMemSize() : long
+getPersistSize() : long
+moveToPersist() : void
+toPersist() : void
+remove() : void
lps
LPSStrategy
+calculateRemovalValue(removeEl : ...
default
DefaultRemovalStrategy
+add(cacheEl : CacheElement, urls : List<Sou...
+update(cacheEl : CacheElement, urls : List<...
+remove(removeEl : RemovalElement) : void
+isUsed(cacheEl : CacheElement) : void
+getLast(inMem : boolean) : LinkedElement
+calculateRemovalValue(removeEl : Remov...
lru
LRUStrategy
+calculateRemovalValue(removeEl : ...

Figure 4-19. *Query Service - Cache*: RemovalStrategy package.

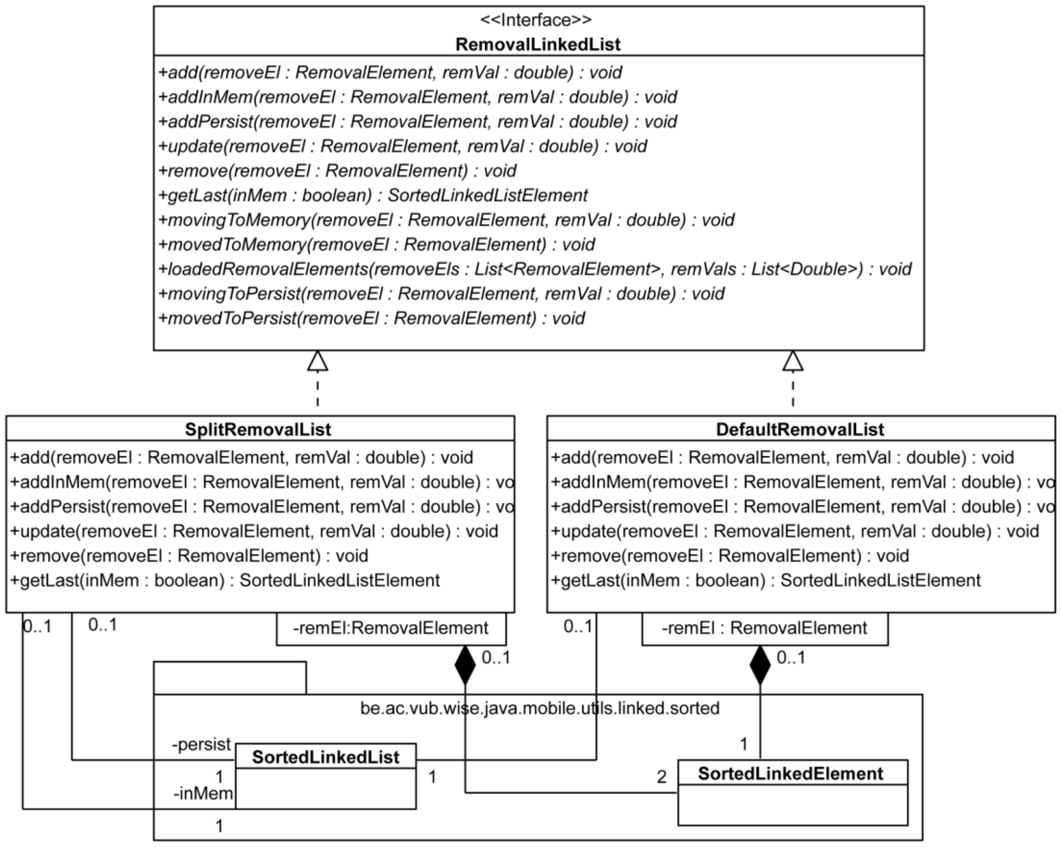
The RemovalElement interface represents the unit of removal in the cache. RemovalElement declares methods enabling memory management; for moving data to persistent storage (moveToPersist method) and removing the data entirely (remove method). As mentioned, in most cases, the unit of removal will be equal to the storage unit (represented by the CacheUnit class). To accommodate such cases, the CacheUnit class implements the RemovalElement interface (see section 4.7.2.1). When the Least-Recently-Used (LRU) strategy is applied to Meta Cache, the CacheTripleSet class (subclass of CacheUnit) will stand for both the unit of storage and removal. In contrast, the Least-Popular-Sources (LPS) removal strategy requires both units to be decoupled, whereby the unit of removal equals the origin source of cached data. In this case, the SourceUnit class (implementing the RemovalElement interface) serves as removal unit.

The RemovalStrategy class is responsible for calculating replacement (or removal) values for each RemovalElement. In addition, RemovalStrategy keeps a RemovalLinkedList instance, which represents a linked list of RemovalElements sorted on their removal value. In case not enough space is available, the CacheManager component requests the “least” (i.e., last) RemovalElement wrapped as a LinkedElement (see section 4.7.2.3). As new cache elements are created, updated, removed or referenced, CacheManager notifies the configured RemovalStrategy (see add, update, remove and isUsed methods), allowing it to re-calculate removal values and update its RemovalLinkedList.

Below, we elaborate on the RemovalLinkedList implementation (section 4.7.2.8), as well as the implementation of the two concrete RemovalStrategy subclasses, namely LRUStrategy (section 4.7.2.9) and LPSStrategy (section 4.7.2.10).

4.7.2.8 RemovalLinkedList

The RemovalLinkedList interface represents a linked list of removal elements, sorted on the calculated removal values. Figure 4-20 shows the RemovalLinkedList interface, together with its two implementing classes (not all subclass methods shown).



<<Interface>>
RemovalLinkedList
+add(removeEl : RemovalElement, remVal : double) : void
+addInMem(removeEl : RemovalElement, remVal : double) : void
+addPersist(removeEl : RemovalElement, remVal : double) : void
+update(removeEl : RemovalElement, remVal : double) : void
+remove(removeEl : RemovalElement) : void
+getLast(inMem : boolean) : SortedLinkedListElement
+movingToMemory(removeEl : RemovalElement, remVal : double) : void
+movedToMemory(removeEl : RemovalElement) : void
+loadedRemovalElements(removeEls : List<RemovalElement>, remVals : List<Double>) : void
+movingToPersist(removeEl : RemovalElement, remVal : double) : void
+movedToPersist(removeEl : RemovalElement) : void
SplitRemovalList
+add(removeEl : RemovalElement, remVal : double) : void
+addInMem(removeEl : RemovalElement, remVal : double) : vo
+addPersist(removeEl : RemovalElement, remVal : double) : vo
+update(removeEl : RemovalElement, remVal : double) : void
+remove(removeEl : RemovalElement) : void
+getLast(inMem : boolean) : SortedLinkedListElement
-remEl:RemovalElement
DefaultRemovalList
+add(removeEl : RemovalElement, remVal : double) : void
+addInMem(removeEl : RemovalElement, remVal : double) : vo
+addPersist(removeEl : RemovalElement, remVal : double) : vo
+update(removeEl : RemovalElement, remVal : double) : void
+remove(removeEl : RemovalElement) : void
+getLast(inMem : boolean) : SortedLinkedListElement
-remEl : RemovalElement
0..1
0..1
0..1
0..1
0..1
be.ac.vub.wise.java.mobile.utils.linked.sorted
-persist
1
-inMem
1
SortedLinkedList
1
1
2
SortedLinkedElement

Figure 4-20. *Query Service – Cache*: RemovalLinkedList.

Both subclasses rely on the SortedLinkedList and SortedLinkedElement classes. In order to efficiently implement the update and remove methods, which require updating the kept linked list(s), each RemovalLinkedList subclass keeps a mapping from RemovalElement instances to SortedLinkedElement objects.

The DefaultRemovalList subclass represents a straightforward implementation. This class keeps a single SortedLinkedList instance, containing all RemovalElements packaged as SortedLinkedElement objects. However, we observed this approach incurs a serious drawback during memory management, when the CacheManager component iterates over the linked list to clear space (see section 4.7.2.3). More specifically, each time memory space needs to be cleared, CacheManager needs to iterate over both in-memory and persistently stored removal elements, although operations are only applied on in-memory elements (vice versa when clearing persistent storage space). Therefore, many irrelevant elements are iterated, yielding large performance overhead.

By separating in-memory and persistently stored elements into two different lists, the currently utilized linked list (e.g., keeping in-memory elements) only contains removal elements relevant to the current task (e.g., clearing memory space). This solution is implemented by the SplitRemovalList subclass, which keeps two separate SortedLinkedList instances, respectively containing in-memory and persistent removal elements. A drawback of this approach is that both lists need to be carefully kept up-to-date. The required housekeeping is implemented by the extra methods movingToMemory, movedToMemory, movingToPersist and movedToPersist in the RemovalStrategy and RemovalLinkedList classes. For instance, the movingToMemory method needs to be invoked right before a cache element is loaded into memory; in case its associated removal element(s) is not yet in-memory, the removal element is added to the in-memory list. Aside from the extra complexity caused by this housekeeping, the continuous moving of removal elements between the two linked lists incurs insertion and removal costs. We observed during experiments that this leads to a large performance overhead during the Data Query phase, where large amounts of elements are loaded in memory and thus moved from the persistent to the in-memory list. To avoid this overhead slowing down query execution, an extra loadedRemovalElements method was introduced, which is called after query execution and indicates that the passed removal elements have been loaded into memory. Both the LRU and LPS strategy implementations utilize the SplitRemovalList subclass.

4.7.2.9 LRUStrategy

LRUStrategy (diagram not shown) realizes the well-known Least-Recently-Used (LRU) removal strategy. LRU estimates the likelihood of data being required in the future, by assuming recently referenced items will likely be referenced again (temporal locality).

LRUStrategy is a subclass of DefaultRemovalStrategy. In these removal strategies, the same unit is employed for storage, removal and retrieval (i.e., CacheUnit class). The DefaultRemovalStrategy superclass implements most operations, deferring the removal value calculation to the subclasses. The LRUStrategy subclass performs this calculation based on the number of references made to each cache element.

4.7.2.10 LPSStrategy

The LPSStrategy class implements the Least-Popular-Sources (LPS) strategy presented in this thesis. This strategy considers the source’s popularity, reflected by 1/ the popularity of its source data, i.e., the number of stored cache units containing the data, and 2/ the popularity of its own metadata, i.e., how often the contained metadata is found in other sources. Additionally, the download cost of the associated online sources is taken into account. Figure 4-21 shows the LPSStrategy package.

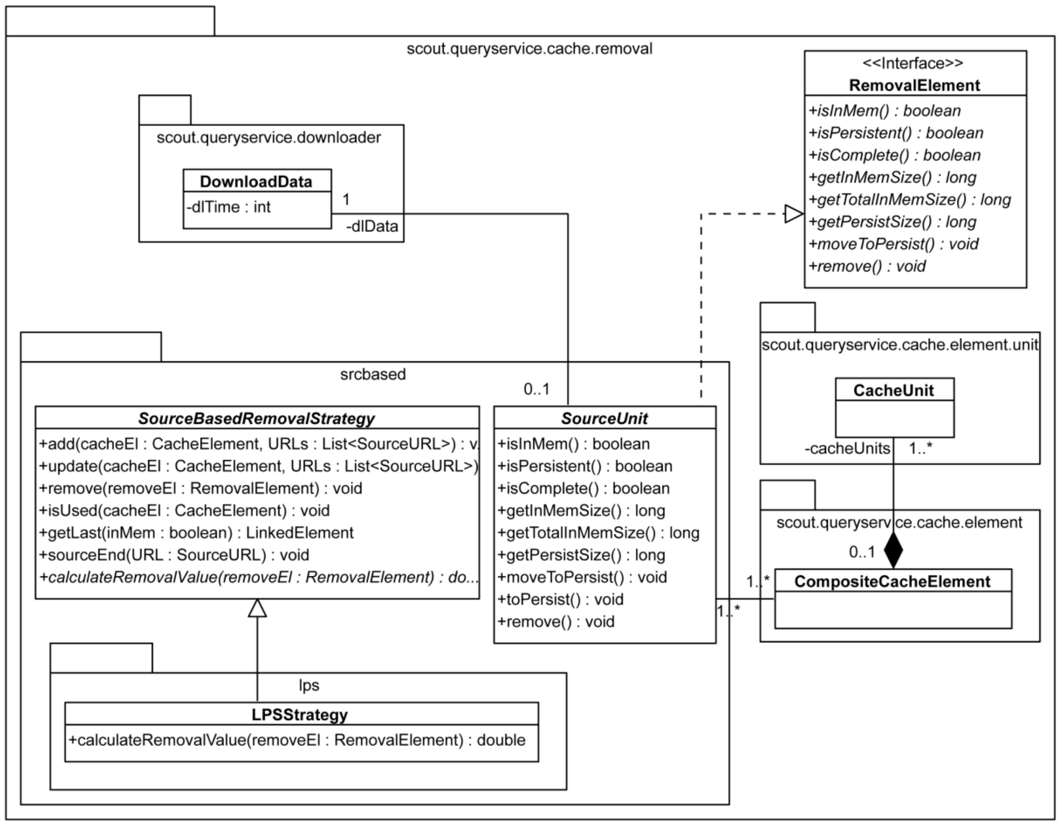

scout.queryservice.cache.removal
scout.queryservice.downloader
DownloadData
-dlTime : int
1
-dlData
<<Interface>>
RemovalElement
+isInMem() : boolean
+isPersistent() : boolean
+isComplete() : boolean
+getInMemSize() : long
+getTotalInMemSize() : long
+getPersistSize() : long
+moveToPersist() : void
+remove() : void
scout.queryservice.cache.element.unit
CacheUnit
-cacheUnits
1..*
srcbased
0..1
SourceBasedRemovalStrategy
+add(cacheEl : CacheElement, URLs : List<SourceURL>) : v
+update(cacheEl : CacheElement, URLs : List<SourceURL>)
+remove(removeEl : RemovalElement) : void
+isUsed(cacheEl : CacheElement) : void
+getLast(inMem : boolean) : LinkedElement
+sourceEnd(URL : SourceURL) : void
+calculateRemovalValue(removeEl : RemovalElement) : do...
SourceUnit
+isInMem() : boolean
+isPersistent() : boolean
+isComplete() : boolean
+getInMemSize() : long
+getTotalInMemSize() : long
+getPersistSize() : long
+moveToPersist() : void
+toPersist() : void
+remove() : void
scout.queryservice.cache.element
0..1
1..*
CompositeCacheElement
1..*
lps
LPSStrategy
+calculateRemovalValue(removeEl : RemovalElement) : double

Figure 4-21. *Query Service - Cache:* LPSStrategy package.

The LPSStrategy class is a subclass of the SourceBasedRemovalStrategy class. For source-based removal strategies, the *removal unit* equals the data's origin source, and is represented by the SourceUnit class. SourceUnit realizes the RemovalElement interface, and allows memory management operations (e.g., store data persistently, remove data) to be applied to all cached data from a specific origin source. Each SourceUnit also keeps a DownloadData object, encapsulating download data related to the origin source (e.g., download time). This information is used to calculate the LPS removal value.

In order to persistently store or remove all origin source data, we leverage the memory management support supplied by cache units, in particular those cache units that store the corresponding source data. However, these cache units likely keep data from other sources as well (e.g., CacheTripleSet instances typically contain RDF triples from multiple sources). To gain more flexibility, the *storage unit* is further decoupled from the *unit of retrieval*. In our setup, each individual CacheUnit keeps data originating from a single source, and suiting a specific CacheKey. Consequently, storing or removing a cache unit only influences data from one particular online source. At the same time, the

CompositeCacheElement class is employed as retrieval unit, grouping all CacheUnits adhering to the same CacheKey.

To realize its role as RemovalElement, the SourceUnit class keeps references to CompositeCacheElements wrapping related source data. For each RemovalElement method (e.g., moveToPersist, remove), the SourceUnit class iterates over each of its CompositeCacheElements, delegating the operation to only those cache units keeping its particular source's data. Regarding retrieval, the cache store locates CompositeCacheElements matching the search key.

It can thus be observed that, by separating the units of removal, storage and retrieval, data can be removed per origin source (or any other criteria); while data matching a given CacheKey can still be easily returned. The SourceBasedRemovalStrategy abstract class implements most methods, and delegates responsibility for calculating removal values towards subclasses (e.g., LPSStrategy). Finally, to efficiently implement update and remove operations, the SourceBasedRemovalStrategy class also keeps a map from SourceURLs to their corresponding SourceUnit, as well as from CacheElements to their associated SourceUnits.

4.7.2.11 PersistentStorage

In order to store its payload persistently, the CacheUnit class (and its subclasses) relies on the PersistentStorage class. Figure 4-22 shows the PersistentStorage package.

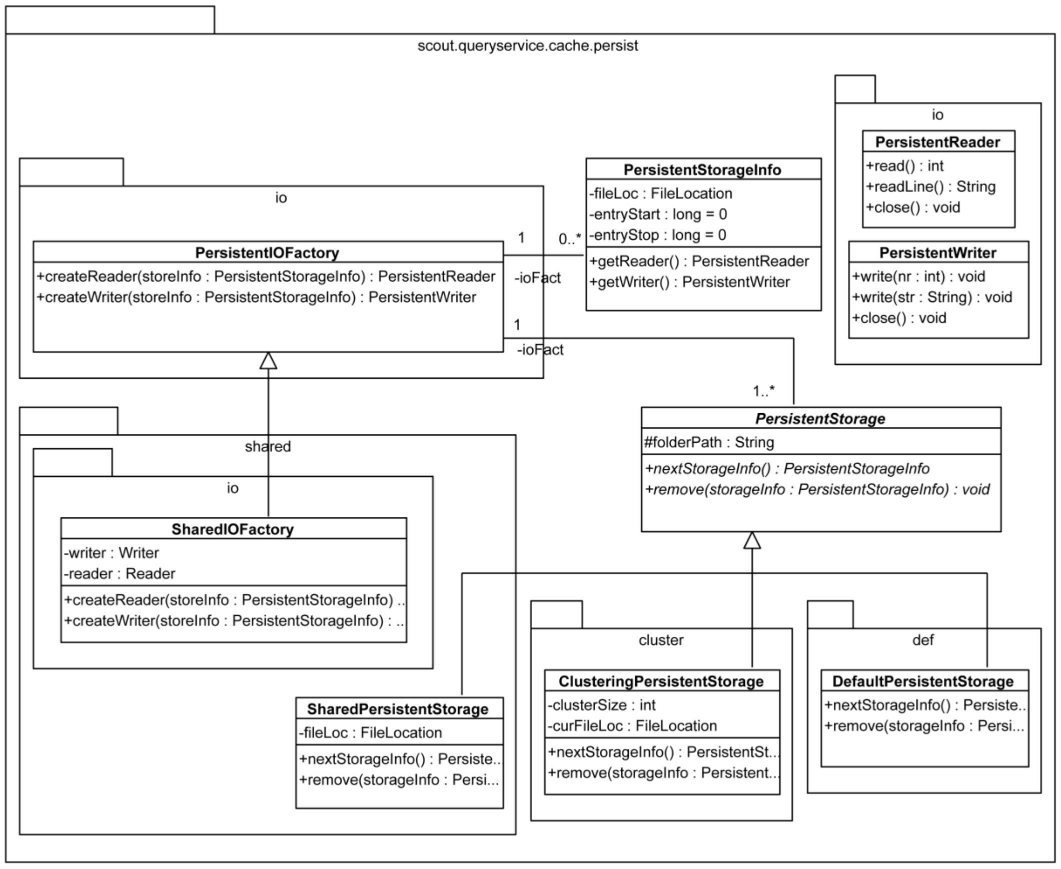

scout.queryservice.cache.persist
io
PersistentIOFactory
+createReader(storeInfo : PersistentStorageInfo) : PersistentReader
+createWriter(storeInfo : PersistentStorageInfo) : PersistentWriter
1
0..*
-ioFact
1
-ioFact
1..*
PersistentStorageInfo
-fileLoc : FileLocation
-entryStart : long = 0
-entryStop : long = 0
+getReader() : PersistentReader
+getWriter() : PersistentWriter
io
PersistentReader
+read() : int
+readLine() : String
+close() : void
PersistentWriter
+write(nr : int) : void
+write(str : String) : void
+close() : void
PersistentStorage
#folderPath : String
+nextStorageInfo() : PersistentStorageInfo
+remove(storageInfo : PersistentStorageInfo) : void
shared
io
SharedIOFactory
-writer : Writer
-reader : Reader
+createReader(storeInfo : PersistentStorageInfo) ..
+createWriter(storeInfo : PersistentStorageInfo) : ..
SharedPersistentStorage
-fileLoc : FileLocation
+nextStorageInfo() : Persiste..
+remove(storageInfo : Persi...
cluster
ClusteringPersistentStorage
-clusterSize : int
-curFileLoc : FileLocation
+nextStorageInfo() : PersistentSt..
+remove(storageInfo : Persistent..
def
DefaultPersistentStorage
+nextStorageInfo() : Persiste..
+remove(storageInfo : Persi...

Figure 4-22. *Query Service:* PersistentStorage package.

When storing their payload, cache units invoke the nextStorageInfo method of the configured PersistentStorage object, which returns a PersistentStorageInfo object defining the next available storage location. Importantly, PersistentStorageInfo supplies PersistentWriter and PersistentReader objects, which can respectively be used to easily write payload to the defined storage location, and read previously written payload.

Currently, the implementing classes of PersistentStorage all utilize the file system[81]. For the DefaultPersistentStorage subclass, each PersistentStorageInfo instance points to a single persistent file. It should however be noted that, in case of Meta Cache, there might be hundreds of thousands of cache units[82]. Since only a relatively small number is kept in memory (depending on the configurable in-memory limit), this means a huge amount of files will be stored on the file system, which may pose problems when clearing the persistent cache on a mobile device (e.g., when the mobile query service needs to be reset). The ClusteringPersistentStorage subclass groups payloads

[81] As payload data always needs to be entirely written / read by a CacheUnit, there is no need for a queryable persistent database (which incurs performance overhead).
[82] One for each individual metadata combination; in case the LPS removal strategy is applied, cache units are even further partitioned to store triples according to origin source.

into “cluster” files, up to a configurable amount. As such, it allows keeping the total number of persistent files in check.

During experiments, we noticed both storage mechanisms negatively impact performance in case of the Least-Popular-Sources (LPS) strategy, due to the separation of retrieval, storage and removal units. In case of LPS, when performing memory management operations, all cache units (storage unit) related to a particular source (removal unit) need to be stored or removed. For the aforementioned storage mechanisms, this means writing each separate payload to an individual file (DefaultPersistentStorage) or writing them sequentially to one or more cluster files (ClusteringPersistentStorage). When retrieving data, all cache units need to be loaded adhering to the search metadata combination (retrieval unit), meaning all their individual (cluster) files need to be loaded; leading to a large performance overhead.

In response, a third PersistentStorage subclass called SharedPersistentStorage was introduced, which allows arbitrarily grouping particular payloads into single files. For our purposes, grouping may occur via unit of retrieval (i.e., shared metadata) or removal (i.e., origin source). In case grouping occurs via retrieval unit, all payloads adhering to a particular metadata combination are stored in a single file. Concretely, this means a single retrieval only requires accessing one file, while memory management requires accessing each file keeping a metadata combination from a particular origin source. In case grouping is performed via removal unit, all payloads originating from the same source are stored in a single file. In this case, memory management only involves accessing the origin source’s persistent file, while a single retrieval involves accessing each file keeping source data adhering to the given metadata combination. Consequently, both clustering methods present a consideration between extra memory management and data retrieval overhead. In our evaluation chapter (see Chapter 5, section 5.4), we compare both clustering methods.

The clustering methods discussed above allow reading/writing operations to be confined to single files. To support this efficiently, PersistentReader and PersistentWriter objects can re-use the same Java FileReader/FileWriter behind the scenes. The SharedIOFactory abstract factory subclass re-uses the same FileReader and FileWriter objects for created PersistentReader and PersistentWriter objects, respectively.

4.7.3 Type mediation

In this section, we first review the implementation supporting the type mediation process (section 4.7.3.1). Then, we elaborate on the particular type mediation processes updating the SIM (section 4.7.3.2) and Meta Cache (section 4.7.3.3), respectively.

4.7.3.1 Supporting implementation

This section first discusses the additional memory overhead yielded by type mediation. Afterwards, we review the implementing package structure.

Type mediation necessitates keeping track of resource types across different data sources. This requires indexing the encountered resources, and keeping type information for each resource (as well as other data required by the particular type mediation process). It can be noted that indexing resources (i.e., instance-level) significantly increases memory overhead, and thus reduces the benefits of focusing on source metadata (i.e., schema-level) in the mobile query service. On the other hand, the developed resource index (see below) still consumes less memory space than the ones utilized by RDF stores. Since RDF stores need to efficiently return query-relevant RDF triples, the stores need to support multiple query access patterns, for instance resulting in a separate index for each triple pattern part (i.e., subject/predicate/object; Androjena) or multiple indices to cover each possible access pattern (resulting in 6 indices; YARS [71] and HexaStore [72]). In contrast, type mediation only requires a single index. Moreover, our resource index does not have to store blank nodes, since their identifiers are scoped locally and are not portable across RDF graphs[83].

In Figure 4-23, we show the general ResourceIndex package and related classes. Below, we shortly elaborate on these classes.

[83] http://www.w3.org/TR/rdf11-concepts/#section-blank-nodes (access date: 27/05/2013)

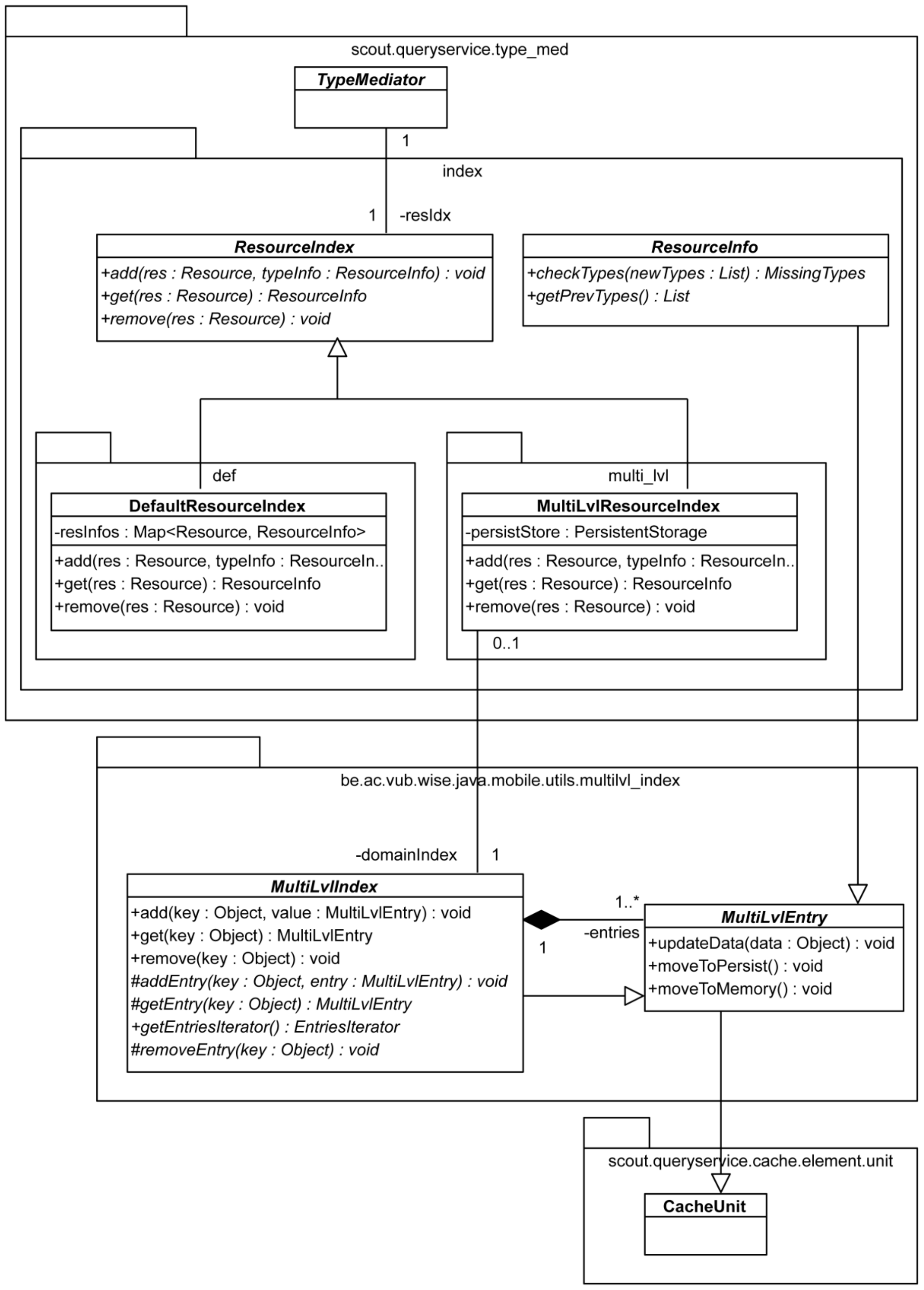

scout.queryservice.type_med
TypeMediator
1
index
1 -resIdx
ResourceIndex
+add(res : Resource, typeInfo : ResourceInfo) : void
+get(res : Resource) : ResourceInfo
+remove(res : Resource) : void
ResourceInfo
+checkTypes(newTypes : List) : MissingTypes
+getPrevTypes() : List
def
DefaultResourceIndex
-resInfos : Map<Resource, ResourceInfo>
+add(res : Resource, typeInfo : ResourceIn..
+get(res : Resource) : ResourceInfo
+remove(res : Resource) : void
multi_lvl
MultiLvlResourceIndex
-persistStore : PersistentStorage
+add(res : Resource, typeInfo : ResourceIn..
+get(res : Resource) : ResourceInfo
+remove(res : Resource) : void
0..1
be.ac.vub.wise.java.mobile.utils.multilvl_index
-domainIndex 1
MultiLvlIndex
+add(key : Object, value : MultiLvlEntry) : void
+get(key : Object) : MultiLvlEntry
+remove(key : Object) : void
#addEntry(key : Object, entry : MultiLvlEntry) : void
#getEntry(key : Object) : MultiLvlEntry
+getEntriesIterator() : EntriesIterator
#removeEntry(key : Object) : void
1..*
-entries
1
MultiLvlEntry
+updateData(data : Object) : void
+moveToPersist() : void
+moveToMemory() : void
scout.queryservice.cache.element.unit
CacheUnit

Figure 4-23. *Query Service – Type mediation:* ResourceIndex package.

The type mediation process is implemented via a TypeMediator subclass, and depends on the particular query service component. The ResourceIndex class indexes the required resource information, while the resource information itself is represented by the ResourceInfo class. Subclasses of ResourceInfo keep resource data required by the specific type mediation process (see following sections for more details). ResourceIndex subclasses are independent of the actual process; we discuss two subclasses below.

The first resource index subclass, DefaultResourceIndex, stands for a straightforward solution, keeping a map (hash table) to link resource URIs to their corresponding ResourceInfo instances. However, this indexing solution presents a significant memory overhead, since all (dictionary-encoded) resource URIs, together with their ResourceInfo instances, need to be kept in memory. In order to reduce memory usage, we also provide a MultiLvlResourceIndex subclass. This resource index leverages a multi-level index solution, implemented by the MultiLvlIndex class. In particular, this multi-level index supports swapping with persistent storage to save memory, whereby a configurable amount of indices can be persistently stored. Since MultiLvlIndex is a subclass of MultiLvlEntry, indices can be nested. By further letting the MultiLvlEntry class inherit from the CacheUnit class, we can re-use the existing CacheManager class to manage the in-memory space occupied by the indices (see section 4.7.2.3). In our case, the multi-level index solution is configured to span two levels. The first-level MultiLvlIndex object links the resource URI domains to second-level MultiLvlIndex instances. In turn, these second-level indices connect the local parts of resource URIs to their corresponding ResourceInfo instances. The second-level MultiLvlIndex objects are registered with the CacheManager, meaning these indices will be moved to persistent storage in case memory is full.

We configured the CacheManager instance with the Least-Frequently-Used (LFU) removal strategy, which proved to perform well for our multi-level index solution. Instead of a concrete in-memory byte limit, we limit the number of (second-level) indices to be kept in-memory. We note that, although the MultiLvlResourceIndex introduces a significant performance overhead due to the persistent swapping of data, it allows us to keep the utilized memory in check. We discuss the performance of both resource indices in our evaluation chapter (see Chapter 5, section 5.5.2.2).

4.7.3.2 Source Index Model

The SIMResourceInfo subclass encapsulates the resource information required by the Source Index Model's type mediation process. Aside from the resource URI and previously found types, SIMResourceInfo also keeps a list of sources in which the resource was found. Below, we discuss the SIM type mediation process during the Source Encounter and Data Query phases.

Source Encounter phase

RDF resources from newly encountered online sources, together with their found types, are checked against the information in the resource index. The goal of this check is to determine whether the resources were previously encountered, and if so, whether the source defines new types for these resources. If new types are found, the type mediation process ensues, which updates the multi-level index kept by the SIM (see section 4.4.1). More specifically, new metadata combinations are added, reflecting the newly found types for the resource. For instance, consider a new `dcmi:Agent` type for a particular resource, which was previously found as subject with predicate `foaf:knows` and object type `bio:Agent`. In that case, the metadata combination *foaf:knows – dcmi:Agent – bio-Agent* needs to be added to the SIM.

To construct these new metadata combinations, we need to know to which predicate (e.g., `foaf:knows`) and subject/object type(s) (e.g., `bio:Agent`) the particular resource was previously linked. In order to obtain this information, the type mediation component retrieves all data sources in which the resource occurs, as indicated by the associated SIMResourceInfo instance. These sources are obtained either from the cache (e.g., Source Cache; see section 4.5.1.1) or by re-downloading them. From these sources, the type mediator obtains the predicates and subject/object resources occurring together with the particular resource, and retrieves those resources' types via the resource index. It can be observed that, in case no local cache is present, this mediation process will incur a significant overhead caused by the required source downloads. As a result, this is an optimistic solution, which assumes that type mediation does not need to occur often.

Data Query phase

During the Data Query phase, the SIM identifies query-relevant sources based on its up-to-date metadata. However, while the SIM metadata was updated by the type mediation process, the online sources cannot be updated with the mediated resource types. This is a result from our specific setting, where data is captured in online files not under our control (see second challenge, *Data captured in online files*). To efficiently deal with this issue, the ResourceIndex additionally stores a mapping between source URLs and their contained resource URIs. During the Data Query phase, the mappings for the identified sources are loaded, whereby the mediated types of contained resources are added to the final query graph.

4.7.3.3 Meta Cache

The CacheResourceInfo class wraps the resource information required by the type mediation process for Meta Cache. This data includes the resource URI, together with references to cache units containing the resource. Below, we shortly detail the type mediation process for Meta Cache during the two different phases.

Source Encounter phase

RDF resources extracted from newly encountered sources, together with their associated types, are checked against the resource index. In case corresponding CacheResourceInfo objects are found, the previous types are obtained via one of the associated cache units' metadata combinations. In case the new source defines new resource types, the type mediation process ensues. Essentially, this involves moving the resource's cached triples to other cache units, matching the resource's new types. Afterwards, the CacheResourceInfo objects, corresponding to the particular triple resources, are updated with the new corresponding cache units.

During experiments, we found that this process, which involves extracting triples from their cache units, lead to significant performance issues (as it required loading persistently stored cache units entirely into memory). To optimize the extraction process, we utilize a persistent map (in particular, a B+ tree[84]) connecting each stored resource URI to pairs of byte offsets, pointing to the resource's triples in the stored data. This index enables us to load only those triples specifying the particular resource, significantly reducing the read overhead. In addition, a list of byte offset pairs is kept, pointing to the triples that were previously extracted. When the cache unit is afterwards loaded into memory, the indicated triples are simply skipped. Finally, constructing and maintaining this extraction index naturally incurs a performance overhead. Therefore, this index is only kept for cache units exceeding a certain size, for which the regular extraction process would lead to problematic performance.

Data Query phase

The up-to-date metadata associated with retrieved cache units, represented by MetaCacheKey objects, is directly utilized as (mediated) type information for the cached resources (see section 4.7.2.6). On the other hand, type mediation needs to be re-applied to sources (see process above) that were downloaded in response to cache misses. As before, this is a result from our setting where data is captured in online files not under our control (see second challenge, *Data captured in online files*; section 4.2).

4.8 Summary

In this chapter, we introduced a mobile, general-purpose and client-side query service, supplying transparent and integrated query access to large amounts of small online sources. In doing so, the mobile query service unlocks a large Semantic Web segment for consumption by mobile devices, consisting of online RDF files and semantically annotated websites (e.g., via RDFa).

In order to meet the challenges arising in our particular querying scenario, two solutions naturally present themselves. By *identifying online data relevant to posed queries*, we meet our two challenges of reducing the final query dataset and decreasing the number of source downloads. Moreover, by *locally caching data* likely to be frequently referenced, the number of required source downloads can further be reduced. Furthermore, two key requirements should be considered: *fine-grained data identification and retrieval*, whereby online sources are identified with high selectivity and locally cached data is retrieved in a fine-grained way; and *reducing memory usage and processing effort*, to cope with the fact that mobile device capabilities are still relatively limited compared to larger devices (e.g., laptops).

The aforementioned solutions are implemented via the Source Index Model (SIM), which identifies query-relevant online sources by indexing source metadata; and the Source and Meta Cache

[84] http://bplusdotnet.sourceforge.net/ (access date: 21/05/2013)

components, which locally cache source data. Respectively, these caches organize the cached data via origin source and shared source metadata. In order to efficiently identify query-relevant online sources, the SIM utilizes a multi-level index structure. By investigating these two separate cache components, as well as SIM variants keeping varying amounts of source metadata, we aim to study the impact of source metadata on data selectivity and performance (see requirements above). On the other hand, we note that our focus on source metadata incurs a drawback related to the Semantic Web's Open World Assumption (OWA). In case online sources specify extra types for already processed source data, previously indexed metadata may become out-of-date. To resolve this issue, a type mediation process should be applied (see below). Finally, to keep the cached data up-to-date, we apply a data validity strategy.

We introduced a removal strategy called Least-Popular-Sources (LPS) for Meta Cache, suited towards our particular setting where data originates from online files. In order to optimize query resolution, LPS considers the "popularity" of online sources, as signified by 1/ the popularity of its source data, i.e., the number of stored cache units containing the data, and 2/ the popularity of its own metadata, i.e., how often the contained metadata is found in other sources. These factors reduce the probability of cache misses and keep the number of source re-downloads in check. In order to make it less likely that sources with a high download cost are removed, source download times are considered as well.

Finally, our mobile query service implements two features to support the Semantic Web's Open World Assumption (OWA). The first feature, called type mediation, is applied in case online sources specify different types for the same RDF resources. This process involves keeping track of resource types across sources, and updates components to reflect new resource types. This feature can be enabled or disabled, based on the composition of the online semantic dataset. Furthermore, we note that related approaches also encounter this issue but do not attempt to resolve it [2, 7, 8]. Type inferencing enriches the kept source metadata, as well as extracted search constraints, using ontological knowledge (i.e., domain/range restrictions, subclass relations). To suit mobile application needs and device capabilities, type inferencing can likewise be enabled or disabled (cfr. RDF store inferencing support).

Chapter 5

Experimental validation

The previous chapter elaborated on the mobile query service. We discussed issues and challenges arising in our specific querying scenario, and presented two solutions to tackle these challenges; namely, *identifying online query-relevant data* and *locally caching data*. These solutions were respectively implemented by 1/ the Source Index Model, and 2/ the Source Cache and Meta Cache. We further specified a number of requirements to which these software components should adhere. We discussed our aim to investigate whether source metadata (i.e., predicates and types) enables us to achieve a balance between fine-grained data retrieval on the one hand, and memory and processing overhead on the other. Regarding the cache, we introduced the Least-Popular-Sources removal strategy, which is fine-tuned to our specific setting where data originates from online sources. The chapter also discussed support for the Semantic Web's Open World Assumption (OWA).

This chapter presents an experimental validation of the query service components. In our experiments, we apply a mobile context-aware scenario, where SCOUT plays the role of client. SCOUT utilizes the mobile query service to transparently query online semantic data, associated with the user's surroundings. A real-world dataset, extracted from existing online sources, serves as the online semantic dataset.

The evaluation encompasses the following three sub-evaluations:

- **Source Index Model**

In order to accurately evaluate the impact of source metadata on data selectivity and performance, we study and compare different variants of the SIM. Each variant keeps varying amounts of metadata: *SIM1*, which only indexes predicates; *SIM2*, indexing predicates and subject types; and *SIM3*, which indexes predicates, subject and object types. In addition, we check the case where no SIM is used (i.e., native query engine performance) and queries are simply executed on the entire dataset. We offset the runtime overhead and memory requirements to the resulting source selectivity and query execution times.

- **Cache**

The Source Cache and Meta Cache organizations are compared, which respectively organize cached data via origin source and shared metadata. Similar as for the SIM, we contrast runtime performance and memory overhead to the fine-graininess of data retrieval and query execution performance. Furthermore, we investigate the effect of applying the Least-Popular-Sources (LPS) strategy; and compare it to when an arbitrary removal strategy is employed, in this case the Least-Recently-Used (LRU) strategy. For LPS, we test different weights for the factors in the removal value calculation, and evaluate their impact. Furthermore, we study how different persistent data clustering methods influence performance.

- **Open World Assumption**

We evaluate the two Semantic Web Open World Assumption (OWA) features, namely type inferencing and type mediation. The performance and memory overhead incurred by these features is weighed against the resulting improvement in data access, yielded by retrieving additional query-relevant results and ruling out irrelevant data.

This chapter is structured as follows. Section 5.1 outlines the setup for the experiments. In the following sections, the individual experiment results for the Source Index Model (section 5.2), cache (section 5.3), removal strategies (section 5.4) and OWA features (section 5.5) are presented and discussed in detail. Section 5.6 presents a general conclusion based on the experimental results. In section 5.7, we summarize this chapter.

5.1 Experiment setup

This section elaborates on the experiment setup. We detail the mobile device used for the experiments (section 5.1.1), as well as the experiment dataset (section 5.1.2) and querying scenario (section 5.1.3). The section also discusses the different aspects of the experiment methodology (section 5.1.4).

5.1.1 Device

The experiments were performed on a Samsung Galaxy S III, with a 1.4GHz quad-core processor, 1GB RAM and 16GB persistent storage. The installed Android OS was version 4.1.2 (Jelly Bean) with API level 16. On this device, Android applications obtain a maximum Java heap space of 64Mb.

5.1.2 Dataset

The online semantic dataset consists of 5000 data sources, with a total size of 526Mb (average size is ca. 108Kb). These data sources were extracted from various online datasets, some of which were found on the Billion Triples Challenge (BTC) 2012 Dataset webpage[85]. For our experiments, the data sources (in N-Triple format) were distributed across four different web servers (not located on the local network). Below, we shortly discuss the datasets from which the sources were extracted.

1) *Freebase*[86] is a community-curated online information system for people, places and things, and supplies an RDF version of its data. This RDF data was retrieved by the BTC team and provided on their webpage.

2) The *Timbl* dataset was crawled by the BTC team starting from Tim Berners-Lee's FOAF file[87], and provided on the BTC webpage.

3) *LinkedGeoData*[88] (LGD) is an online semantic dataset, based on data collected from the OpenStreetMap[89] project. We retrieved this dataset via the data dump files provided on their website.

4) *DataHub*[90] is a community-run catalogue of useful data on the Web. The BTC team retrieved the RDF data from DataHub and provided it on their webpage.

5) The *BestBuy*[91] website is a large online webshop providing RDF versions of its data. We performed a crawling process to collect a subset of this RDF data.

6) *DBPedia*[92] is an online semantic dataset containing structured information from Wikipedia. We utilized data from DBPedia to enrich the extracted Freebase and LinkedGeoData datasets. This was made possible via freely available interlinks, which connect equivalent resources from these datasets via the owl:sameAs predicate. Due to these interlinks, datasets such as DBPedia, LinkedGeoData, NYTimes, and Geonames are not closed data silos, but instead become interlinked parts of the Linked Data cloud[93].

7) The *NYTimes*[94] maintains an RDF version of its online news information. As above, we leveraged the available interlinks to enrich the Freebase dataset with information from this online dataset.

8) *Geonames*[95] is an online geographical database which provides its data in RDF. We exploited interlinks to enrich the LinkedGeoData dataset with information from the Geonames dataset.

[85] http://km.aifb.kit.edu/projects/btc-2012/ (access date: 28/04/2013)
[86] http://www.freebase.com (access date: 28/04/2013)
[87] http://www.w3.org/People/Berners-Lee/card.rdf (access date: 28/04/2013)
[88] http://linkedgeodata.org (access date: 28/04/2013)
[89] http://www.openstreetmap.org/ (access date: 30/07/2013)
[90] http://datahub.io/ (access date: 28/04/2013)
[91] http://www.bestbuy.com (access date: 28/04/2013)
[92] http://dbpedia.org (access date: 28/04/2013)
[93] http://lod-cloud.net/ (access date: 19/07/2013)
[94] http://data.nytimes.com/ (access date: 28/04/2013)
[95] http://www.geonames.org/ontology/ (access date: 28/04/2013)

Note that these datasets comprise general information on people (Timbl dataset), places and things (Freebase, DBPedia, and NYTimes datasets), location-related and geographical information (LinkedGeodata and Geonames datasets), as well as products for sale (BestBuy dataset) and randomly collected data (DataHub). Therefore, they suit the context-aware query scenario applied in our experiments (see section 5.1.3). The dataset is available on http://wise.vub.ac.be/william/phd/index.htm#experiments.

5.1.3 Query scenario

Our experimental validation applies a context-aware query scenario, where SCOUT poses as a client. In this scenario, SCOUT continuously discovers new physical entities, including people, places and things, in the user's vicinity (e.g., using sensing technologies such as RFID/NFC), and extracts references to associated online semantic sources (e.g., by reading URLs from RFID tags). These detected source references are passed to the mobile query service during the Source Encounter phase (see Chapter 4, section 4.3).

During the Data Query Phase, SCOUT utilizes the query service to achieve transparent and integrated query access to the detected semantic sources. We constructed five queries that request useful information in a context-aware setting, and refer to the different types of data in the experiment dataset (e.g., geographical entities, people). One query (the first query) returns all shopping centers together with their names, absolute coordinates, town, and (optionally) photos; and another query (the third query) returns all airports together with their absolute coordinates. Such queries, which return geographical data, allow plotting relevant physical entities (i.e., shopping centers and airports) on a map. Another query (the second query) selects all persons and the groups they are member of, optionally with images depicting these persons and their online chat accounts. The fourth query retrieves all exhibitions, together with their names, start- and end-dates, venues, displayed art pieces and the names of those pieces. The fifth and final query selects products for sale below 20 dollars, their price, manufacturer, product name, and user comments. Such queries retrieve an overview of interesting physical entities in the user's vicinity, such as people, art exhibits, and products for sale, together with useful details and an indication of their relevance. For instance, the second query indicates the interests of people (implied by their groups) and returns informal contact information (chat accounts); the fourth query returns the displayed art pieces, as well as the time period and venue in which each exhibit takes place; and the final query returns products in an affordable price range, accompanied by useful consumer data (e.g., manufacturer, user comments). The queries can be found in Appendix B, as well as on http://wise.vub.ac.be/william/phd/index.htm#experiments.

5.1.4 Methodology

Each individual experiment evaluates a particular aspect of the approach; i.e., a query service component, or the impact of removal strategies or OWA features on a component. All experiments include the Source Encounter phase, where sources from the experiment's dataset are retrieved and processed, and Data Query phase, where the experiment's queries (see section 5.1.3) are executed on the encountered dataset. During the Source Encounter phase, the query service encounters all 5000 dataset sources, unless stated otherwise (e.g., fewer sources are encountered in case of memory issues). It should further be noted that, when evaluating the effect of the OWA features on the query service components, we only show experiment results for the component variant that performed best in general. In those cases, we clearly indicate which component variant is used in the experiment.

Below, we discuss other points pertaining to the experiment methodology, including measurement methods, component configurations, and data retrieval.

5.1.4.1 Measurement methods

In this section, we discuss how performance times and memory usage were measured during the experiments.

Performance times: In order to minimize the effect of external factors influencing execution times (i.e., external / OS processes such as garbage collection), the experiments were run five times and the average times are shown. Note that experiments involving the OWA features took an exceedingly long time to execute, as can be seen from the experiment results (see section 5.5). Because of these very high execution times, and the fact that these experiment results only serve to indicate performance overhead (i.e., they are not used to determine better-performing components), the OWA experiments were only performed once.

Memory usage measurements: In order to accurately measure memory usage, snapshots were taken of the Android Java heap at runtime (using the Android API), and analyzed using the Eclipse Memory Analysis tool[96]. We show the retained heap size of the relevant classes, as well as the dictionary-encoder component (see Chapter 4, section 4.7.1.1). Note that for the Source Encounter phase, we typically show multiple memory usage results, whereby each result was measured after a certain number of sources were encountered during the phase.

JRE experiment results: It should be noted that, aside from the Android implementation, the mobile query service was also implemented for Java Runtime Environment (JRE) v. 6 (Java Development Kit (JDK) v. 1.6.0). Both query service versions are based on a shared Java code base, whereby each version adds platform-specific classes and libraries (e.g., RDF, Bluetooth libraries). Initially, the JRE mobile query service was implemented for easier debugging during development. In our experiments, the JRE version was used in case an experiment failed on the Android device (due to limited memory) but still yielded interesting results, for instance regarding data selectivity or memory usage. We clearly indicate when this is the case.

[96] http://www.eclipse.org/mat/ (access date: 28/04/2013)

5.1.4.2 Component configuration

In this section, we detail the configuration of the query service components for our experiments.

Cache memory and storage usage: The cache components are configured to use up to 75% of the total dataset size as persistent storage, and 8Mb volatile memory space for storing cached data[97] (called payload). We elected to specify an upper limit for the persistent storage, since a user likely does not want the cache to consume the entire mobile device storage. The relatively low in-memory limit was chosen since other parts and components also consume memory (e.g., Androjena RDF graphs), and the limit only covers the cached payload.

Resource indices: When applying the OWA type mediation feature, information on RDF resources such as types and origin sources needs to be indexed (see Chapter 4, section 4.7.3.1). In our experiments, we evaluate a default resource index that utilizes a single hash table for indexing resources, called default resource index; and a multi-level resource index, which swaps index data to persistent storage to save memory. The multi-level index is configured to contain two levels, keeping *n* second-level indices in-memory and utilizing a persistent SQLite database. The *n* value indicates a balance between memory and performance overhead; less indices kept in-memory (i.e., lower n value) result in less memory usage, but incur higher persistent read and write times. We chose the following *n* values, based on test results; for the SIM, n = 10 (i.e., 10 second-level indices are kept in-memory), and for Meta Cache, n = 100 (in case n = 10 is set for Meta Cache, runtime overhead becomes impractically high). We chose to keep two levels in the multi-level index, corresponding to the structure of URI keys (i.e., domain / local name); the first level indexes on URI domain, and the second level on local name.

5.1.4.3 Data retrieval

This section elaborates on how data (e.g., RDF files, ontologies) is retrieved in our experiments.

RDF source retrieval: In order to avoid temporary disconnections or network delays to influence experiment results, RDF sources were not downloaded from online locations but instead retrieved from persistent storage. Afterwards, the persistent retrieval times were substituted by average download times. The latter were obtained by downloading 1000 random sources from their online location, over 5 runs, and calculating the average times. Clearly, temporary network disconnections and delays will occur in real-world environments. On the other hand, the goal of these experiments is to evaluate the performance and memory usage of mobile query service components when handling large amounts of data, as well as the influence of component features on data access. Evaluating the impact of network properties, as well as other mobility-related issues such as battery consumption, is not included in the presented experiments and considered future work.

Ontology Manager: In the same vein, all referenced ontologies were stored locally for later use by the Ontology Manager (see Chapter 4, section 4.6.1). Although the Ontology Manager supports

[97] This does not include extra data such as indices.

dynamically downloading ontologies, this automatic download mechanism has its drawbacks: many ontologies do not adhere to the linked data principles, and cannot be obtained by simply de-referencing their URIs; while some ontologies also take a long time to download, due to their particular hosting server. In order to avoid these issues influencing our experiments, ontologies are retrieved from persistent storage when referenced. We found online ontologies for a total of 191 namespaces referenced in the RDF sources and queries.

5.2 Experiment 1: Source Index Model

This section presents the experiment evaluating the Source Index Model. In this experiment, we consider three different SIM variants, each keeping an increasing amount of metadata: SIM1 (predicates), SIM2 (predicates and subject types), and SIM3 (predicates, subject, and object types).

In section 5.2.1, we present the experiment results. This includes the memory and computational overhead of each SIM variant during the Source Encounter phase, and the data access efficiency supplied during the Data Query phase. In section 5.2.2, we study these results in more detail.

5.2.1 Experiment 1: Results

This section shows the results of the Source Index Model experiment during the Source Encounter (section 5.2.1.1) and Data Query (section 5.2.1.2) phases.

5.2.1.1 Source Encounter phase

In this section, we show the experiment results of the Source Index Model variants during the Source Encounter phase. Table 5-1/a shows the memory sizes taken up by the SIM variants, measured each time an additional 1000 sources were encountered. Specifically, we show the collective retained heap size of the implementing SIM class, together with the dictionary encoder (see section 5.1.4.1). The computational overhead to maintain the SIM is shown in Table 5-1/b, and includes average times of downloading a source, extracting the source metadata and adding the metadata to the SIM.

#sources (size)	**SIM1**	**SIM2**	**SIM3**
1000 (81Mb)	360	2566	4137
2000 (200Mb)	597	3859	6142
3000 (336Mb)	846	5220	8163
4000 (442Mb)	1063	5663	8725
5000 (526Mb)	1235	5882	8986

(a) Memory size (Kb)

	SIM1	**SIM2**	**SIM3**
extract & add	178	286	482
download	301		
total	479	587	783

(b) Avg. processing times / source (ms)

Table 5-1. *SIM – Source Encounter phase*.

5.2.1.2 Data Query phase

This section shows the Source Index Model experiment results during the Data Query phase. Table 5-2 illustrates the source selectivity by showing the number of identified query-relevant sources for each of the five experiment queries (see section 5.1.3). The table also shows the number of results per query, which are the same for each SIM variant. Tables 5-3 to 5-5 show the total query execution times, which include query analysis time (analyze), source identification time (id), data collection (collect) and query execution times (execute). SIM1 fails with an out-of-memory error for the final two queries. Therefore, Table 5-3 does not show entries for those queries; the related selectivity results (see Table 5-2) were obtained via the JRE version of the query service. The case where no SIM is employed (i.e., native query engine performance) fails with this error for any query, so again no data is available.

query	**SIM1**	**SIM2**	**SIM3**	**# query results**
Q1	2116	254	254	4
Q2	313	305	272	272
Q3	1293	319	319	319
Q4	1984	87	87	77
Q5	2146	256	256	148

Table 5-2. *SIM – Data Query phase*: Source selectivity (#sources).

SIM1					
query	*analyze*	*id*	*collect*	*execute*	*total*
Q1	34	2344	1055884	22253	1080515
Q2	11	23	156187	583	156804
Q3	10	1042	645207	1490	647749
Q4	53	2333	-	-	-
Q5	87	3228	-	-	-

Table 5-3. *SIM – Data query phase (SIM1)*: Query execution times (ms).

SIM2					
query	*analyze*	*id*	*collect*	*execute*	*total*
Q1	67	183	126746	1354	128350
Q2	8	24	152195	472	152699
Q3	7	73	159181	655	159916
Q4	8	13	43413	207	43641
Q5	13	192	127744	35878	163827

Table 5-4. *SIM – Data query phase (SIM2)*: Query execution times (ms).

SIM3					
query	*analyze*	*id*	*collect*	*execute*	*total*
Q1	68	230	126746	1464	128508
Q2	10	19	135728	398	136155
Q3	8	107	159181	761	160057
Q4	10	19	43413	173	43615
Q5	18	293	127744	39731	167786

Table 5-5. *SIM – Data query phase (SIM3)*: Query execution times (ms).

5.2.2 Experiment 1: Discussion

In the sections below, we investigate and discuss the SIM experiment results during the Source Encounter (section 5.2.2.1) and Data Query phase (section 5.2.2.2).

5.2.2.1 Source Encounter phase

The Source Index Model is responsible for identifying query-relevant online information, in order to minimize source downloads and reduce the final query dataset. Note that in the experiment, the SIM was used individually, without a cache component to locally cache source data. In the next section, we discuss experiment results when the SIM is used in combination with a local cache.

As expected, Table 5-1/a shows that the memory space taken up by the SIM increases as more metadata is kept; whereby SIM3 yields the largest memory overhead. We observe that the memory size of any SIM variant corresponds to only a small fraction of the total dataset; the largest variant, SIM3, stores around 1,7% for 5000 sources. For instance, the largest SIM (ca. 8,8Mb) fits easily into the smallest Android heap size (16Mb)[98]. This adheres to our requirement of reducing memory usage

[98] The available heap size depends on the specific device and Android version; to our knowledge, the smallest available heap size is currently 16Mb.

(see Chapter 4, section 4.2). On the other hand, we observe the SIM could exceed the memory limit in case larger amounts of sources are encountered. In fact, when applying the type inferencing feature (see section 5.5) and enabling all its options, the resulting SIM size already exceeds the maximum heap space for 5000 sources (see Table 5-22/a; *+subclass* column). In order to cope with this, parts of the SIM can be swapped to persistent storage, incurring a large performance overhead (as illustrated by our persistent-swapping, multi-level resource index; see section 5.5).

The computational overhead of extracting and adding data to the SIM (see Table 5-1/b) is reasonable compared to the average download time. This complies with our requirement for minimal computational overhead. In line with expectations, this overhead increases as more metadata needs to be extracted; whereby SIM3 shows the largest processing overhead. It should be noted that we initially executed predefined SPARQL queries to extract the required source metadata (predicates, subject/object types). This was feasible in previous experiments [86, 93], where the experiment dataset was partially synthetic. However, this extraction method proved problematic for our real-world dataset, where relatively large files (e.g., 3,5Mb) resulted in huge extraction times (e.g., 31s). On average, files in our real-world dataset contain much more distinct metadata than in our previous datasets, leading to much larger extraction times. To resolve this issue, we implemented an iterator class that dynamically parses RDF data (N-Triple format) as new statements are requested (see Chapter 4, section 4.7.1.2). Via this iterator, we reduced the metadata extraction time on average by a factor 10.

In conclusion, we observe that while the computational and storage overhead rises with the complexity of the SIM, the overhead is still acceptable. As such, the SIM component complies with our requirements of reducing memory usage and processing effort.

5.2.2.2 Data Query phase

Conforming to expectations, the source selectivity (see Table 5-2) increases with the amount of kept metadata. The selectivity of SIM1 is so poor that an out-of-memory error occurs when assembling the sources for the last two queries (see Table 5-3). A scenario where the final query dataset includes all sources, equivalent to native query engine performance (i.e., without SIM support), proved to be entirely unfeasible as it led to out-of-memory exceptions for all queries.

It can be observed that SIM2 and SIM3 rule out a large number of sources (95% on average). Focusing on the individual selectivity results, we only observe a difference in selectivity between SIM2 and SIM3 for the second query (Q2; see Appendix B.1.1). This query restricts the object types of each triple pattern, meaning SIM3 can utilize the richer search constraints to increase selectivity. On the other hand, Tables 5-4 and 5-5 show that SIM2 and SIM3 both still incur a very high query execution overhead. As indicated by the results, most of this overhead occurs during the collection step, which involves downloading the sources and integrating all source data into a single queryable Androjena RDF graph (which also requires parsing the data). It can be noted that such overheads are unavoidable for an indexing solution deployed without caching support, since all query-relevant sources need to be downloaded as a whole and integrated. In fact, the Androjena library was already extended for efficiently combining RDF graphs (see Chapter 4, section 4.7.1). By additionally employing a cache for locally storing downloaded data, the collection overhead can be significantly reduced, as we illustrate in section 5.3. Finally, regarding the query execution times (execute column; this involves executing queries on the already collected data), our experiments show that these times are mostly bound by query complexity, as the SIM2 and SIM3 execution times are similar to the query execution times incurred by Meta Cache (we shortly revisit this issue in section 5.3.2.2).

We conclude that executing queries on online semantic datasets is made feasible via the SIM, as the native query engine, without SIM support, fails for any query due to out-of-memory exceptions. The SIM variants utilizing increased amounts of source metadata, namely SIM2 and SIM3, represent the best solutions. As such, these SIM variants confirm that source metadata indeed enables a balance between fine-grained data retrieval on the one hand, and memory and processing requirements on the other. At the same time however, we observe that in case the SIM is deployed individually, without any caching support, the total execution time is impractically high; ranging from around 1-3 minutes for SIM2 and SIM3. In order to improve performance and reduce the data collection overhead, the SIM should be combined with a caching system.

5.3 Experiment 2: Cache

In this section, we present the experiment evaluating the cache components. We evaluate and compare two cache organizations: Source Cache, which organizes cached data based on origin source; and Meta Cache, which groups cached data according to their shared metadata. The Meta Cache performs both the online source identification and local caching tasks, and is thus deployed autonomously. For Source Cache, the best performing SIM variant (SIM3) is utilized to identify query-relevant online sources. Furthermore, the general-purpose LRU removal strategy was applied to both caches. Section 5.4 shows the effects of applying different removal strategies.

Section 5.3.1 shows the experiment results, summarizing the memory and processing overhead of the cache organizations during the Source Encounter phase, as well as the query execution performance during the Data Query phase. Section 5.3.2 investigates the results in more detail.

5.3.1 Experiment 2: Results

This section presents the results of the cache experiment during the Source Encounter (section 5.3.1.1) and Data Query (section 5.3.1.2) phases.

5.3.1.1 Source Encounter phase

This section shows the experiment results for the two caches during the Source Encounter phase. In Tables 5-6 and 5-7, we show the in-memory and persistent storage spaces respectively used by the Source and Meta Cache, measured each time an additional 1000 sources were encountered. The total memory space includes the retained heap sizes of the cache store and cache unit objects, together with the dictionary encoder. We separately indicate the measured payload size, which comprises the heap size of the cache units (not including e.g., indices), and offset this value to the estimated payload size (shown between brackets). The measured payload size was obtained via snapshots of the Android Java heap (using the Android API; see section 5.1.4.1) and accurately reflects the memory usage. On the other hand, the estimated payload size is approximated at runtime to dynamically manage memory space (see Chapter 4, section 4.7.2.3). For ease of reference, Table 5-6 includes the corresponding SIM sizes, as the Source Cache is used together with the Source Index Model. The second part of the table shows the composition of the cache, which includes the total number of cache units, together with the number of in-memory, persistent and removed units.

Source Cache								
#sources (size)	*sizes (Kb)*				*composition (# units)*			
	in-memory			*persistent*	*total*	*in-memory*	*persistent*	*removed*
	SIM	*cache*	*payload*					
1000 (81Mb)	4137	6249	6027 (8192)	56849	1000	44	956	0
2000 (200Mb)	6142	6580	6130 (8192)	153515	2000	45	1955	0
3000 (336Mb)	8163	6847	6226 (8192)	286911	3000	45	2955	0
4000 (442Mb)	8725	7223	6322 (8192)	395542	4000	45	3955	0
5000 (526Mb)	8986	7291	6323 (8192)	395776	5000	45	3968	987

Table 5-6. *Source Cache – Source Encounter phase*: Sizes and compositions.

Meta Cache							
#sources (size)	*sizes (Kb)*			*composition (# units)*			
	in-memory		*persistent*	*total*	*in-memory*	*persistent*	*removed*
	total	*payload*					
1000 (81Mb)	15780	12092 (6660)	61454	10531	805	9726	0
2000 (200Mb)	21783	15228 (8192)	162468	15552	1902	13650	0
3000 (336Mb)	28270	17846 (8191)	278579	22439	871	21568	0
4000 (442Mb)	31870	18540 (8191)	369703	23590	836	22754	0
5000 (526Mb)	37125	17708 (8079)	394351	24068	926	11952	11190

Table 5-7. *Meta Cache – Source Encounter phase*: Sizes and compositions.

In Tables 5-8 and 5-9, we show the average computational overhead during the Source Encounter phase, resulting from adding the data to the cache (add column) and performing the replacement (or removal) function in case the cache is full. The Source Cache incurs an extra overhead of updating the SIM, which is used to identify query-relevant cached sources during the Data Query phase. Additionally, since the Source Cache stores Androjena RDF graphs as payload (see Chapter 4, section 4.7.2.5), the add time for Source Cache also includes loading the source data into an Androjena RDF graph (creation time shown between brackets). Meta Cache incurs two extra overheads; namely, extracting metadata from source triples, and updating existing cache units.

Source Cache				
insert			*replacement*	*total*
SIM update	*add*	*total*		
517	253 (198)	770	2212	2982

Table 5-8. *Source Cache – Source Encounter phase:* Avg. processing times / source (ms).

Meta Cache					
insert				*replacement*	*total*
extract	*add*	*update*	*total*		
830	140	536	1506	804	2310

Table 5-9. *Meta Cache – Source Encounter phase*: Avg. processing times / source (ms).

5.3.1.2 Data Query phase

In this section, we show the experiment results for the Source Cache and Meta Cache during the Data Query phase. Tables 5-10 and 5-11 show the total query execution times. The following parts are distinguished: 1) query analysis, which includes extracting search constraints; 2) cache access, which comprises retrieving cached data and downloading missing sources; 3) data assembly, which involves collecting retrieved triples into a final RDF graph for querying; and 4) query execution, where the query is executed on the assembled RDF graph. In more detail, the cache access time is divided into 1/ cache retrieval time, which includes accessing indices and loading persistently stored data; and 2/ cache miss time, which are incurred on a cache miss whereby missing source data needs to be re-downloaded. Also, the data assembly part shows the total number of returned triples (including triples resulting from cache misses), illustrating the data retrieval fine-graininess. For Meta Cache, the amount of extra type statements, which are required to make the types of cached resources explicit in the query graph, is shown between brackets[99] (see Chapter 4, section 4.7.2.6). Finally, the total execution time, including all the constituent times, is also shown.

Source Cache							
query	*query analysis*	*cache access*		*assemble data*		*query execution*	*total*
		retrieval	miss	*# triples*	*time*		
Q1	314	21661	0	25364	2139	1739	25853
Q2	12	1346	76153	31706	1575	585	79671
Q3	13	22888	0	24396	1517	823	25241
Q4	19	12274	0	13295	1113	243	13649
Q5	184	14937	0	13392	736	50713	66570

Table 5-10. *Source Cache – Data Query phase*: Query execution (ms).

[99] It should be noted that these type statements may include duplicates (in case the resource occurs in multiple cache units).

Meta Cache							
	query	*cache access*		*assemble data*		*query*	
query	*analysis*	retrieval	miss	*# triples*	*time*	*execution*	*total*
Q1	198	8023	28373	1592 (19597)	3446	1403	41443
Q2	14	588	3933	867 (1759)	542	749	5826
Q3	92	2220	27653	1804 (25745)	4365	757	35087
Q4	11	598	0	545 (648)	181	291	1081
Q5	19	22	0	2914 (4292)	2562	55301	57904

Table 5-11. *Meta Cache – Data Query phase:* Query execution (ms).

Tables 5-12 and 5-13 show additional information regarding cache access. Regarding cache retrieval, we indicate the number of retrieved in-memory / persistent cache units; and concerning cache misses, we show the total number of misses, accompanied by the resulting amount of sources to re-download between brackets. For ease of reference, we repeat the total cache retrieval and cache miss times. It should be noted that for Source Cache, the cache-miss time includes loading the source data into an Androjena RDF graph; and for Meta Cache, this time includes extracting the required source metadata and associated triples. For Source Cache, we also show the SIM access time.

Source Cache							
	SIM	*retrieval*			*misses*		
query	*access*	*# in-memory*	*# persistent*	*time*	*# misses*	*time*	*total*
Q1	216	8	246	21445	0	0	21661
Q2	20	0	19	1326	253 (253)	76153	77499
Q3	107	0	319	22781	0	0	22888
Q4	21	0	87	12253	0	0	12274
Q5	278	4	252	14659	0	0	14937

Table 5-12. *Source Cache – Data Query phase*: Cache access (ms).

Meta Cache						
	retrieval			*misses*		
query	*# in-memory*	*# persistent*	*time*	*# misses*	*time*	*total*
Q1	1	562	8023	223 (66)	28373	36396
Q2	0	5	588	2 (7)	3933	4521
Q3	66	125	2220	25 (74)	27653	29873
Q4	0	6	598	0	0	598
Q5	8	1	22	0	0	22

Table 5-13. *Meta Cache – Data Query phase*: Cache access (ms).

Table 5-14 shows the cache maintenance times resulting from cache access. This comprises 1/ updating the cache with new source data, in case missing data was downloaded; and 2/ running the removal strategy, resulting from loading cached data into memory and adding new data to the cache (i.e., in case of cache misses). Since cache maintenance occurs after query execution, it is not included in the cache access times shown in the previous tables. It should be noted that these times heavily depend on the employed removal strategy; see the next section (section 5.4) for a comparison of removal strategies.

	Source Cache		**Meta Cache**	
query	*add / update*	*replacement*	*add / update*	*replacement*
Q1	0	133183	36571	4790
Q2	2157	195673	6	1
Q3	0	82450	9635	5003
Q4	0	105947	0	2
Q5	0	103051	0	31

Table 5-14. *Cache – Data Query phase*: Cache maintenance (ms).

5.3.2 Experiment 2: Discussion

In the sections below, we study and discuss the cache experiment results during the Source Encounter (section 5.3.2.1) and Data Query phase (section 5.3.2.2).

5.3.2.1 Source Encounter phase

In order to reduce data collection time, the cache is responsible for locally storing online source data. Note that the Source Cache is used in combination with the Source Index Model, whereby the latter is responsible for identifying query-relevant online sources. On the other hand, the Meta Cache is used without the SIM, as it is able to perform the identification task by keeping information on previously removed data ("missing" data; see Chapter 4, section 4.7.2.6).

Tables 5-6 and 5-7 summarize the cache sizes and compositions. Firstly, we observe that the payload size estimated at runtime (between brackets) only inaccurately reflects the actual payload memory size (i.e., heap space occupied by cache units). Due to the lack of accurate and efficient runtime memory measurement support on Android, a rough estimation of the payload has to be made, whereby for instance object sizes are not considered (see Chapter 4, section 4.7.2.6). The difference is largest in case of Meta Cache, which is to be expected since more objects are kept (see composition). Because of this, the actual in-memory payload may exceed the specified memory limit. Devising a better way to estimate actual memory usage is future work.

As can be seen from the tables, the total memory usage of Meta Cache is larger than Source Cache, even when including the SIM memory space for the latter cache. Meta Cache includes three indices instead of just one, and the indices themselves are also much larger; where the Source Cache only indexes source URLs, Meta Cache indexes on each metadata part (i.e., predicate, subject/object types). Additionally, Meta Cache keeps much more cache unit objects with their accompanying memory overhead (see composition part). This extra memory consumption, not counting concrete payload, amounts to ca. 0,2% for Source Cache, and 3,6% for Meta Cache, of the total dataset size (5000 sources). The total occupied memory space amounts to ca. 7% of the total dataset size for Meta Cache, and 3% for Source Cache (also counting the SIM). Since the total memory usage by both caches is significantly lower than the maximum heap space (64Mb), this still adheres to the requirement of reduced memory usage. To avoid the cache exceeding the available memory space for large amounts of data, index data can be swapped to persistent storage, which however incurs a large performance penalty (as exemplified by our multi-level resource index; see section 5.5). In addition, a more accurate estimation of memory usage could be devised, making sure the actual in-memory payload does not exceed the defined memory limit.

Regarding composition, the Meta Cache contains a total of 24068 cache units, each of which corresponds to a unique metadata combination. It can further be observed that the number of in-memory cache units decreases when considering composition results after encountering 2000 and 4000 sources, while the defined memory limit remains the same. As increasing amounts of source data are being encountered, it becomes more likely that new source data shares the same metadata as previous source data. Therefore, existing cache units will be updated rather than new ones created, meaning existing cache units increase in size over time. As a result, the total number of cache units will decrease given the same memory limit. For Source Cache, each cache unit corresponds to a source, thus adding up to 5000 cache units (including removed ones).

Tables 5-8 and 5-9 show the processing overhead. Even when including the SIM update cost for Source Cache, the overall cost of inserting cached data, including extraction, add and update times, is higher for Meta Cache. The extraction process takes up the bulk of the insertion time (ca. 55%), which involves extracting source triples together with their metadata. As mentioned before (see section 5.2.2.1), the source metadata extraction process was already optimized by a factor of 10 on average. Additionally, inserting a single source into Meta Cache may require updating multiple units; in case these units are persistently stored, the new data is immediately pushed to persistent storage, increasing update time. In case of Source Cache, the Androjena RDF graph creation time (shown between brackets) represents the largest part of the add time. This extra step allows improving the data combination time during the Data Query phase (see Chapter 4, section 4.7.2.5). When disregarding this graph creation overhead, the add time amounts to less than for Meta Cache, which is to be expected since less indices need to be updated. At the same time, we observe that cache replacement is less costly for Meta Cache. The Meta Cache keeps smaller and more fine-grained cache units, leading to smaller storage and removal times. Moreover, smaller cache units also allow the actual memory usage to be kept closer to the memory limit. Overall, we note that the performance overhead of both caches is relatively high compared to the Source Index Model, especially due to the extraction and memory management times.

We conclude that, as expected, utilizing a cache component significantly increases memory usage and processing effort, thus reducing adherence to the requirement of reducing memory/performance overhead. Due to more elaborate indices and increased object count, Meta Cache occupies more memory than the Source Cache – SIM solution, and incurs higher insertion times. On the other hand, due to the fine-graininess of the kept cache units, replacement times are lower for Meta Cache. In the end however, given their benefits during the Data Query phase (see next section), the memory and performance overheads of both caches are still acceptable.

5.3.2.2 Data Query phase

Tables 5-10 and 5-11 show the overall execution times and their constituent times. Firstly, we observe that the execution times are much lower compared to when the Source Index Model (SIM3) is employed individually. In case the SIM is used together with the Source Cache, overall execution times are decreased by (avg.) ca. 67%. If Meta Cache is employed, execution times are reduced by (avg.) ca. 81%. Below, we study the results in more detail, and compare Meta Cache to Source Cache performance.

At a first glance, we observe that Meta Cache retrieves data in a much more fine-grained way than Source Cache (see #triples column), leading to much lower cache retrieval times. As such, Meta Cache adheres to our requirement of fine-grained data retrieval. On the other hand, cache misses pose problems for both caches (see miss sub column). We detail these cache access issues in the paragraph below. Regarding data assembly, the average data combination time for Source Cache is lower than for Meta Cache (ca. 1,4s vs. 2,2s), which seems unintuitive since more data needs to be combined. However, as mentioned (see section 5.3.2.1), Source Cache keeps cached source data in Androjena RDF graphs, which can be combined very efficiently. In case of Meta Cache, the retrieved cached triples are added one-by-one to the final query graph, increasing collection overhead. The data assembly column for Meta Cache also includes the number of extra required types (between brackets) for the cached resources in the final query graph. Adding these types is also included in the assembly time. In our experiments, we observe the separate query execution times are mostly bound by the query complexity; while the final query dataset for most queries is much smaller for Meta Cache, the individual query execution times (i.e., time required to execute the query on the already collected dataset) remain more or less the same for both cache organizations. We made a similar observation for the SIM (see section 5.2.2.2).

Looking in more detail at the cache access results (see Tables 5-12 and 5-13), we note that most query-relevant cache units need to be loaded from persistent storage, since most cache units are stored persistently (in either cache organization) due to the relatively low in-memory limit. However, due to the course-graininess of data retrieval for Source Cache (i.e., per source), a lot of data irrelevant to the query needs to be loaded as well, leading to much larger retrieval times. Although cache misses cause serious overhead for both cache organizations, they have the potential to be worse for Meta Cache. For instance, in case of queries 2 and 3, the number of source re-downloads resulting from a cache miss is significantly higher than the total amount of cache misses. This is a direct result from the Meta Cache organization; when a cache unit is removed and later referenced again (cache miss), all sources containing the associated metadata need to be re-downloaded. In case the metadata is contained in a large number of sources, source re-download times thus become very high. In fact, the overhead resulting from cache misses for queries 1 and 3 leads to the overall cache access times to be larger for Meta Cache than for Source Cache.

Finally, Table 5-14 shows the cache maintenance times associated with cache access. Corresponding to our observations during the Source Encounter phase (see section 5.3.2.1), replacement times are exceedingly high for Source Cache. For Meta Cache, the times range from several milliseconds to ca. 5s; while for Source Cache, these range from ca. 1,4m to 3,3m. More course-grained cache units are kept by the Source Cache, yielding larger storage and removal times in case memory space needs to be cleared.

In conclusion, utilizing a local cache yields significantly increased querying performance, which more than compensates for the additional performance overhead during source processing. When comparing both caches, Meta Cache retrieves cached data in a much more fine-grained way, resulting in much lower data collection times and thus better overall execution times. While the SIM makes executing queries on large online datasets feasible, Meta Cache thus makes querying large online datasets much more efficient and realistic. As before, Meta Cache confirms that focusing on source metadata enables a balance between fine-grained data retrieval and memory/processing overheads. At the same time, cache misses have the potential to cause huge problems for Meta Cache (resulting in serious performance decrease for some queries). This issue was also observed in previous work [86]. In the section 5.4, we evaluate a removal strategy aiming to reduce this problem.

5.4 Experiment 3: Removal strategies

This section presents the experiment that evaluates the impact of two removal strategies on Meta Cache, namely Least-Recently-Used and Least-Popular-Sources. Since the source-based Least-Popular-Sources strategy is only applicable on caches that do not organize their data via origin source, the Source Cache is not included in this experiment. The first removal strategy, Least-Recently-Used (LRU), assumes a temporal locality indicating that items that have been recently referenced will likely be referenced again. The second removal strategy, Least-Popular-Sources (LPS), considers the "popularity" of the source in the cache (see Chapter 4, section 4.5.2.2). Note that the download cost of sources can also be taken into account; however, since we aim to avoid network issues (i.e., disconnections or delays) influencing our experiment results (see section 5.1.4.3), we do not consider download times for the purposes of this experiment. Finally, as a source-based removal strategy, the LPS strategy removes data per origin source, which involves decoupling the units of storage, retrieval and removal (see Chapter 4, section 4.7.2.10). Because of this decoupling, persistently stored data may be grouped (clustered) in different ways, namely via retrieval unit (i.e., shared metadata; called *retrieval-level*) or removal unit (i.e., origin source; called *removal-level*); each with their effects on performance.

Section 5.4.1 shows the performance of both removal strategies during the Source Encounter phase, and the effect of the removal strategies on the Data Query phase. Furthermore, the effect of the different data clustering techniques on the performance of the LPS strategy is indicated. In section 5.4.2, we investigate and discuss the experiment results in more detail.

5.4.1 Experiment 3: Results

This section shows the results of the removal strategies experiment during the Source Encounter (section 5.4.1.1) and Data Query (section 5.4.1.2) phases.

5.4.1.1 Source Encounter phase

In this section, we show the experiment results for the two removal strategies during the Source Encounter phase. Table 5-15 shows the in-memory sizes taken up by Meta Cache for both removal strategies, measured each time an additional 1000 sources were encountered. We differentiate between the retained heap size of the payload, which includes the concrete payload data as well as storage and retrieval unit objects; and the size of the removal strategy, comprising the removal units and removal strategy itself. Storage units are responsible for keeping the actual payload, while retrieval units are returned by a cache store in response to a lookup. Removal units are persistently stored or removed by the memory management process in case memory is full (see Chapter 4, section 4.7.2.7). Note that the total space shown also includes other components not separately displayed here (e.g., dictionary encoder, cache store).

LRU – Meta Cache				**LPS – Meta Cache**		
#sources (size)	*in-memory sizes*			*in-memory sizes*		
	total	*payload*	*removal*	*total*	*payload*	*removal*
1000 (81Mb)	15780	12092	796	19504	15782	830
2000 (200Mb)	21783	15228	1283	26046	19278	1496
3000 (336Mb)	28270	17846	1668	35257	24556	1945
4000 (442Mb)	31870	18540	1733	40137	26206	2334
5000 (526Mb)	37125	17708	1112	45975	27905	2325

Table 5-15. *Removal strategies – Source Encounter phase:* Sizes (Kb).

Table 5-16 shows the cache composition after encountering 5000 sources, focusing on the missing data. A missing key represents a metadata combination of which the associated source data was previously removed. In case the key is referenced again (cache miss), the removed sources need to be re-downloaded (see Chapter 4, section 4.7.2.6). The table shows the total number of missing keys, together with the total number of sources to be re-downloaded (or missing sources), in the cache. Furthermore, it illustrates the distribution of required source re-downloads across the missing keys. In particular, we show the number of missing keys associated with a certain range of missing source amounts, namely 1-10, 10-50, 50-100, 100-250 and 250-500. For instance, a missing key in range 1-10 incurs 1-10 source re-downloads in case the missing key is referenced (cache miss). For LPS, we show the results for different weightings of the popularity factors used in the removal value calculation; whereby f1 stands for the popularity of the source data, and f2 for the popularity of the source metadata. These weightings were obtained by either considering only one of the two factors, or the sum of both factors, whereby the impact of one factor is potentially reduced (i.e., divided by a power of 10). Note that since factor *f1/100 + f2* yields the same results as when f1 is not considered, it is not shown in the table.

strategy	*#missing keys*	*#missing sources*	*missing sources distribution*		
LRU	11058	1084	**1-10:** 10854	**10-50:** 196	**50-100:** 1
			100-250: 6	**250-500:** 1	
LPS					
f1+f2	9322	819	**1-10:** 9232	**10-50:** 82	**50-100:** 8
			100-250: 0	**250-500:** 0	
f1	741	1586	**1-10:** 643	**10-50:** 60	**50-100:** 4
			100-250: 21	**250-500:** 7	
f1+f2/100	961	1049	**1-10:** 857	**10-50:** 86	**50-100:** 4
			100-250: 11	**250-500:** 3	
f1+f2/10	2061	752	**1-10:** 1967	**10-50:** 85	**50-100:** 3
			100-250: 6	**250-500:** 0	
f2	10314	785	**1-10:** 10224	**10-50:** 82	**50-100:** 8
			100-250: 0	**250-500:** 0	
f1/10+f2	10266	792	**1-10:** 10176	**10-50:** 82	**50-100:** 8
			100-250: 0	**250-500:** 0	

Table 5-16. *Removal strategies – Source Encounter phase*: Missing data.

In Table 5-17, we summarize the cache maintenance overhead in the Source Encounter phase. The table only shows the overhead of running the removal strategy, since the extraction, add and update operations (see Table 5-9) are not influenced by the utilized removal strategy. For the LPS strategy, we show the experiment results for both data clustering techniques (i.e., retrieval-level and removal-level) and with the *f1 + f2/100* popularity factor weighting, since this struck the best balance between the number of missing keys and sources to be re-downloaded.

LRU	**LPS**	
	removal-level	**retrieval-level**
829	1496	1799

Table 5-17. *Removal strategies – Source Encounter phase:* Replacement times / source (ms).

5.4.1.2 Data Query phase

In this section, we show the effects of the two removal strategies on the Data Query phase. Tables 5-18 and 5-19 show the only time influenced by the chosen removal strategy, namely the cache access time. This time is further broken down into 1/ cache retrieval time, involving accessing indices and loading persistently stored data; and 2/ cache miss time, incurred on a cache miss whereby removed source data needs to be re-downloaded. The resulting total execution time is also shown (note that this total time also includes other constituent times, such as executing the query on the collected data, which are not shown here).

For cache retrieval, we show the total number of retrieved cache elements. In case of LPS, each retrieved cache element (retrieval unit) will require loading one or multiple cache units (storage unit). Therefore, Table 5-19 also indicates the number of loaded cache units between brackets. In addition, the table shows retrieval times for each tested clustering technique; whereby removal-level is indicated by t(a) and retrieval-level by t(b). Regarding cache misses, both tables 5-18 and 5-19 show the total number of misses, accompanied by the resulting amount of sources to re-download (between brackets). Note that, for LPS, we show the cache misses associated with the *f1 + f2/100* popularity factor weighting, which turned out to strike the best balance between the number of missing keys and sources to be re-downloaded (see Table 5-16).

LRU – Meta Cache						
	retrieval		*misses*			
query	*#retrievals*	*time*	*#misses*	*time*	*cache access*	*total*
Q1	563	8023	223 (66)	28373	36396	41443
Q2	5	588	2 (7)	3933	4521	5826
Q3	191	2220	25 (74)	27653	29873	35087
Q4	6	598	0	0	598	1081
Q5	9	22	0	0	22	57904

Table 5-18. *LRU – Data Query phase:* Cache access and query execution (ms).

LPS – Meta Cache									
	retrieval			*misses*		*cache access*		*total*	
query	*#retrievals*	*t(a)*	*t(b)*	*#misses*	*time*	*t(a)*	*t(b)*	*t(a)*	*t(b)*
Q1	784 (935)	13017	17603	5 (5)	1560	14601	19163	21042	25870
Q2	6 (272)	4560	878	0	0	4560	878	6127	2203
Q3	216 (957)	11475	4477	0	0	11475	4477	17429	10924
Q4	6 (467)	5915	579	0	0	5915	579	6439	1404
Q5	9 (2146)	26488	4043	0	0	26488	4043	95538	76370

Table 5-19. *LPS – Data Query phase*: Cache access and query execution (ms).

Table 5-20 shows the number of cache misses for different weightings of the LPS popularity factors. The total number of cache misses is shown for each query, while the resulting number of source re-downloads is shown between brackets. Results for factor *f1/100 + f2* are not shown, since it yields the same results as when f1 is not considered.

LPS – Meta Cache						
query	*f1+f2*	*f1*	*f1+f2/100*	*f1+f2/10*	*f2*	*f1/10+f2*
Q1	68 (26)	1 (1)	5 (4)	33 (16)	79 (28)	75 (27)
Q2	0	1 (156)	0	0	0	0
Q3	111 (74)	0	0	6 (2)	111 (72)	111 (72)
Q4	6 (3)	0	0	0	6 (3)	6 (3)
Q5	0	0	0	0	0	0

Table 5-20. *LPS – Data Query phase*: Cache misses.

Finally, Table 5-21 shows the removal strategy times resulting from cache access. During cache access, the loading of cached data, and adding of new data due to cache misses, may cause the in-memory limit to be exceeded; thus necessitating cache maintenance. We indicate the removal times for removal-level clustering by t(a) and retrieval-level clustering by t(b). Since maintenance occurs after query execution, it is not included in the cache access times shown in the previous tables.

	LRU	**LPS**	
		replacement	
query	*replacement*	*t(a)*	*t(b)*
Q1	4790	54703	57690
Q2	1	170383	211477
Q3	5003	99003	189915
Q4	2	102507	150259
Q5	31	947873	1075984

Table 5-21. *LPS – Data Query phase*: Cache maintenance (ms).

5.4.2 Experiment 3: Discussion

Below, we study and discuss the experiment results when applying the removal strategies on Meta Cache during the Source Encounter (section 5.4.2.1) and Data Query (section 5.4.2.2) phases.

5.4.2.1 Source Encounter phase

Table 5-15 summarizes the relevant cache sizes. We observe that the in-memory payload size is larger for the LPS removal strategy. Since the units of storage and retrieval are decoupled, more objects need to be kept in memory (see Chapter 4, section 4.7.2.10). Furthermore, the LPS removal strategy itself also takes up more memory. Since the removal and retrieval units are decoupled as well, this leads to additional removal unit objects; which are included in the removal strategy size. In addition, due to this separation, the removal strategy component also needs to keep a mapping between the retrieval and removal units.

Before studying cache composition, we shortly recap the Least-Popular-Sources (LPS) strategy. This removal strategy is centered on the popularity of sources in the cache, and considers two factors to estimate this popularity. The first factor indicates the popularity of the source data itself, which is determined by the number of cache units containing the data. By considering this factor, less cache units are influenced by the removal of an individual source, thus reducing the likelihood of cache misses. The second factor pertains to the popularity of source metadata; i.e., how often the source's metadata is found in other sources. A first goal of this factor is to further reduce the number of cache misses. Specifically, this assumes that popular combinations of metadata (i.e., which occur often across the online dataset) are more likely to be referenced by posed queries. Secondly, this factor should keep in check the potential number of source re-downloads per cache miss (see Chapter 4, section 4.5.2.2). Both factors can be weighted differently, influencing cache composition. As mentioned, the download cost factor is not studied in this experiment.

Table 5-16 shows the effects of both removal strategies, LRU and LPS, on cache composition, whereby we tested different weightings for the two LPS factors. In particular, the table focuses on missing data, including the total number of missing keys and missing sources, as well as the distribution of required source re-downloads across the missing keys[100]. Regarding LPS, we observe that the experiment results are in line with the expectations discussed above. As more preference is given to factor 1 (f1), namely source data popularity, the total number of missing keys (i.e., metadata combinations with missing source data) is significantly reduced, compared to LRU and other factor weightings. As such, the likelihood of cache misses is decreased. On the other hand, when factor 2 (f2) is preferred, namely source metadata popularity, the total number of missing sources decreases. At the same time, the number of sources to re-download per missing key is capped as well, as there are no more outliers in ranges 100-250 and 250-500. In our experiment, the f1 + f2/100 configuration reaches a good balance, since the number of missing keys and missing sources are relatively low. However, high-overhead source re-downloads are still possible, whereby 100-250 and 250-500 sources could be re-downloaded as a result from single cache misses. By weighting both factors equally (i.e., f1 + f2), such problematic re-downloads are avoided; but on the other hand, the likelihood of cache misses is again increased, due to the increased number of missing keys. The next section discusses cache misses incurred by the experiment queries for different factor weightings.

Table 5-17 shows the effect of both removal strategies on memory management. The performance overhead caused by LPS is larger than for LRU, and is comparable to the overhead incurred by Source Cache (see Table 5-8). The underlying reason remains the same (see section 5.3.2.1): since the removal unit in case of LPS is more coarse-grained (i.e., per origin source) than for LRU (i.e., per

[100] For instance, the entry "**10-50**: 82" indicates that 82 missing keys yield between 10 and 50 source re-downloads.

metadata combination), memory management is less effective. As such, this increased memory management overhead results from the fact that LPS is a source-based removal strategy. Looking at the results for the LPS data clustering techniques, we observe that grouping data persistently per origin source (removal-level) is less costly than per metadata combination (retrieval-level). This was expected, since a single memory management operation (e.g., storage) works per origin source, likely incurring multiple storage operations for retrieval-level clustering (see Chapter 4, section 4.7.2.11).

Judging from the cache compositions after applying the removal strategies, the LPS indeed allows us to cope with the issue of cache misses, which are problematic in case of Meta Cache. In particular, LPS allows reducing the likelihood of cache misses, and keeping the number of source re-downloads in check. On the other hand, due to the decoupling of storage, retrieval and removal units, LPS incurs more memory overhead. Moreover, due to its source-based nature, memory management is much more expensive, an issue that needs to be resolved in future work.

5.4.2.2 Data Query phase

Tables 5-18 and 5-19 show the component execution times influenced by the employed removal strategy. Table 5-19 shows the LPS results for the f1 + f2/100 factor weighting, which proved to present a good balance between cache misses and source re-downloads (see section 5.4.2.1). Furthermore, we observe that the configuration also yields the smallest number of cache misses for the experiment queries. Later on, we discuss cache misses incurred by different LPS factor weightings in more detail. Regarding cache retrieval, the LPS strategy incurs more performance overhead than LRU (for either data clustering technique). Due to the significantly reduced number of cache misses, an increased amount of cache units are retrieved, most of which need to be loaded from persistent storage. Moreover, due to the separation of retrieval and storage units, a single retrieval likely results in accessing and combining data from multiple storage units (see Chapter 4, section 4.7.2.10).

Table 5-19 shows the number of query-relevant retrieval units, accompanied by the corresponding number of storage units (between brackets). Looking more closely at the retrieval times associated with the two data clustering techniques, we observe that the retrieval time is significantly reduced for the retrieval-level technique. During retrieval, persistently stored data is loaded per unit of retrieval, meaning only a single file needs to be loaded in case of retrieval-level clustering. However, for removal-level clustering, multiple files likely need to be read in order to collect all data (see Chapter 4, section 4.7.2.11). The amount of required read operations is illustrated by Table 5-19. For retrieval-level clustering, the amount of read operations corresponds to the number of retrieval units; for removal-level clustering, this amount equals the number of storage units. Overall, retrieval-level clustering presents a ca. factor 2,2 performance increase during retrieval.

Table 5-20 elaborates on cache misses incurred by LPS for different factor weightings. We observe that when the first factor is preferred, the number of cache misses is reduced as expected. However, at the same time, single cache misses may result in an exceedingly high number of sources to re-download (e.g., *f1;* query 2). As more weight is given to the second factor, the number of resulting source re-downloads is kept in check, while the number of cache misses increases. Finally, Table 5-21 shows the cache maintenance times for both removal strategies. In accordance with observations during the Source Encounter phase (see section 5.4.2.1), the replacement times are much higher for the source-based LPS strategy: for t(a), ranging from ca. 54s to ca. 16m; and for t(b), ranging from ca. 57s to 18m. More course-grained removal units need to be stored or removed, resulting in much higher storage and removal times in case space needs to be cleared.

In line with observations for the Source Encounter phase, we conclude that the LPS strategy allows us to cope with a major problem in Meta Cache, namely cache misses. By fine-tuning the LPS factor weightings, we are able to influence the likelihood of cache misses, as well as the amount of required source re-downloads, for the experiment queries. Regarding the two LPS data clustering techniques, retrieval-level clustering performs much better during querying than removal-level clustering; making it the choice clustering technique, as significantly increased performance during querying is preferable to (slightly) higher source processing times. Consequently, the Meta Cache configuration utilizing the LPS removal strategy, with *f1 + f2/100* factor weighting and retrieval-level clustering technique, yields the best query execution performance.

At the same time, cache maintenance times incurred by the LPS removal strategy are problematic. This issue results from LPS's source-based nature, and will need to be resolved in future work. Therefore, utilizing the LPS strategy at this time yields a consideration between coping with cache misses on the one hand, and increased cache maintenance times on the other.

5.5 Experiment 4: OWA features

In this section, we present the experiment evaluating the effect of the OWA features on the Source Index Model and local cache component. For both components, we consider the best performing variant / organization; respectively, SIM3 and Meta Cache.

Section 5.5.1 shows the memory and computational overhead incurred when enabling OWA features during the Source Encounter phase, and the effect of the OWA features during the Data Query phase. The experiment results are investigated in more detail in section 5.5.2.

5.5.1 Experiment 4: Results

This section shows the results of the OWA features experiment during the Source Encounter (section 5.5.1.1) and Data Query (section 5.5.1.2) phases.

5.5.1.1 Source Encounter phase

In this section, we show the experiment results of enabling OWA features for the SIM and Meta Cache components during the Source Encounter phase.

Type inferencing can be applied in two locations: at the query side (@query-side) and at the source side (@source-side). Clearly, only type inferencing at the source side influences performance during the Source Encounter phase. Furthermore, two sub-options are available, namely 1/ inferring subject and object types based on property domain and range restrictions (indicated as *+dom/ran*), and 2/ additionally inferring subclasses of found types (indicated as *+subclass*). Note that in our experiments, the second option (*+subclass*) is always enabled in addition to the first option (*+dom/ran*), and thus stands for full type inferencing support.

Table 5-22 shows the SIM memory and performance overheads when type inferencing is enabled. Table 5-22/a shows the increased memory space taken up by the SIM (for brevity, only results for 3000 and 5000 sources are shown), while Table 5-22/b shows the extra computational overhead of type inferencing, including the inferencing time itself (shown per source) and ontology retrieval time (shown per ontology). It can be observed from Table 5-22/a that the SIM memory size becomes exceedingly high for 5000 sources when subtypes are also inferred (ca. 63Mb). Therefore, experiments with the *+subclass* option enabled were only run for 3000 sources on the Android device; results for 5000 sources were obtained via the JRE version of the query service.

SIM3		
#sources (size)	**@ source-side**	
	+dom/ran	*+subclass*
3000 (336Mb)	11200	55705
5000 (526Mb)	12401	64115

(a) Memory size (Kb)

SIM3		
	@ source-side	
	+dom/ran	*+subclass*
infer metadata (/ source)	23488	36485
retrieval time (/ ontology)	1210	

(b) Avg. processing times (ms)

Table 5-22. *SIM – Source Encounter phase – Type inferencing (SIM3).*

Table 5-23 shows the type inferencing results for Meta Cache. Table 5-23/a shows the increased memory space taken up by Meta Cache (for brevity, only results for 3000 and 5000 sources are shown), and Table 5-23/b shows the performance overheads. Comparable to the SIM, the size of Meta Cache becomes exceedingly high if subtypes are inferred for 5000 sources (ca. 52Mb). Therefore, we only performed experiments with the *+subclass* option enabled for 3000 sources. The resulting Meta Cache size for 5000 sources was obtained via the JRE version.

Meta Cache		
#sources (size)	**@ source-side**	
	+dom/ran	*+subclass*
3000 (336Mb)	29410	41228
5000 (526Mb)	37867	53163

(a) Memory size (Kb).

Meta Cache		
	@ source-side	
	+dom/ran	*+subclass*
infer metadata (/ source)	12552	18842

(b) Avg. processing times / source (ms).

Table 5-23. *Meta Cache – Source Encounter phase – Type inferencing.*

Type mediation requires resource information to be kept, such as types and origin sources. In order to index this resource information, two index data structures are available, namely a default resource index (keeping a single hash table) and a multi-level resource index (swapping to persistent storage to save memory).

Table 5-24 shows the memory space taken up by both index data structures in case of the SIM. From the table, we observe that the memory size taken up by the default index for 5000 sources is very high (ca. 78Mb). Therefore, we only performed experiments utilizing the default index for 3000 sources on the Android device. We obtained the default index size for 5000 sources via the JRE version of the query service. In Table 5-25, we show the computational overhead of type mediation for the SIM; we separately show the persistent storage and database access times incurred by the multi-level index. We note that for SIM, the mediation time also includes the download time to re-download the necessary sources (see Chapter 4, section 4.7.3.2).

#sources (size)	**default index**	**multi-level index**
3000 (336Mb)	41215	21564
5000 (526Mb)	79453	34567

Table 5-24. *SIM – Source Encounter phase – Type mediation (SIM3)*: Memory size (Kb).

type mediation	**multi-level index**	
	db access	*persist storage*
840	484	6088

Table 5-25. *SIM – Source Encounter phase – Type mediation (SIM3)*: Avg. processing times per source (ms).

In Table 5-26, we show the memory sizes of both index data structures in case of the Meta Cache. Note that type mediation requires different resource information to be kept for SIM and Meta Cache (see Chapter 4, section 4.7.3), resulting in different index sizes and maintenance times for both components. As was the case for the SIM, the memory space required by the default index for 5000 sources is exceedingly high (ca. 71Mb). Furthermore, the memory space taken up by the multi-level index for 5000 sources is also very high (ca. 42Mb). The default index sizes above 2500 sources, and the multi-level index size above 4000 sources, were obtained via the JRE version of the query service.

Table 5-27 shows the computational overhead of the type mediation process for Meta Cache. We again note that, since the type mediation processes for the SIM and Meta Cache are different, they result in different mediation times for both components. In addition, the table shows the storage and database access times for the multi-level index. Due to the aforementioned memory issues, we only performed experiments utilizing the default index for 2500 sources, and the multi-level index for 4000 sources, on the Android device.

#sources (size)	**Default index**	**Multi-level index**
2500 (275Mb)	30654	19700
4000 (442Mb)	59730	27218
5000 (526Mb)	73094	42932

Table 5-26. *Meta Cache – Source Encounter phase – Type mediation*: Memory size (Kb).

type mediation	**Multi-level index**	
	db access	*persist storage*
853	2604	26507

Table 5-27. *Meta Cache – Source Encounter phase – Type mediation*: Avg. processing times / source (ms).

5.5.1.2 Data Query phase

This section shows the effects of OWA features on experiment results during the Data Query phase, for the SIM and Meta Cache components.

Regarding type inferencing, we consider three cases; applying type inferencing at the query side (@query-side), at the source side (@source-side) and at both sides (@both sides). For each case, we indicate the results for each sub-option; namely, infer subject and object types via domain and range restrictions (*+dom/ran*) and additionally infer subtypes (*+subclass*).

Table 5-28 shows the influence of type inferencing on data access for both components. It shows the amount of sources identified by the SIM, together with the number of query results returned by both components (between brackets). For ease of reference, the table also shows the original selectivity, with the original number of query results between brackets. In case the new results differ from the original results, the new results are shown in bold. Note that for both SIM and Meta Cache, inferring subtypes (*+subclass* option) at the source side incurs too high a memory overhead for 5000 sources. Selectivity results for those cases were thus obtained from the JRE version.

query	**original**	**@ query-side**		**@source-side**		**@both sides**	
		+dom/ran	*+subclass*	*+dom/ran*	*+subclass*	*+dom/ran*	*+subclass*
Q1	254 (4)	49 (**0**)	49 (**0**)	254 (4)	254 (4)	215 (4)	215 (4)
Q2	272 (272)	271 (272)	271 (272)	281 (**282**)	313 (**658**)	281 (**282**)	313 (**658**)
Q3	319 (319)	0 (**0**)	0 (**0**)	319 (319)	319 (319)	319 (319)	319 (319)
Q4	87 (77)	87 (77)	87 (77)	87 (77)	87 (77)	87 (77)	87 (77)
Q5	256 (148)	256 (**0**)	256 (**0**)	256 (148)	256 (148)	256 (148)	256 (148)

Table 5-28. *Query Service – Data Query phase – Type inferencing (Meta Cache, SIM3)*: Data access.

In Table 5-29, we show the type inferencing overhead for the SIM during querying. It can be observed that type inferencing at the source side (@source-side) also incurs an overhead during the Data Query phase, since type inferencing needs to be re-applied to the identified sources (see Chapter 4, section 4.6.1). Note that, due to out-of-memory issues, experiments with the *+subclass* option were only run for 3000 sources.

SIM3				
query	**@ query-side**		**@source-side**	
	+dom/ran	*+subclass*	*+dom/ran*	*+subclass*
Q1	109	119	1027565	3296976
Q2	124	797	922511	1023149
Q3	988	4309	2884838	8578673
Q4	0	0	16293	43627
Q5	3260	3268	499867	1693493

Table 5-29. *SIM – Data Query phase – Type inferencing (SIM3)*: Processing times (ms).

Table 5-30 shows the same type inferencing overhead for Meta Cache. As for the SIM, type inferencing at the source side (@source-side) incurs an overhead during querying; since type inferencing needs to be re-applied to online source data retrieved in response to cache misses (see Chapter 4, section 4.6.1). Again, due to memory issues, results are shown for the *+subclass* option at the source side for 3000 sources.

Meta Cache				
	@query-side		**@source-side**	
query	*+dom/ran*	*+subclass*	*+dom/ran*	*+subclass*
Q1	263	293	1120833	1574279
Q2	720	727	319577	638294
Q3	693	3366	4828998	6262059
Q4	1	1	32930	62582
Q5	2169	2180	0	0

Table 5-30. *Meta Cache – Data Query phase – Type inferencing*: Processing times (ms).

Table 5-31 illustrates the effect of type mediation for the SIM on data access. In addition to showing source selectivity, the table indicates the new amount of query results between brackets (in case one of the amounts differs from the original, it is shown in bold). The table also shows the performance overhead during the Data Query phase. In case of the SIM, the original contents of online identified sources need to be synchronized with the previously mediated resource types (see Chapter 4, section 4.7.3.2). This overhead is shown for both indices, whereby the result for the multi-level index also includes persistent reading and database overheads.

SIM3			
		sync overhead	
query	*selectivity*	*default index*	*multi-level index*
Q1	254 (4)	6434	412829
Q2	272 (**273**)	195	191632
Q3	319 (319)	4223	765070
Q4	87 (77)	1190	48129
Q5	256 (148)	5352	367945

Table 5-31. *SIM – Data Query phase – Type mediation (SIM 3)*: Data access & processing times (ms).

Finally, Table 5-32 shows the overhead of type mediation on Meta Cache. For ease of reference, the table again indicates the impact of type mediation on data access, showing the new amount of query results together with the original amount (between brackets); in case the new amount differs, it is shown in bold. Similar to type inferencing, type mediation needs to be re-applied to online source data retrieved in response to cache misses (see Chapter 4, section 4.7.3.3). Due to memory issues, we were only able to run the default index up to 2500 sources; therefore, we only show mediation results for the multi-level resource index, which could be run for a larger amount of sources (4000) and thus yields more cache misses. Since the multi-level index swaps to persistent storage, the shown time includes database access and persistent storage times as well.

Meta Cache		
query	*# query results*	*mediation time*
Q1	4	3734
Q2	**273** (272)	3277
Q3	319	1989
Q4	77	128
Q5	148	0

Table 5-32. *Meta Cache – Data Query phase – Type mediation*: Data access & processing times (ms).

5.5.2 Experiment 4: Discussion

In this section, we investigate the effects of the two OWA features, namely type inferencing (section 5.5.2.1) and type mediation (section 5.5.2.2), on both the SIM and Meta Cache components during the Source Encounter and Data Query phases.

5.5.2.1 Type inferencing

Type inferencing may be applied at the query side, whereby it extends the extracted search constraints, and at the source side, whereby source metadata is enriched. First, we elaborate on the impact of the type inferencing process on the Source Encounter phase. Then, we discuss its influence on the Data Query phase.

Source Encounter phase

Table 5-22/a shows the increased memory usage of the SIM after applying type inferencing. According to this table, inferring types based on domain/range restrictions as well as subclass relations (*+subclass*[101]) yields a factor 7,1 increase (ca. 63Mb). Table 5-23/a illustrates the same memory usage results for Meta Cache, and shows that the *+subclass* option results in a factor 1,43 increase (ca. 52Mb) in storage space. For Meta Cache, additional types simply result in extra index entries and cache unit objects (i.e., associated with new metadata combinations); while for the SIM, additional types give rise to extra map data structures in its multi-level index structure, yielding larger memory overheads. For both components, the *+subclass* option results in exceedingly large memory usage given the maximum memory limit (max. 64Mb), contradicting our requirement of reduced memory usage.

In Table 5-22/b and Table 5-23/b, we observe that type inferencing also incurs a huge computational overhead for both components. The incurred inferencing time with both options enabled (*+subclass* column) is exceedingly high (max. ca. 36s / source), while the *+dom/ran* option also yields huge processing times (max. ca. 23s / source). Furthermore, the necessary ontologies need to be loaded

[101] This option stands for utilizing subclass relations in addition to domain/range restrictions.

from persistent storage, incurring loading times (ca. 1,2s / ontology). This contradicts our requirement of reduced computational overhead.

From the above, we conclude that type inferencing yields a huge overhead during source processing, regarding memory usage and performance, for both components.

Data Query phase

Table 5-28 shows the selectivity results when type inferencing is enabled. Regarding type inferencing at the query side (@query-side), the results indicate that an increased amount of sources is ruled out. This is in line with expectations, as more extensive search constraints (e.g., extra subject/object types) rule out more irrelevant sources (see Chapter 4, section 4.6.1). Compared to the original selectivity, 205 additional sources can be ruled out for query 1, while 1 extra source is excluded for query 2. However, no results at all are returned for queries 1, 3 and 5, while no query-relevant sources are found for query 3. When investigating the results more closely, in case of query 3, an extra subject type was inferred (i.e., wgs:SpatialThing) that did not occur in combination with the subject type specified in the query[102] (i.e., sch:Airport) in our semantic dataset. Additionally inferring subclass types at the query side did not further influence selectivity results. Therefore, we observe that applying type inferencing solely @query-side is not workable; due to the fact that we are dealing with a real-world dataset, where content authors likely do not exhaustively type RDF resources with all applicable types. By applying type inferencing @source-side in in addition to @query-side (see further), this issue is mitigated.

The experiment results show that type inferencing at the source side (@source-side) leads to an increased number of query-relevant sources, as well as extra query results. This is again expected, since an increased amount of source metadata allows for more query-relevant sources to be identified, as well as more relevant query results to be returned. For the second query, 9 additional sources are identified and 10 extra results are returned. On closer inspection, the extra query results originally lacked the explicit type information specified in the query, but otherwise suited the information request. For instance, some person resources were not explicitly typed as foaf:Person, but via their participation as subjects in properties such as foaf:knows and foaf:firstName, which have a foaf:Person domain restriction, the foaf:Person type can be inferred. Additionally inferring subtypes further increases the number of query-relevant sources for query 2, as well as the number of query results.

In case type inferencing is enabled both at the query and source side (@both-sides), we observe a balance between increased selectivity on the one hand, and retrieving extra query-relevant information on the other. The same amount of extra query-relevant data is returned for query 2 (cfr. @source-side); while source selectivity is increased for query 1 (39 irrelevant sources are ruled out). Importantly, no more query-relevant information is lost, which was the case when only applying type inferencing @query-side.

[102]See Appendix B.1.1 and http://wise.vub.ac.be/william/phd/index.htm#experiments for the queries.

Finally, tables 5-29 and 5-30 show the overhead of type inferencing for both components during the Data Query phase. Although the overhead associated with @query-side is relatively low, the @source-side overhead is extremely high. This performance overhead results from re-applying type inferencing to either 1/ each identified and re-downloaded data source (SIM), or 2/ all sources that were downloaded in response to cache misses (Meta Cache). In case of Meta Cache, the processing time for the *+subclass* option ranges from ca. 63s to 104m (!). The necessity of performing this inferencing step during the Data Query phase is a direct result of our specific setting, where data is captured in online files not under our control. Therefore, they cannot be updated with the types inferred during the previous phase.

We conclude that type inferencing indeed improves data access, allowing extra query-relevant data to be found and more irrelevant sources to be ruled out. In order to obtain an optimal result, type inferencing should occur both at the query and source side. However, type inferencing @source-side incurs an extremely high performance overhead, for either component and during both the Source Encounter and Data Query phase; especially when subtypes are additionally being inferred. These performance overheads, which result from querying ontologies on-the-fly, cannot be avoided during type inferencing. Analogously to RDF stores, type inferencing can be switched on / off to suit device capabilities and application requirements.

5.5.2.2 Type mediation

Type mediation also involves indexing RDF resource information (e.g., previously found types). For this purpose, we evaluate two resource index structures: a default resource index, simply keeping a hash table; and a multi-level resource index, which swaps index data to persistent storage to save memory. Below, we elaborate on the impact of type mediation on both components during the Source Encounter and Data Query phases.

Source Encounter phase

Tables 5-24 and 5-26 show the memory space required by the two index structures, respectively for the SIM and Meta Cache components. As expected, the default resource index yields a high memory overhead in both cases, which contradicts our requirement of reduced memory usage (also exceeding the maximum heap size; 64Mb). Tables 5-25 and 5-27 show the performance overhead of the type mediation for both components. Although the multi-level resource index significantly reduces memory usage, it also incurs a large performance overhead caused by database access and persistent storage operations. As such, the multi-level index clearly presents a balance between memory usage on the one hand, and performance effort on the other.

In case of the SIM, the mediation overhead includes the time needed to re-download the necessary sources, which is required to construct the new metadata combinations (see Chapter 4, section 4.7.3.2). For Meta Cache, this overhead includes moving cached triples to new cache units, reflecting the RDF triples' new associated metadata combinations (see Chapter 4, section 4.7.3.3). Since both units are likely stored persistently, this yields persistent read and write costs. Note that the triple extraction process is already optimized, via a disk-backed map data structure keeping the physical locations of triples in persistent storage. This map is used to avoid loading the cache units, from which triples need to be extracted, entirely into memory (see Chapter 4, section 4.7.3.3).

We conclude that the type mediation overhead during source processing is relatively low for both components, compared to the type inferencing overhead (see section 5.5.2.1). On the other hand, the memory-efficient multi-level resource index is required to make type mediation feasible on current mobile devices given their memory restrictions, and incurs a large performance overhead.

Data Query phase

Tables 5-31 and 5-32 illustrate the effects of type mediation on data access, respectively for the SIM and Meta Cache. Due to the up-to-date query service metadata, we expect data access to be improved by having more query-relevant information returned (see Chapter 4, section 4.6.2). From the results, we observe there is only a small impact, as the SIM exhibits the same source selectivity and only 1 extra query result is returned (see query 2). The extra result in question concerns an RDF resource, whereby a first online source referenced the resource but does not specify any types; and another online source stated that the resource is of type foaf:Person.

During source processing, potentially problematic situations, where newly found RDF resources had already been encountered, occurred 2097746 times. In 36009 cases (1,7%), sources specified different types for the same RDF resources; while in 1719 cases (0,08%), new sources specified additional types for previously found resources, necessitating type mediation to update internal data structures. We observe that the latter number (1719) is very low, compared to the number of times different resource types are specified (36009). Typically, an RDF resource is only detailed (e.g., assigned types) in a limited number of sources, while the resource is referenced in a larger number of sources (i.e., simply mentioned as subject/object in a triple). For instance, a person resource will typically only be detailed in its own FOAF profile (e.g., given a foaf:Person type), while other FOAF profiles simply reference the person resource (e.g., via foaf:knows). Therefore, problematic typing issues (i.e., necessitating type mediation) only occur a very limited number of times; namely, after the detailed online sources are processed, and components need to be updated with the new types. The fact that the number of problematic typing issues (0,08%) is very limited, as well as occurrences where sources specify different types for the same resources (1,7%), might explain why related approaches [12, 67, 68] do not consider type mediation.

As shown in both tables, type mediation also involves a performance effort during the Data Query phase. For the SIM component, the original contents of identified sources need to be synchronized with the previously mediated resource types (see Chapter 4, section 4.7.3.2). Regarding the Meta Cache, type mediation needs to be re-applied to sources that were downloaded in response to cache misses (see Chapter 4, section 4.7.3.3). In both cases, this involves accessing the resource index for each individual resource found in the particular data sources, to retrieve their mediated resource types. For the SIM, in case the multi-level resource index is employed, this yields very high persistent read and database access times (ranging from ca. 48s to 13m). For Meta Cache, this leads to relatively low mediation times, since type mediation only needs to be applied to cache-missed online sources (as opposed to all online query-relevant sources). As was the case for type inferencing, this required effort is again a result of our setting where data is captured in online, third-party files, where the online files cannot be updated with the mediated resource types.

We conclude that in our particular online dataset, a very limited number of typing issues occur, and the type mediation process needs to be performed in an even smaller number of cases. Also, the feature only has a limited effect on data access for our specific experiment queries. Compared to type inferencing, the type mediation feature yields relatively low overhead during both phases. However, when utilizing the multi-level resource index to reduce memory consumption and make type mediation feasible on memory-restricted devices, an exceedingly high performance penalty is incurred. To improve performance, further optimization of the multi-level index implementation is needed, and additional research should be performed on more efficient indexing structures.

5.6 Conclusion

In Chapter 4 (see section 4.2), we discussed a number of issues and challenges occurring when querying an online semantic dataset, consisting of small online semantic sources. In order to make querying the retrieved semantic data feasible on mobile devices, the *total query dataset should be reduced*. Furthermore, the online dataset is captured in individual files, meaning the *number of source downloads should be minimized* to decrease the data-retrieval overhead and reduce the impact of connectivity loss. In order to tackle these challenges, we presented two solutions: 1/ identifying query-relevant online sources; and 2/ locally caching data. To further meet our challenges, the following requirements should be fulfilled: 1/ retrieving query-relevant data in a fine-grained way; and 2/ reducing processing effort and memory usage on mobile devices.

In our experiments, we utilized a large, real-world dataset (526Mb) collected from a variety of heterogeneous sources; comprising semantic data on people, places and things (e.g., including products for sale), as well as location-specific and geographical data. Furthermore, we authored five queries that request useful information in a context-aware setting; for instance, allowing interesting nearby entities to be plot on a map view, or indicating the usefulness of physical entities (e.g., product price & comments, displayed exhibition artifacts, etc.) to the mobile user. In addition, these queries reference the range of data types found in the experiment dataset (e.g., products, people, and geographic entities). Consequently, we believe the experiment setup reflects a realistic usage scenario of the mobile query service.

The Source Index Model realizes the first solution of identifying query-relevant sources, while the Source Cache and Meta Cache implement the caching solution. In order to deploy both solutions, the mobile query service either utilizes the SIM together with the Source Cache; or the standalone Meta Cache, which is also able to perform the identification task. These components have proven to suit the requirements, in some cases compromising performance and memory usage to increase fine-graininess of data retrieval. While the lightweight Source Index Model makes querying the online semantic dataset feasible, utilizing a local cache makes the querying performance more realistic, in particular Meta Cache. These observations confirm our initial hypothesis, stating that keeping source metadata yields a balance between fine-grained data retrieval and memory / performance overhead (see Chapter 4, section 4.3).

On the other hand, overall execution times still remain high for Meta Cache, even for the best performing configuration where the LPS removal strategy is applied (ranging from ca. 1,5s to 1,3m; average 23s). Major overheads include persistent loading times, source download times and individual query execution times. The former two times may be kept in check by more fine-grained data retrieval and a suitable removal strategy, but are in the end unavoidable in a scenario where memory and storage are limited. The query execution time is bounded by the underlying query engine (in our case, Androjena), as well as device processing capabilities. For instance, we observe that it takes up to 1m to execute query 5, regardless of the query dataset size. This is a known problem with current mobile query engines; e.g., the MobiSem article [18] already reported issues with accessing relatively small RDF graphs (containing several hundred triples) on mobile devices using the µJena library[103]. In addition, cache maintenance times are very high when data is persistently stored or removed per origin source (e.g., LPS), due to more course-grained removal units and resulting persistent write times. Future work consists of optimizing these cache maintenance operations for source-based removal strategies.

Regarding the LPS removal strategy, we manually chose an LPS factor weighting in our experiments that yielded the best results for our dataset and experiment queries. However, the composition of the semantic dataset likely influences the optimal factor weighting. For instance, previous experiments [86] used a partially synthetic dataset, with fewer metadata combinations that were more evenly spread across the data. In that case, larger numbers of source re-downloads per cache miss were incurred, since the missing metadata combinations typically occurred in many more online sources. Automatically inferring suitable factor configurations, based on the composition of the encountered data, is considered future work.

The OWA features, including type inferencing and type mediation, proved useful in improving data access. Applying type inferencing at both the query and source side resulted in increased data selectivity, as well as more query-relevant data being identified. Although type mediation only had a very limited effect for our particular experiment queries and online dataset, it proved effective at keeping resource type information up-to-date. On the other hand, the type inferencing process incurs extremely high performance overheads; meaning that applying type inferencing in our mobile query service, using current mobile device resources and our straightforward inferencing component (OntologyManager class; see Chapter 4, section 4.6.1), is practically not feasible at this point. Either a more advanced type inferencing component needs to be plugged in, or more powerful mobile devices need to be employed. Regarding type mediation, the incurred memory overhead

[103] http://poseidon.ws.dei.polimi.it/ca/?page_id=59 (access date: 25/06/2013)

necessitates employing a multi-level resource index that swaps to persistent storage, significantly decreasing efficiency and again leading to impractical performance. Future work involves devising a more efficient implementation of the multi-level index. We note that for mobile devices with increased memory (i.e., large available heap spaces) and a reasonable amount of online sources, the default resource index can be employed, which would already significantly improve performance. Furthermore, in case it is a priori known that type mediation is not necessary, either because the online dataset can be extended with missing types or due to a particular composition of the online dataset, type mediation can be turned off.

5.7 Summary

In this chapter, we performed an experimental validation of the mobile query service. In these experiments, we applied a mobile, context-aware scenario where SCOUT plays the role of client. Suiting this scenario, we constructed five queries and collected a real-world dataset, extracted from existing online sources, to be used as online semantic dataset. In particular, we performed an experimental validation of the Source Index Model and cache components, as well as cache removal strategies and the Semantic Web Open World Assumption (OWA) features. Below, we shortly summarize the experiment setup.

In order to investigate the impact of source metadata on data selectivity and memory/processing overhead, we compared different variants of the SIM, each keeping varying amounts of metadata. In order to evaluate the utility of the SIM, we also checked the case where no SIM is used (i.e., native query engine performance). The source processing overhead and memory requirements were contrasted to the increase in source selectivity and overall execution times during querying.

We further studied the two different cache organizations called Source Cache and Meta Cache, whereby cached data is respectively organized via origin source and shared metadata. As for the SIM, we offset performance and memory usage to the fine-graininess of data retrieval and query execution performance. The effects of applying the two different removal strategies, namely Least-Recently-Used (LRU) and Least-Popular-Sources (LPS), were also studied. For the LPS strategy, we tested different weights for the factors in the removal value calculation, and investigated the impact of different persistent data clustering techniques.

Finally, we evaluated the two supported Semantic Web's Open World Assumption (OWA) features, namely type inferencing and type mediation. The incurred performance and memory penalties are weighted against the improvement in data access, yielded by increased source selectivity and additional query-relevant results.

We presented and extensively discussed the experiment results, comparing the performance of the different components, removal strategies and features during source processing and querying, and checked adherence to the formulated requirements (see Chapter 4, section 4.2). Finally, we drew general conclusions from the experiment results.

Chapter 6

Applications using SCOUT

In the previous chapter, we performed an experimental validation of the mobile query service. These experiments were applied on a real-life dataset, retrieved from a variety of heterogeneous existing sources, and involved a mobile context-aware scenario. Different variants of core query service components were evaluated and compared, as well as the effects of the Semantic Web's Open World Assumption (OWA) features. In these experiments, SCOUT played the role of client, utilizing the query service to transparently access online semantic data describing the user's surroundings.

While the previous chapter evaluated data access performance, this chapter presents proof-of-concept applications that utilize the SCOUT functionality. In particular, we discuss three mobile applications that demonstrate SCOUT's use and capabilities, and rely on the context access features to realize their own functionality. Below, we shortly summarize each application.

The **Person Matcher** [31] demonstrates the data filtering capabilities of the SCOUT framework. The application pro-actively identifies nearby people of interest, and pushes the information to the user. To perform this identification, the Person Matcher executes a compatibility check between the user and nearby people, which involves crawling their FOAF networks and checking for overlaps. In order to be notified of nearby people and their online FOAF profiles, the Person Matcher relies on the push-based query access supplied by SCOUT.

COIN (COntext-aware INjection) [29, 32] implements a client-side web augmentation approach, which injects context-aware features into existing websites on-the-fly. In particular, COIN focuses on enriching websites to suit the mobile user's needs, which are often related to his current environment; for instance, where to find a nearby place to have lunch, or the closest metro station connecting to the university campus. In order to perform its augmentation task, COIN thus requires access to rich environment context data. Moreover, COIN should be notified in case of context changes, so injected features can be kept up-to-date. Fulfilling these requirements, SCOUT is utilized to supply the required context data in a push- and pull-based way. Furthermore, the non-centralized nature of SCOUT perfectly suits the client-side COIN approach.

The **AdaptIO** approach [33] dynamically adapts the obtrusiveness of mobile interactions to suit the mobile user's current situation. Due to the ubiquity of mobile devices, mobile interactions may occur in a variety of situations, thus increasing their potential for obtrusiveness. In order to accurately define and determine obtrusive situations (e.g., in a meeting), in any mobile environment, AdaptIO requires rich environment context data. Moreover, any a priori unknown environment should be supported, possibly only minimally (or not at all) outfitted with context-aware infrastructure. For this purpose, we aligned [30] the existing AdaptIO approach with SCOUT, which provides the necessary context data in a push- and pull-based way. Moreover, SCOUT is specially targeted towards sensing context in heterogeneous and minimally outfitted environments.

The chapter is structured as follows. Section 6.1 discusses the Person Matcher application. In section 6.2, we elaborate on the COIN approach, while section 6.3 details the extended AdaptIO application. Finally, section 6.4 summarizes this chapter.

6.1 The Person Matcher

Due to the increased connectivity of powerful new mobile devices, combined with location awareness (e.g., GPS) as well as identification and sensing technologies (e.g., RFID), a realistic opportunity presents itself to automatically deliver content related to the user's current environment. However, mobile users can easily be overwhelmed by the huge amount of information available from their surroundings; most of which is not relevant for the given user at a given time. As such, a main challenge is to filter and personalize the available information, depending on the needs of the user.

The Person Matcher [31] makes use of SCOUT's ability to detect nearby entities, thus demonstrating the pro-active data filtering support provided via the push-based semantic query access. In particular, the Person Matcher is registered to be notified by SCOUT's Environment Notification Service when specific entities, i.e. people, are in the user's vicinity. For each detected person, the Person Matcher performs a detailed compatibility check based on the FOAF profiles of the user and the detected person, and notifies the user when a person of interest is found. This compatibility check is grounded in finding connections between the FOAF networks of the user and the detected person. A configurable weighting scheme specifies the kind of people the user is interested in (e.g., potential collaborators or people sharing interests).

Below, we shortly position the Person Matcher in related work (section 6.1.1). Section 6.1.2 elaborates on the general process implementing the aforementioned compatibility check. In section 6.1.3, we illustrate the mobile user interface.

6.1.1 Related work

To determine compatibility between the user and nearby people, the Person Matcher performs a crawling process of the user's and the detected person's FOAF networks, and finds connections between them. There already exist a number of generic Semantic Web crawlers (also known as "Scutters"), which exploit the rdfs:seeAlso properties to crawl sets of connected RDF files. This includes the RDF Crawler[104] [94], Elmo Scutter[105], and Slug [95]. To our knowledge, no implementation already existed for a mobile environment.

Tools such as FoaF Explorer[106], FoafNaut[107], and WidgNaut[108] visualize FOAF networks, relying on FOAF scutters to provide the necessary information. Furthermore, generic search engines for Semantic Web datasets also exist (e.g., Swoogle [96]). However, none of these tools focus on finding connections between two given FOAF profiles in a mobile setting, and cannot be configured to emphasize certain relationships (e.g., colleagues).

6.1.2 General process

The Person Matcher registers a semantic query with the SCOUT Environment Notification Service (see Chapter 3, section 3.4.3), which returns nearby persons together with their online FOAF profile location. This SPARQL query is shown in Code 6-1 (namespaces omitted for brevity). The query returns nearby persons (type foaf:Person) currently nearby (sm:isNearby) the user (type um:User), together with their online FOAF profile location (indicated via rdfs:seeAlso or foaf:PersonalProfileDocument):

```
SELECT ?person ?foaf_profile
WHERE {
    ?person rdf:type foaf:Person .
    ?stat rdf:subject ?person ;
          rdf:predicate sm:isNearby ;
          rdf:object ?user .
    ?user rdf:type um:User .
    {
        ?person rdfs:seeAlso ?foaf_profile .
    } UNION {
        ?foaf_profile rdf:type foaf:PersonalProfileDocument .
        ?foaf_profile foaf:maker ?person .
    }
 }
```

Code 6-1. *Person Matcher*: Query registered with the Environment Notification Service.

[104] http://ontobroker.semanticweb.org/rdfcrawl/ (access date: 03/05/2013)
[105] http://www.openrdf.org/doc/elmo/1.0/user-guide/x458.html (access date: 03/05/2013)
[106] http://xml.mfd-consult.dk/foaf/explorer/ (access date: 03/05/2013)
[107] http://crschmidt.net/semweb/foafnaut/ (access date: 03/05/2013)
[108] http://www.softpedia.com/get/Windows-Widgets/Widget-Miscellaneous/Widgnaut.shtml (access date: 03/05/2013)

As the mobile user is walking around, the Person Matcher is thus continuously provided with FOAF profiles of people in the vicinity. For these FOAF profiles, the Person matcher computes compatibility with the user's own FOAF profile, using a configurable matching algorithm. Below, we first summarize this matching algorithm responsible for finding connections between two FOAF networks. Then, we shortly discuss the weighting scheme that can be used to configure the matching algorithm to suit different purposes.

6.1.2.1 Matching algorithm

The goal of the matching algorithm is to find connections between two FOAF networks, whereby a connection consists of a sequence of direct or indirect links. A link connects two persons and is based on one or several FOAF properties. In particular, direct links *directly* link a person to another person, and involve a single RDF property (e.g., foaf:knows). Indirect links *indirectly* connect two persons via a number of RDF properties and intermediary resources. For instance, the foaf:made property connects a person to a resource (e.g., document) he made, to which other person may also be connected via foaf:made. Other examples involve properties such as foaf:interest and foaf:workplaceHomepage, whereby two persons are linked in case they share the same interest or workplace resources, respectively. Figure 6-1 illustrates both types of links.

The matching algorithm constructs the two FOAF networks (or graphs) on-the-fly, by crawling the two FOAF profiles concurrently in a breadth-first way. In these graphs, the nodes represent Persons, while edges correspond to direct or indirect links. During the crawling process, online RDF sources associated with relevant resources, including other linked persons or intermediary resources (e.g., co-authored documents), are dynamically retrieved. By leveraging the links found in these retrieved online sources, the graphs are further extended and the crawling process is continued. During this process, the algorithm stops exploring link sequences in case their total score (see below) falls below a configured threshold. The matching algorithm extends both graphs by recursively performing the crawling process, until a predefined number of iterations are reached.

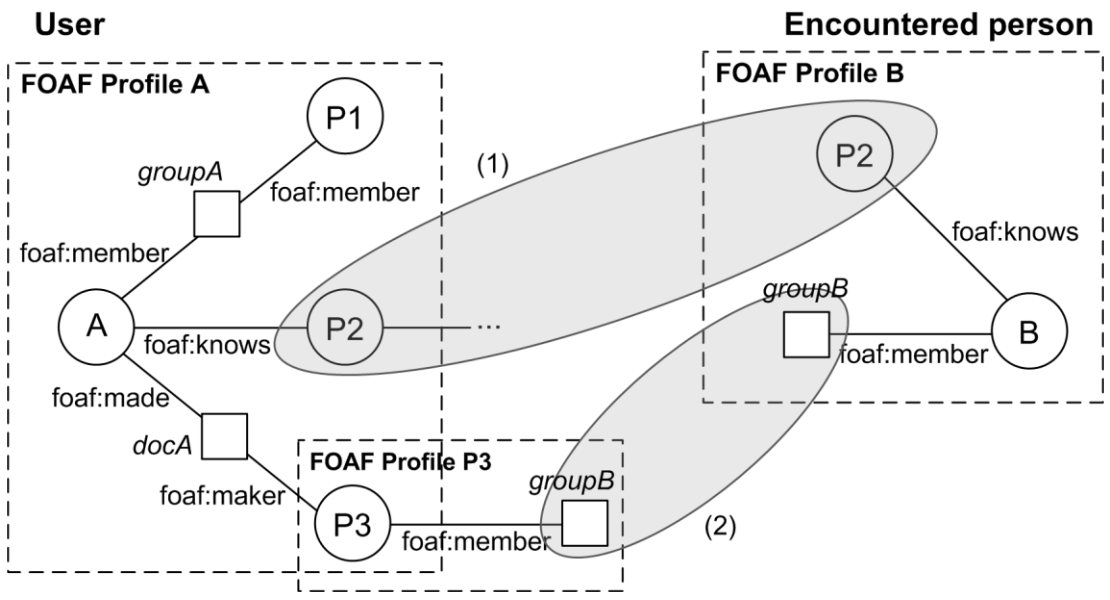


Figure 6-1. *Person Matcher*: Overlaps between to FOAF networks.

A connection is found in case the two constructed graphs overlap; in other words, when one or more link sequences connect the two initial Person nodes. Figure 6-1 illustrates two example FOAF networks and the connections between them. In this case, two connections are found: (1) both the user and the encountered person have a common acquaintance (P2; via foaf:knows); and (2) the user co-authored (via foaf:made) a document with another person (P3) belonging to the same group (foaf:member) as the encountered person. It should be noted that, via the crawling process, the algorithm combines information from multiple online RDF sources, resulting in connections for which the relevant information was not present in any single source.

The compatibility score between two FOAF profiles is the sum of the individual scores of connections between the two graphs. Each connection's score is calculated as follows (j stands for the total number of links in the connection; $weight_i$ represents the weight of the ith link in the connection):

$$score(connection) = \left(\prod_{0<i \le j} weight_i\right)\left(1 - \frac{j}{10}\right)$$

A connection's score multiplies the weights of its individual links. The last factor ensures that the score decreases with the length of the path. Below, we discuss how the link weights are specified.

6.1.2.2 Weighting scheme

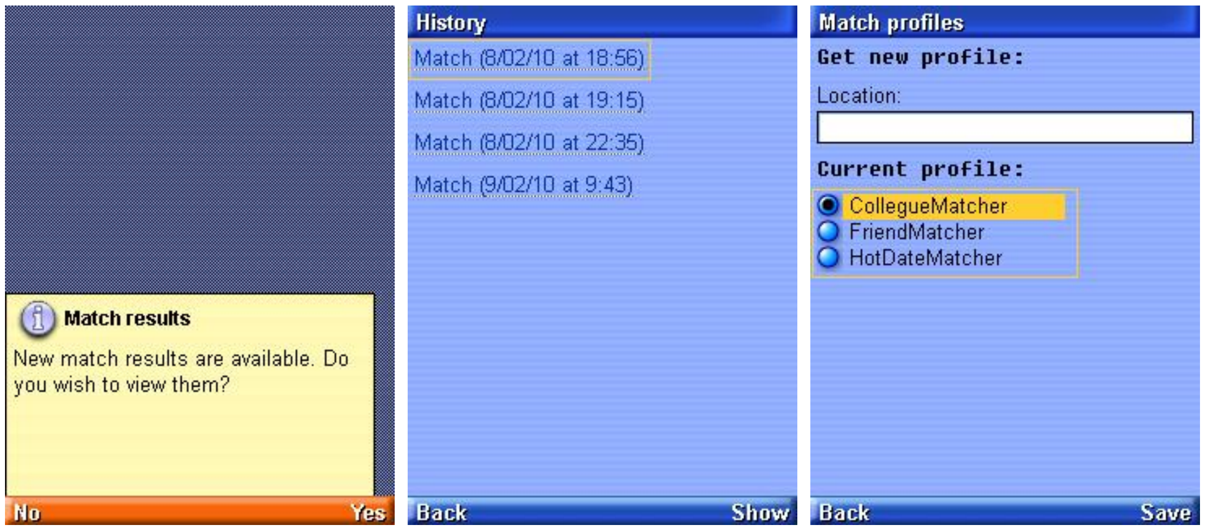
Clearly, whether or not a nearby person is interesting depends on the purpose for which the Person Matcher application is employed. For example, in a conference setting, the user is likely to be interested in finding future colleagues to collaborate with; while in his free time, he is probably on the lookout for people sharing his interests. Depending on the purpose for matching, some sequences of links will become more important when determining compatibility, while others become less important or even irrelevant. For instance, in a conference setting, links involving FOAF properties such as foaf:made (indicating (co-)authorship) and foaf:group (indicating group membership, e.g., research groups) will become important. On the other hand, FOAF properties such as foaf:depicts (indicating co-depiction in photos) become less important or irrelevant.

In order to reflect the reason of use, the Person Matcher application can be configured with different weighting schemes (specified in RDF format). Each scheme identifies relevant FOAF property sequences and their importance (between 0 and 1). In other words, these schemes denote which connections between the two persons should be searched for. Users may specify custom weighting schemes, and utilize the mobile user interface (see next section) to download new schemes and choose between loaded weighting schemes. Two example weighting schemes can be found in Appendix C.1.1 and on http://wise.vub.ac.be/william/phd/index.htm#person_matcher.

6.1.3 Mobile user interface

The Person Matcher is a mobile application that, once started, runs continuously in the background. Whenever a person is encountered and the resulting compatibility calculation is complete, the user is pro-actively notified (see Figure 6-2/a). Figure 6-3/a illustrates the Match Result screen showing the details of the match. This includes the total compatibility score, the name of the matched person, his profile picture (if found in his FOAF profile) and an overview of the found connections. For each connection, the user can obtain its component links and found intermediate persons (see Figure 6-3/b). Details on each link can be retrieved as well, including link type, relevant FOAF properties and intermediary resources (called groupers in the UI; see Figure 6-3/c). Finally, short summaries of persons can also be obtained (not shown).

Overviews of previous results are available as well, including 1/ last 5 persons matched, 2/ best 5 matches of the day and 3/ complete matching history (see Figure 6-2/b). From these overviews, a detailed view of each match can be retrieved (see Figure 6-3/a-c). Finally, Figure 6-2/c shows how the user can choose different weighting schemes (called match profiles in the UI) and download new ones, given an online location.



Match results

New match results are available. Do you wish to view them?

No Yes

History

Match (8/02/10 at 18:56)

Match (8/02/10 at 19:15)

Match (8/02/10 at 22:35)

Match (9/02/10 at 9:43)

Back Show

Match profiles

Get new profile:

Location:

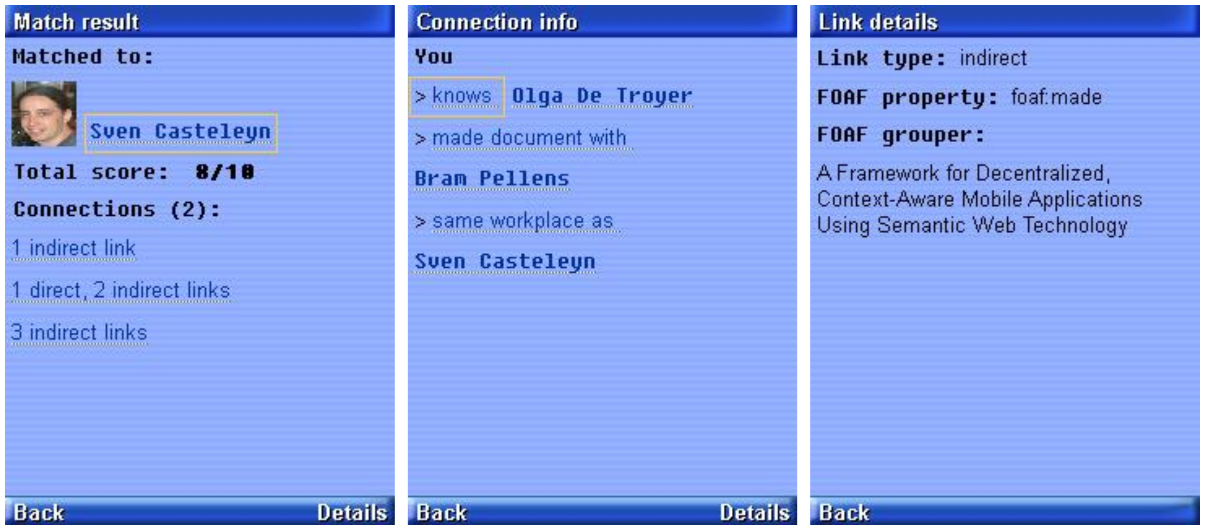
Current profile:

● CollegueMatcher
○ FriendMatcher
○ HotDateMatcher

Back Save

(a) Result notification (b) Match history (c) Configure weighting schemes

Figure 6-2. *Person Matcher:* Mobile UI interaction screenshots.



Match result

Matched to:

Sven Casteleyn

Total score: 8/10

Connections (2):

1 indirect link

1 direct, 2 indirect links

3 indirect links

Back Details

Connection info

You

> knows Olga De Troyer

> made document with

Bram Pellens

> same workplace as

Sven Casteleyn

Back Details

Link details

Link type: indirect

FOAF property: foaf:made

FOAF grouper:

A Framework for Decentralized, Context-Aware Mobile Applications Using Semantic Web Technology

Back

(a) Match result (b) Connection details (c) Link details

Figure 6-3. *Person Matcher:* Match result details screenshots.

The Person Matcher application was written for Java Micro Edition (Java ME; MIDP 2.0, CLDC 1.1) and built on top of SCOUT[109]. The above screenshots were taken from the Java ME emulator.

6.2 COntext-aware INjection (COIN)

COIN (COntext-aware INjection) [29, 32] is a generic, client-side web augmentation approach. The COIN approach enriches existing websites with context-aware features, to better satisfy the needs of

[109] Initially, the SCOUT framework was written in Java ME, but this version was abandoned when more powerful mobile platforms became available (i.e., Android, iOS).

mobile users. By performing this enrichment on existing websites and on-the-fly, i.e. while the user is visiting a webpage, no expensive re-engineering of the website is required. The main contribution of COIN is the automatic enhancement of existing websites with context-aware features; to our knowledge, no other frameworks are available to perform this task. We implemented COIN as a customizable and extendable software framework for the Android platform (version 2.2), where different components and techniques can be plugged in at each step.

Any client-side approach aiming to automatically enrich existing website content requires knowledge on the meaning of the content. As such, COIN is motivated by the increasing availability of semantic information in websites, due to the use of semantic annotation languages (e.g., RDFa[110], microformats[111]). Secondly, to enrich website content with context-aware features, with the goal of suiting mobile user needs typically related to his current environment, COIN requires access to rich environment context data. As the injected features need to be kept up-to-date, COIN should also be made aware of context changes. In order to obtain push- and pull-based access to the required context data, COIN utilizes the SCOUT framework. The non-centralized nature of SCOUT is a perfect fit for the client-side COIN approach, since all necessary context data is locally available. Moreover, the supplied semantic context data can be directly matched to webpage content semantics, reducing the complexity of feature injection.

Below, we discuss state of the art (section 6.2.1). Section 6.2.2 summarizes the three-step, semantics-based process for adding context-aware features to existing websites.

6.2.1 Related work

Most existing websites have been designed for access from desktop computers with sufficient screen size, resources and input capabilities. In order for desktop-oriented websites to be usable in a mobile environment, webpages should be tailored to mobile devices [97, 98] as well as to a user's mobile context [99, 100]. As part of the field of Adaptive Hypermedia, multi-platform context-aware methodologies have been developed that adapt a website according to the user, his device and/or context. For this purpose, the application design can include embedded conditions (e.g., in the form of queries [101]) or context-matching expressions [102]), which reference the user's context to adapt the website to various contextual parameters (e.g., device, preferences). Although such systems allow for the adaptation specification to be hand-crafted towards the specific website, they require extensive engineering at design time, and are not suitable for already deployed, existing websites.

By moving adaptation tasks from the server to the client, existing websites can be tailored on-the-fly, and a more dynamic, adaptive and secure solution can be realized in general. For instance, [103] presents a generic, model-driven WebML [104] website containing a variety of learning objects, whereby a client-side UML-Guide [105] component locally generates hypermedia presentations suiting a particular learning goal. This results in a more dynamic solution, since any personalized

[110] http://www.w3.org/TR/xhtml-rdfa-primer/ (access date: 11/05/2013)
[111] http://microformats.org/ (access date: 11/05/2013)

learning goal can be supported; and more secure, since no private context needs to be passed to the server. So-called transcoding approaches are deployed entirely at the client side, and adapt existing websites at runtime. Although they lack the accuracy of site-specific adaptation engineering, they allow any existing webpage to be tailored to the user. In such an approach, a transcoding module transforms third-party webpages, usually to view them on mobile devices [97, 98]. In fact, most transcoding approaches focus only on adapting to device properties (e.g., small screen), and do not take into account the user's full context. More recently, the MIMOSA [99] platform has been proposed, which takes the user's full context into account. This is work in progress, which (until now) only provides a HTML parsing utility accessible by modules implementing the adaptation logic. This means that modules can only adapt a small set of supported websites, as the knowledge on a website's HTML structure needs to be hard-coded in the module. In contrast, our approach utilizes the high-level semantics of the website, which allows adaptation strategies to be reused across websites from which content semantics can be extracted.

Greasemonkey[112] is a Firefox browser plugin that allows users to easily install and execute so-called user scripts (written in Javascript), which customize visited webpages on-the-fly to suit user needs (e.g., inject price comparisons). As such, these scripts realize client-side web augmentation. However, similar to the transcoding approaches mentioned above, website DOM structures need to be manually hard-coded. Sticklet [84] is a domain-specific language for Javascript, which aids developers in authoring scripts for client-side web augmentation. It employs a metaphor where a *wall* stands for the scope of a particular Sticklet script (i.e., set of webpages), and the wall consists of various *bricks* (i.e., page elements). On these bricks, *notes* can be stuck that represent the augmentation content (e.g., price comparisons retrieved from other web shops). Bricks are selected using XPath expressions. By allowing the author to visually indicate page elements of interest, and applying an inferring algorithm, suitable XPath expressions can be automatically generated. Although the Sticklet DSL thus significantly reduces the authoring effort, scripts are still associated with particular website DOMs, meaning scripts encapsulating a particular customization (e.g., price comparisons) cannot be directly re-used across websites.

In most existing (automatic tailoring) approaches, adaptation only takes place once, either when the page is being generated (i.e., after an explicit user request) or after it has been received by the client. In [106], an extension for WebML is presented, where adaptation can also be triggered based on changes in context (as is the case in our approach). Nevertheless, as for the AH systems discussed earlier, the adaptation logic needs to be explicitly engineered for each web application. In [107], current technologies such as AJAX are exploited to dynamically replace certain parts of a webpage by their adapted counterparts, which is called "instant adaptation". However, only the general possibilities of asynchronous technologies in AH systems are discussed. On the other hand, we present a fully-fledged, generic approach and software framework for on-the-fly webpage adaptation, aiming at context-awareness.

[112] https://addons.mozilla.org/nl/firefox/addon/greasemonkey/ (access date: 20/07/2013)

6.2.2 Generic approach for adding context-aware features

Based on an extensive literature review of (pre-engineered) mobile context-awareness, we formulated requirements for our client-side web augmentation approach [29]. Furthermore, we identified useful adaptation methods, from the domain of Adaptive Hypermedia, to serve as context-aware features in mobile settings. These methods include 1/ context-aware recommendations, where context-relevant items from the website are recommended to the user; 2/ injection of contextual information and aids, whereby contextual information is either directly injected into the visited webpage or used to highlight certain context-relevant content; and 3/ guiding the user through the website, whereby the user is guided towards certain webpages containing relevant information. For a full elaboration on the methods and requirements, we refer to [29].

Based on the formulated requirements and identified methods, we defined a generic step-by-step approach, enabling the injection of context-aware features into existing websites. As a key element, we rely on the extraction of semantic information from webpages, which enables us to identify suitable locations in a webpage where context-aware features can be added. Figure 6-4 provides an overview of the proposed approach. Below, we summarize the different steps. Then, we illustrate the step-by-step approach with a context-aware scenario, which will serve as a running example.

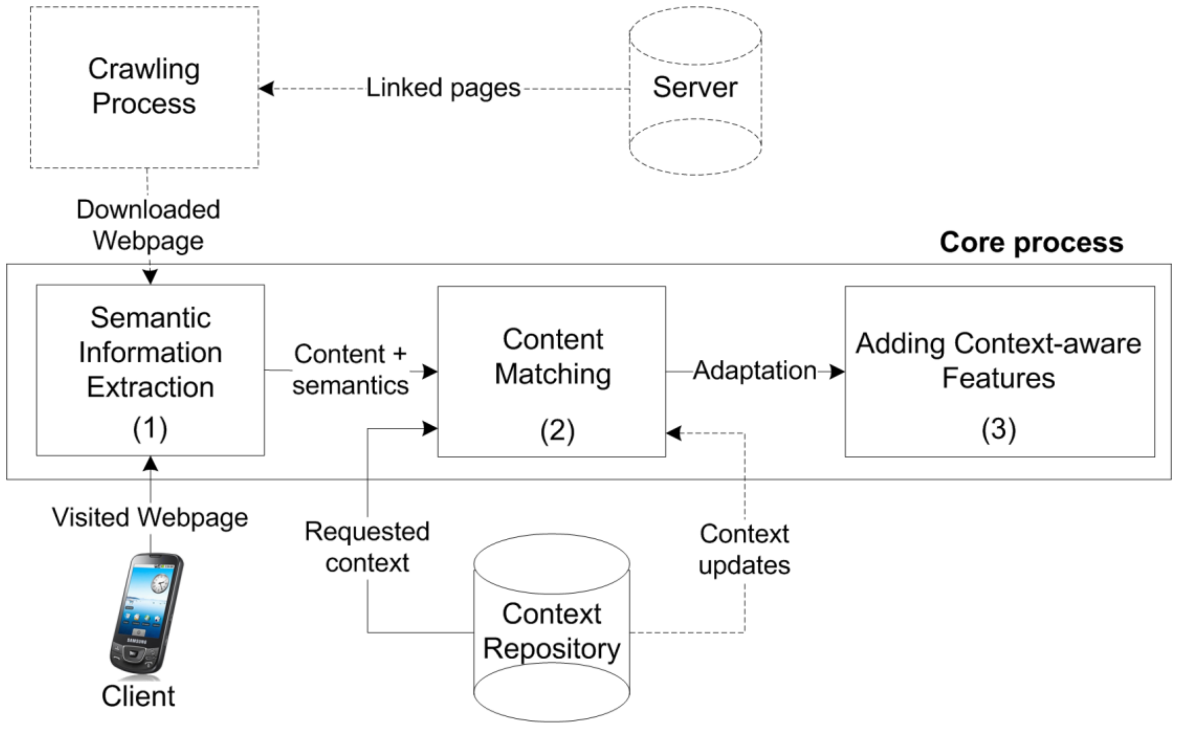


Figure 6-4. *COIN*: Generic COIN approach.

In a nutshell, the approach works as follows. First, semantic information about the content of the website is extracted (step 1). This includes the requested webpage, and optionally also the pages linked by this requested page. By also investigating linked pages, it for instance becomes possible to guide the user through the website. The semantic information extracted from a webpage is necessary to perform the second step, where the content is matched to the user's current context (step 2). The goal of the matching process is to find relevant content that can be enhanced with context-aware features. Access to context data is implemented by the Context Repository, which may also communicate changes in context (see context updates arrow). The COIN software framework relies on SCOUT to serve as Context Repository. Note that, due to its customizable nature, the COIN software framework also allows plugging in other context-provisioning frameworks, given the necessary context data and data access features are supplied. In the third and final step (step 3), one or more context-aware features are injected into the webpage. This is achieved by applying adaptation techniques (borrowed from the field of Adaptive Hypermedia) on the requested page, using the results from the matching process.

To illustrate our approach, let us assume that a mobile user is walking around in a shopping district. While taking a rest, he browses some product websites, such as the BestBuy[113] website, to compare and select products potentially sold by nearby shops. Using our approach, products related to the user's interests will be recommended. Furthermore, proximity information is injected as well, indicating proximity between the user and the shops selling these products. Following our approach, the semantics of the content are first extracted (step 1). In this case, these semantics include unique identifiers for sold products (accompanied by for instance the product manufacturer). Subsequently, the extracted semantics are matched to the user's context. The necessary context data is hereby retrieved via the SCOUT framework. In this example, context includes the user's current location (e.g., via GPS), shops in the vicinity with their locations and sold products (e.g., from a tourist service). Based on the unique product identifiers and positional data, the matching process determines whether any nearby shops sell products mentioned on the shopping website. Finally, the webpage is enhanced with context-aware features (step 3); recommending interesting products and injecting proximity information to the shops selling those products.

In the following sections, we discuss each of these steps in more detail.

6.2.2.1 Semantic information extraction

As we are aiming to inject context-awareness into existing, third-party websites, no specific website content or structure can be assumed. This makes the identification of context-relevant content fragments particularly challenging. Therefore, the first step consists of analyzing the requested webpage, as well as pages linked to the requested page (e.g., to realize guiding the user through the website). This analysis involves extracting the high-level semantics of the website content (in RDF format). By levering these high-level semantics, automated methods can be used in the next step to accurately match page content to the user's context (see next section).

[113] http://www.bestbuy.com/ (access date: 08/05/2013)

An important motivation and enabler for COIN is the increasing availability of semantic annotations in websites via annotation languages such as RDFa, microformats and microdata[114]. These annotations unambiguously define the semantics of website content, and thus provide us with high quality content metadata. Major search providers such as Google, Yahoo!, and Bing currently leverage semantic annotations to enrich search results, while for instance Facebook utilizes annotations to integrate webpages into their social graph (via the Open Graph protocol[115]). As such, these major players strongly encourage the widespread use of semantic annotations. For instance, the BestBuy website from our running example is a large-scale commercial webshop that already utilizes RDFa to annotate its products. At the other end, more and more website tools support the automatic generation of annotations during website / web content authoring (e.g., Adobe Dreamweaver, Wordpress, Drupal, Ruby On Rails). RDFa is currently the W3C recommendation for semantic annotations, and is widely supported both by search engines and authoring tools. An RDFa annotation uses XHTML attributes to insert resource URIs and vocabulary terms, giving explicit meaning to the annotated content. For example, consider the following RDFa snippet (Code 6-2):

```
<li about="http://www.atomium.be" property="rdfs:label">
      Atomium
</li>
```

Code 6-2. *COIN:* Example RDFa snippet.

This RDFa annotation states that the content of the LI element (i.e., the string "Atomium") is the label (i.e., rdfs:label) of the resource <http://www.atomium.be>. The following RDF triple can be extracted from this annotation: `<http://www.atomium.be> rdfs:label "Atomium"`.

For non-annotated websites, site-specific data extraction techniques can be employed to extract content semantics. In [108], so-called wrapper induction approaches are discussed, which allow the rapid generation of site-specific data extractors. However, such techniques have serious drawbacks: most of them are only semi-automatic, meaning the semantics of extracted data needs to be manually assigned in a UI (e.g., [109]). Moreover, the generation and maintenance of a site-specific wrapper is known to be very difficult [17]. As a result, we primarily rely on semantic annotations to obtain content metadata.

The content semantics extracted from a single webpage is stored in an RDF graph called the Page Model, which is made available to the second step (see section 6.2.2.2). The extraction process may (optionally) also be applied to pages linked to the requested webpage. This is done by crawling the website and extracting metadata from each linked page. The crawling process can be limited for performance reasons (e.g., maximum amount of levels), while the extracted semantics can also be cached for later re-use. For the requested webpage, as well as each of the crawled pages, the matching process described in the following section is executed.

[114] http://schema.org/ (access date: 12/05/2013)
[115] http://ogp.me/ (access date: 08/05/2013)

The COIN software framework currently supplies two components for semantics extraction, respectively supporting RDFa annotations and site-specific data extraction. The first component relies on a ported version of the Java-RDFa library[116] to extract RDF triples from RDFa annotations. The second component utilizes a ported version of the Webharvest library[117] to extract site-specific content metadata. This library requires custom XSLT files and Java code to be written for each webpage structure; therefore, it requires significant effort from the developer, and is less robust across page structure changes. Note that COIN allows plugging in any component, for instance implementing support for different types of semantic annotations or data analysis techniques.

6.2.2.2 Content matching

The goal of the second step is to identify website content relevant to the mobile user. This is achieved by matching the content semantics, extracted in the first step, to the user's current context. To realize this matching, we thus first need access to the user's context data. First, we discuss the issues related to this access; next, we discuss the matching process itself.

Access to the user's context

The mobile user's context data is obtained from the Context Repository (see Figure 6-4), which maintains the context model and provides access to its information. In the literature, a variety of context information modeling approaches is described (e.g., key-value, object-role, ontology-based, etc.) [110], as well as frameworks implementing context acquisition, context aggregation / interpretation, and provisioning of context data to applications (see Chapter 2, section 2.2.4). Technically, the customizable COIN framework allows any existing solution to serve as Context Repository, and no specific format for the used context model is required. Instead, the requirements for the context model depend on the context-aware features to be applied. For instance, when emphasizing currently nearby buildings, the user's current position is needed; when highlighting interesting products, the user's interests should be accessible. On the other hand, the choice of context model, and how well it corresponds to the format of the extracted semantics, influences the complexity of matching (see below). Furthermore, any chosen Context Repository technology should notify the adaptation process of relevant context changes, enabling the process to keep injected features up-to-date.

As mentioned, the SCOUT framework is well suited to act as Context Repository in the COIN approach, since it fulfills the aforementioned requirements. SCOUT provides push-and pull-based query access to rich environment context, including the user's location and locations of nearby entities, as well as the user's personal profile. Moreover, since the provided context data is in the same format as the extracted content semantics, the matching complexity is reduced (see below). Finally, the non-centralized nature of SCOUT enables the COIN system to be entirely deployed at the client side.

[116] https://github.com/shellac/java-rdfa (access date: 08/05/2013)
[117] http://web-harvest.sourceforge.net/ (access date: 08/05/2013)

Matching content to the user's context

Before context-aware features can be injected, suitable locations on the webpage need to be identified. This is achieved via a matching process, where the extracted webpage content semantics is matched to the user's context. The actual process logic of this step depends on the particular context-aware feature.

In general, matching identifies page content of which the subject (e.g., `<http://www.atomium.be>`; see Code 6-2) either directly corresponds to specific context model elements (e.g., user's interests), or which is context-relevant according to other criteria (e.g., has an absolute position nearby the user's current position). As such, the complexity of this matching depends on the correspondence between the format of the extracted semantics on the one hand, and the context model format on the other. Both the COIN and SCOUT systems utilize Semantic Web technology as a representation format, thus simplifying the matching process. For example, the use of Semantic Web technology allows utilizing URI identifiers to directly establish equivalence between two resources[118], while RDF properties can be employed to find relations between the data.

The COIN software framework allows developers to specify custom context-aware features. This involves 1/ supplying matching strategies, in order to identify context-relevant page locations; and 2/ choosing and configuring suitable adaptation techniques, to inject features into the identified locations. When specifying a custom matching strategy, the developer first authors SPARQL queries for the Page Model and Context Repository (i.e., SCOUT). Below, we show these queries for our running example.

Firstly, the developer defines a query[119] (see Code 6-3) to retrieve the necessary data from the Page Model, which comprises the extracted webpage content semantics (namespaces omitted for brevity):

```
SELECT ?page_product_id ?page_manufacturer
WHERE {
      ?page_product gr:hasMPN ?page_product_id .
      ?page_product gr:hasManufacturer ?page_manufacturer .
}
```

Code 6-3. *COIN:* Example Page Model query.

This query is applied on the Page Model, returning the unique identification (gr:hasMPN) and manufacturers (gr:hasManufacturer) of products found on the webpage. The second SPARQL query (see Code 3-2) is passed to the SCOUT framework (namespaces omitted for brevity):

[118] Equivalence relations such as owl:sameAs and owl:equivalentProperty can also be leveraged to determine equivalence between different resource URIs or properties.

[119] This is a simplified version of the actual query, since the Page Model contains reified data. For the complete query, we refer to Appendix C.2.1.

```
SELECT ?interest ?user_coords ?entity_coords ?shop_product_id
WHERE {
       ?user rdf:type um:User ;
              um:manufacturerInterest ?interest ;
              sm:lastKnownLocation ?user_pos .
       ?user_pos geo:xyCoordinates ?user_coords .

       ?stat rdf:subject ?user ;
              rdf:predicate sm:isNearby ;
              rdf:object ?entity .

       ?entity rdf:type sumo:RetailShop ;
              region:sells ?shop_product ;
              sm:lastKnownLocation ?entity_pos .
       ?entity_pos geo:xyCoordinates ?entity_coords .
       ?shop_product gr:hasMPN ?shop_product_id .
}
```

Code 6-4. *COIN*: Example context query.

This query selects the manufacturers (um:manufacturerInterest) in which the user (type um:User) is interested, as well as the user's current coordinates (sm:lastKnownLocation; geo:xyCoordinates), together with the absolute coordinates (sm:lastKnownLocation; geo:xyCoordinates) of nearby (sm:isNearby) shops (type sumo:RetailShop) and the unique identifiers (gr:hasMPN) of their sold products (region:sells).

Next to these two queries, the developer specifies how the results from both queries need to be matched to determine relevant page elements. In our scenario, the developer specifies that the results for the ?page_product_id and ?page_manufacturer variables from the Page Model query (see query 1) should be equal to one or more ?shop_product_id and ?interest result from the context query (see query 2), respectively. In this way, we are able to identify products on the page (?page_product_id) sold by nearby shops (?shop_product_id), and with product manufacturers (?page_manufacturers) in which the user is interested (?interest). The developer can also register the second query with SCOUT, allowing the features to be kept up-to-date as the context changes.

In order to plug the custom matching strategy into COIN, the developer creates an Element Selector, which is responsible for identifying page elements suitable for feature injection. In this selector, the two required queries, together with the matching specification (see above), are specified. Clearly, this option is only available in case the Context Repository provides SPARQL query access (e.g., either the repository keeps its data in RDF format, or it supplies a SPARQL layer across the data), which is the case for SCOUT. If this is not the case, or a more complex matching process is required, the developer may override the standard matching process with his own implementation. Element Selectors can be re-used for multiple feature injection scenarios, in case the features are injected into the same page content (e.g., highlight / inject contextual info into *interesting articles*).

In case COIN is alerted of relevant context changes, the matching process will be repeated, based on the matching strategies encapsulated in the configured Element Selectors. Furthermore, during the crawling process (see section 6.2.2.1), the matching step will be re-executed for each crawled page. The results of the matching process, including query results and identified page elements, are passed to the next step, where suitable context-aware features are injected.

6.2.2.3 Adding context-aware features

After the matching process has been performed, the necessary context-aware features are injected into the webpage. To realize this injection, adaptation techniques from the field of Adaptive Hypermedia are applied on the requested page. These adaptation techniques, according to [111] (which builds on the well-known taxonomy described by Brusilovsky in [112]), can be divided into three general categories: content adaptation (e.g., showing/hiding/altering content, emphasizing/de-emphasizing content, dimming, stretchtext), adaptive navigation (e.g., link annotation/generation/hiding, adaptive guidance), and adaptive presentation (e.g., layout changes, sorting/ordering). We also introduce a new adaptation technique, namely (content) fragment annotation, which is not included in the taxonomy from [111] and fits into the content adaptation category. Since we are dealing with existing, third-party webpages, where adaptations may have unforeseeable effects on structure, we encourage developers to use non-intrusive techniques, such as emphasizing / de-emphasizing and stretchtext, as opposed to intrusive techniques such as layout changes and link re-ordering.

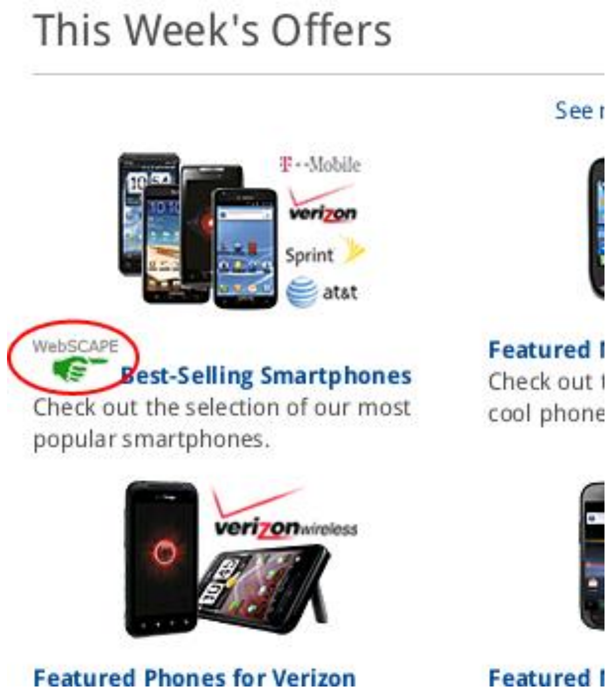
As mentioned, the COIN software framework supports developers in defining custom, context-aware features. A particular feature, such as the recommend, guide and contextual info / aid features, require certain adaptation techniques to be applied on the page (e.g., contextual aid feature involves the link or content annotation techniques). COIN provides a predefined selection of adaptation techniques for these features, which can be fine-tuned or altered by the developer. After choosing or tweaking suitable features, the developer creates a Feature Generator to instantiate these features. In more detail, a Feature Generator instantiates ("generates") suitable features for the page elements identified by an Element Selector (see section 6.2.2.2), and configures them to become context-aware. In order to perform this configuration, the Feature Generator has direct access to the matching query results. For instance, in our running example, the developer first utilizes the positions of the user and relevant nearby shops (obtained from the query results) to obtain detailed proximity information, including walking distance and directions (e.g., obtained via the Google Directions API[120]). Afterwards, the distance is used to configure the annotation color for the contextual aid feature (e.g., green for nearby; red for further away), and the contextual info feature is configured with the walking directions.

After this configuration process, the Feature Generator returns suitable context-aware features to be applied on the page. Analogously to Element Selectors, Feature Generators can be re-used to realize various injection scenarios, in case the same configured features are involved (e.g., *indicate distance to* shops selling page products/theatres playing movies on page).

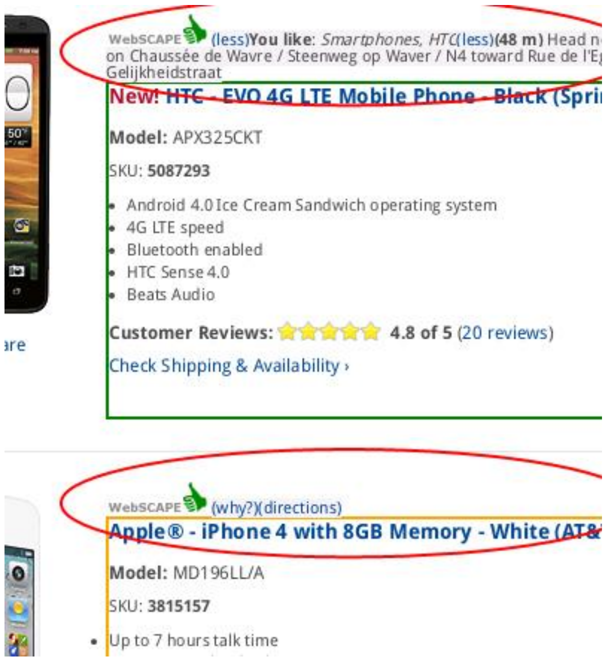
[120] https://developers.google.com/maps/documentation/directions/ (access date: 12/07/2013)

As the mobile user's context changes, page adaptations need to be kept up to date over time. In case of a context change, the matching process is re-executed (see previous section), whereby the new matching results are passed to this step. In response, suitable context-aware features are re-generated by custom Feature Generators. To avoid the same features being injected multiple times, and to reduce the complexity of the feature generation code, the COIN software processes the instantiated features behind-the-scenes. In particular, it checks whether the newly generated features require 1/ injecting new features; 2/ altering existing features, if the features were already injected with different parameters; 3/ removing existing features, in case the features were no longer generated (i.e., the particular page content is no longer context-relevant); or 4/ whether the generated features may be ignored, in case the exact same features were already injected. This relieves the developer of the complexity of these tasks.

Existing Element Selectors, implementing a particular matching strategy, can cooperate with different Feature Generators to realize a range of context-aware feature injections. The developer specifies the cooperation between Element Generators and Feature Generators in a separate configuration file.



a) Guiding the user


WebSCAPE (less)You like: *Smartphones, HTC*(less)(48 m) Head n
on Chaussée de Wavre / Steenweg op Waver / N4 toward Rue de l'E
Gelijkheidstraat

New! HTC - EVO 4G LTE Mobile Phone - Black (Spri

Model: APX325CKT

SKU: **5087293**

- Android 4.0 Ice Cream Sandwich operating system
- 4G LTE speed
- Bluetooth enabled
- HTC Sense 4.0
- Beats Audio

Customer Reviews: ★★★★★ **4.8 of 5** (20 reviews)

Check Shipping & Availability ›

WebSCAPE (why?)(directions)

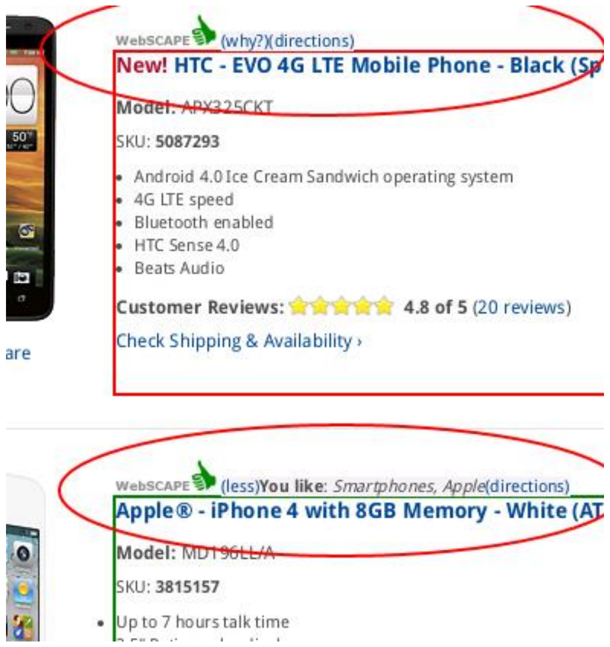
Apple® - iPhone 4 with 8GB Memory - White (AT&

Model: MD196LL/A

SKU: **3815157**

- Up to 7 hours talk time

b) Relevant page



WebSCAPE (why?)(directions)

New! HTC - EVO 4G LTE Mobile Phone - Black (Sp

Model: APX325CKT

SKU: 5087293

- Android 4.0 Ice Cream Sandwich operating system
- 4G LTE speed
- Bluetooth enabled
- HTC Sense 4.0
- Beats Audio

Customer Reviews: 4.8 of 5 (20 reviews)

Check Shipping & Availability ›

WebSCAPE (less)You like: *Smartphones, Apple*(directions)

Apple® - iPhone 4 with 8GB Memory - White (AT

Model: MD196LL/A

SKU: 3815157

- Up to 7 hours talk time

c) After context updates

Figure 6-5. *COIN*: Shopping website screenshots.

Figure 6-5 shows the results of feature injection on the shopping website from our running example. In Figure 6-5/a, the guide feature is injected in the visited webpage, which guides the user towards a page containing relevant products. This feature is realized by applying link annotation, which inserts a recognizable "pointing" icon next to the link. When the user arrives at the relevant page (see Figure 6-5/b), COIN injects the recommend feature, thus recommending products of interest to the user. This feature additionally indicates why the product is being recommended (see "why?" keyword). The recommend feature is realized by applying the content annotation and stretchtext adaptation techniques. For instance, by clicking on the "why?" keyword, the reason for recommendation is revealed, which can be re-collapsed by clicking the keyword again.

Furthermore, the user's proximity to the shops selling the product is indicated. Firstly, the contextual aid feature is injected to indicate the distance to the shop; the green annotation border indicates the user is less than 50m from the shop; orange indicates a distance between 50m and 100m; and red indicates a larger distance. Secondly, the contextual info feature is applied, which inserts walking directions to those shops. Respectively, these two features are realized via the content annotation and stretchtext adaptation techniques. By clicking the "directions" keyword, the walking directions are revealed (see Figure 6-5/b). Finally, Figure 6-5/c illustrates how injected features are kept up-to-date. As the mobile user is walking around, the shop selling the first smartphone is no longer nearby (red annotation border); while one or more shops selling the second smartphone have become nearby (green border).

Regarding implementation, the configured adaptation techniques are applied on the currently visited webpage via a custom Javascript component. We developed a custom Javascript library that implements each of the adaptation techniques described above. The Android-based COIN software communicates with this Javascript component over HTTP via the loopback interface (i.e., localhost).

For all information regarding the running example, including the Element Selectors and Feature Generators code, Page Model and Environment Model instances, as well as the complete queries, we refer to http://wise.vub.ac.be/william/phd/index.htm#coin. Appendix C.2.1 contains the complete queries.

6.2.2.4 Deployment

The COIN software framework consists of an Android application (tested on Android 2.2 using a Samsung Galaxy S) and a single Javascript file, comprising the Javascript library (see previous section). By installing our Mobile Firefox plugin (tested in Mobile Firefox 5.0), the Javascript file is inserted into a visited webpage automatically or on user-request, putting our Javascript library in place.

6.3 AdaptIO

Due to the ubiquity of mobile devices, mobile service interactions (e.g., agenda notifications, incoming calls) may occur in any situation, thus increasing their likelihood for obtrusiveness. Examples of obtrusive interactions include receiving loud notifications while in a meeting, or receiving personal messages while in company of others. To avoid such situations, the obtrusiveness of mobile interactions should be automatically adapted to suit the user's current situation [113, 114]. When focusing on the mobility aspect, obtrusiveness adaptation should furthermore be applicable in any previously unknown, newly discovered heterogeneous environment.

In order to determine the obtrusiveness of interactions, most approaches currently either rely on (semi-)automatic learning techniques [115] or on designer knowledge [6]. However, automatic learning techniques require training data [116] and time to adjust to new user behavior; while the designer cannot capture all situations influencing interaction obtrusiveness, for all users and in initially unforeseen environments (e.g., at a theatre or at work). Clearly, the only stakeholder capable of defining such situations accurately and unambiguously is the user himself. Furthermore, solely relying on local context data collected by sensors (e.g., microphones) or applications (e.g., agenda), as is done by other approaches [38], leads to inaccuracy and ambiguity in a priori unknown environments. For instance, simply turning up the ring volume in loud areas would not work while watching an action movie in a theatre. In order to allow defining and determining situations in previously unforeseen environments, we thus need to rely on rich environment data. For instance, by utilizing descriptive environment data, the user can specify that he is in a "quiet-place" whenever he is inside a place of type "Theatre", thus defining situations in a more fine-grained and generic way.

Our approach [30] adapts mobile interaction obtrusiveness in newly discovered environments, by exploiting rich environment context and putting the user in charge of defining relevant situations.

When considering the requirements for our approach, any solution should support sensing the necessary environment context data in heterogeneous environments, minimally (or not at all) outfitted with context-aware sensors. Furthermore, it should be fully autonomous, since no local server infrastructure may be assumed (e.g., deploying an OSGi[121] installation [33, 117]). AdaptIO [33] is an existing obtrusiveness adaptation approach for mobile interactions, which is deployed on a centralized server (combined with a lightweight mobile client supplying interactions). In order to make the AdaptIO system autonomous and remove reliance on a server infrastructure, it was moved to the mobile platform. Furthermore, to support a priori unknown mobile environments, the AdaptIO approach was extended and aligned with the SCOUT framework. To sense rich environment context, SCOUT utilizes various detection techniques in parallel, thus supporting a range of heterogeneous environments. SCOUT runs entirely on the client-side, suiting the autonomous nature of the approach. Furthermore, SCOUT's push-based query support allows mobile interactions to be adapted dynamically, as the mobile user encounters new situations. Finally, we extended SCOUT with semantic services support, in order to enable interaction with remote services in newly discovered environments; and allow adapting the obtrusiveness of such interactions as well.

We further developed a user-friendly mobile interface, which enables users to expressively define the particular situations in which obtrusiveness adaptation should occur. In order to validate our interaction adaptation approach, where the mobile user becomes a major stakeholder, we evaluate the usability of the mobile user interface by means of a preliminary user study.

This section is structured as follows. The next section positions our approach in the state of the art (section 6.3.1). Section 6.3.2 provides an architectural overview. In section 6.3.3, we summarize the approach methodology, discussing the tasks that need to be performed by the different stakeholders. Section 6.3.4 shows the evaluation results.

6.3.1 Related work

Several studies [115, 118] have been conducted on automatically adapting the modality configurations of mobile devices, based on user context. However, their focus is on context recognition, not on the modality configuration itself and how it influences obtrusiveness. Moreover, they rely on the designer to define the different user situations. In the area of mobile interaction obtrusiveness, research focuses on minimizing unnecessary interruptions for the user [119]. This problem has been addressed directly by means of models of importance and willingness [6]. Also, [114] uses context-aware mobile devices to calculate the adequate timing for interruptions. Sensay [38] infers the user's context from a variety of sensed data, and determines whether or not the phone should interrupt the user while in regular communications. This research focuses primarily on determining *when* to interrupt for a particular application (i.e., either an interaction is allowed or not). In contrast, our approach performs an *adaptation* of interaction obtrusiveness, potentially

[121] http://www.osgi.org/ (access date: 11/07/2013)

reducing the obtrusiveness of interactions to suit the user's current situation. Furthermore, as far as we know, no approach provides support for newly discovered services.

A number of approaches aim to facilitate mobile devices in interacting with newly discovered smart environments. For instance, the Smart Objects For Interactive Applications (SOFIA) project [120] interacts with new, heterogeneous smart environments (e.g., with legacy services, different data formats) by providing mobile applications with shared, interoperable information spaces. In [121], personalized service access is supported across different, heterogeneous environments. However, these approaches require environments to be outfitted with extra middleware, deploying their specific software. On the other hand, Mobile Ad-hoc NETworks (MANETs) enable mobile applications and services to directly discover and communicate with each other, without requiring an existing infrastructure (e.g., via event-based communication) [122, 123]. MANETs allow ad-hoc, loosely coupled communication with newly discovered services; however, approach-specific software needs to be deployed on each component.

In contrast, we rely on open, minimally outfitted, and standards-based environments that do not require middleware. Instead, services are semantically described, and any coordination work is delegated to the client. In addition, by relying on well-known standards, any client can discover new services and interact with them, without requiring support for specific approaches.

6.3.2 Architecture overview

Our integrated system (see Figure 6-6) comprises three layers: the *Environment Discovery and Management Layer*, utilizing SCOUT to discover and manage previously unknown environments; the *Services Layer*, comprising interactive mobile services; and the *Obtrusiveness Adaptation Layer*, which employs AdaptIO to adapt mobile interaction obtrusiveness. Both the SCOUT framework and the AdaptIO system have been extended to support our goals; also, the AdaptIO system was moved to the mobile platform to ensure autonomy[122]. Figure 6-6 indicates new components for both systems in orange. Below, we elaborate on each of the layers.

6.3.2.1 Environment Discovery and Management Layer

This layer discovers a priori unknown (smart) environments, collects context data and enables interaction with environment services. To achieve this, it relies on the SCOUT framework. For context collection, SCOUT utilizes multiple detection techniques in parallel, relying on technologies such as QR codes, RFID/NFC, Bluetooth and GPS, thus supporting a range of heterogeneous mobile environments. To enable interaction with the user's mobile environment, SCOUT was extended with semantic services support.

[122] Initially, the AdaptIO system was deployed on the OSGi platform.

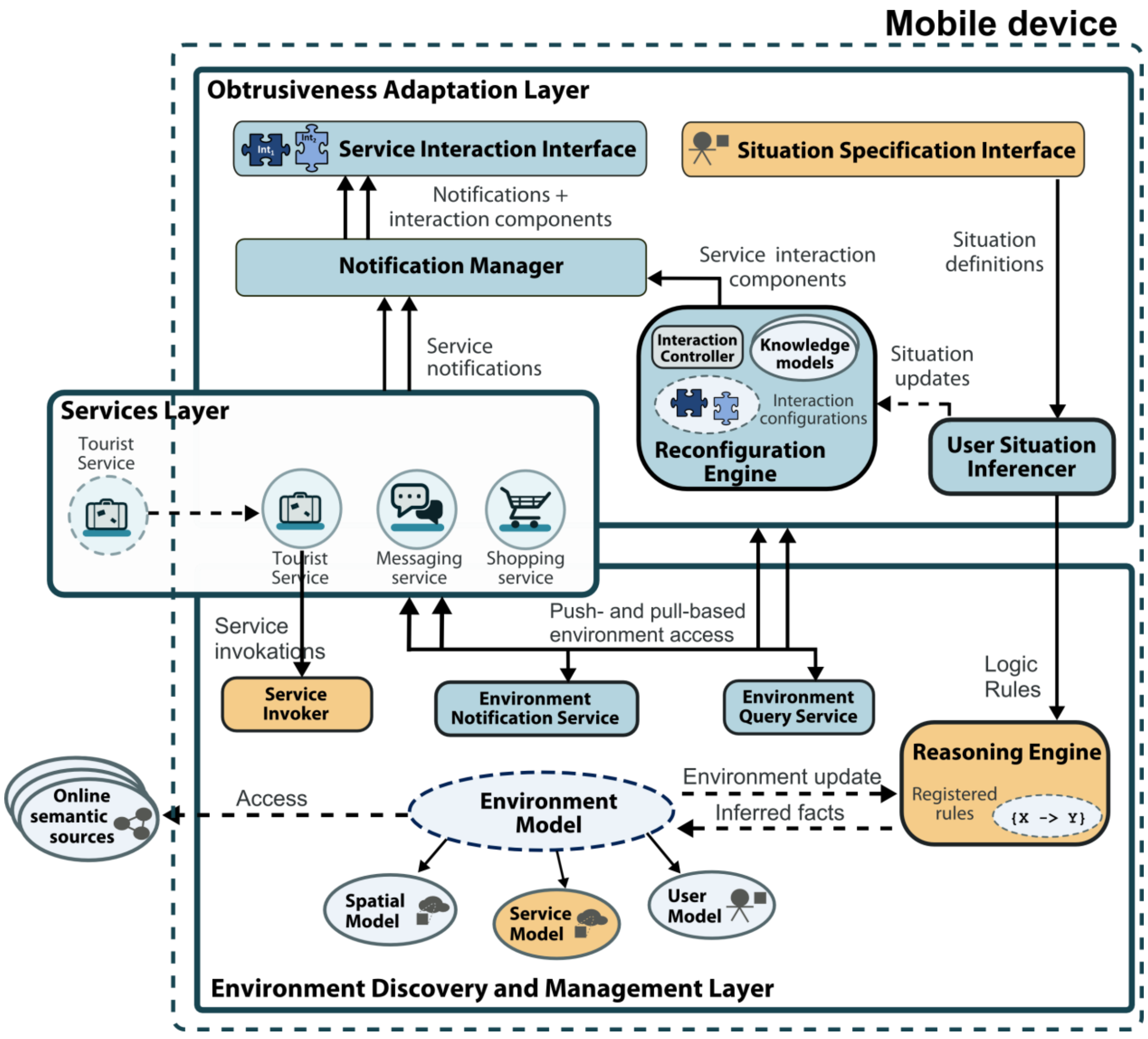


Figure 6-6. *AdaptIO:* System architecture overview.

Figure 6-6 (bottom layer) shows the SCOUT framework architecture. Below, we shortly recap existing parts, and discuss new models and components supporting the obtrusiveness adaptation approach.

Via the **Environment Notification Service** and **Environment Query Service**, mobile applications are provided with push-and pull-based query access to the *Environment Model* (see Chapter 3, section 3.4.3). This model keeps rich environment context and is comprised of the *User Model*, storing the user's personal profile; the *Spatial Model*, specifying which people, places, things and services are spatially related (i.e., nearby or contained in one another); and *online semantic data sources*, which provide semantic data on such nearby entities. We further extended the Environment Model with the *Service Model*, storing the services detected in the user's environment (see below).

To support the upper layers in accurately inferring the mobile user's current situation, SCOUT was extended with a general-purpose **Reasoning Engine**. Each time the mobile user's environment context changes, the engine (re-)evaluates the registered rules, potentially inferring new context facts. By encoding the user-defined situations (see section 6.3.3.2) as rules and registering them with this engine, the user’s current situation (i.e., new fact) can thus be automatically inferred.

We extended SCOUT with mobile services support, which enables the discovery of remote services in the user’s environment and interacting with them. The *Service Model* keeps semantic descriptions of detected services in the user’s environment. In line with our goal to support interaction adaptation across new, heterogeneous environments, SCOUT focuses on *lightweight* smart environments. Such environments are outfitted with various sensing, actuation and information services that contain only the required service hardware and no external middleware. Instead, any discovery, invocation and orchestration work is delegated towards the client. To make this feasible, SCOUT relies on environments that are fully standards-based and contain semantically described services. Specifically, SCOUT relies on the following semantic service stack. The W3C Semantic Annotations for WSDL and XML Schema[123] (SAWSDL) defines mechanisms to complement technical service descriptions (written using the W3C Web Service Description Language[124] or WSDL) with concrete semantics. The Web Service Modeling Ontology (WSMO)-Lite[125] builds on the SAWSDL mechanisms and extends them with a concrete ontology to semantically describe services. SCOUT converts and adds the online semantic service descriptions to the Service Model in RDF format, making them part of the Environment Model. Mobile applications can register a discovery query with the Environment Notification Service (or use the Environment Query Service), to find useful (nearby) services offering specific functionality (see section 6.3.2.2). Mobile applications interact with discovered services via the **Service Invoker**.

In order to convert WSDL descriptions (with SAWSDL annotations) to RDF, part of the SOA4ALL iServe[126] project code was extended and ported to Android. The Service Invoker uses the kSOAP2 library to interact with SOAP services, while the Reasoning Engine is based on the Androjena[127] general-purpose rule engine.

6.3.2.2 Services Layer

The Services Layer (see Figure 6-6) comprises local and remote services that interact with the mobile user. Typically, plenty of local services or applications are running on a user’s mobile device (e.g., agenda, messaging services, shopping app), which may for instance notify the user in case of important events (e.g., agenda deadline, urgent message, good nearby deals).

By leveraging the SCOUT services support (see section 6.3.2.1), remote services can also be plugged in, making their interaction capabilities available on the device. For instance, in Figure 6-6, a local tourism application enables remote tourist services to provide the user with information on good

[123] http://www.w3.org/2002/ws/sawsdl/ (access date: 11/05/2013)
[124] http://www.w3.org/TR/wsdl (access date: 11/05/2013)
[125] http://www.w3.org/Submission/WSMO-Lite/ (access date: 11/05/2013)
[126] http://technologies.kmi.open.ac.uk/soa4all-studio/provisioning-platform/iserve/ (access date: 11/05/2013)
[127] http://code.google.com/p/androjena/ (access date: 11/05/2013)

nearby hotel deals, and nearby points-of-interest. As another example, a local movie application enables remote movie ticket services to notify the user, in case the particular movie theatre plays movies on the user's "to-watch" list. Such local applications register a discovery query with the Environment Notification Service from the Environment Discovery and Management Layer (see section 6.3.2.1). In case a relevant remote service is encountered, the application is notified, and utilizes the Service Invoker for remote communication. Based on the received data, the application provides notifications. For instance, these notifications can inform the user of good hotel deals in the vicinity (tourist service), or whether nearby movie theatres play interesting movies (movie service). Potentially, the user can hereby be provided with a link to a webpage where hotel bookings (tourist service) or movie tickets (movie service) can be purchased.

The concrete service discovery scenarios and queries can be found in Appendix C.3.1, as well as on http://wise.vub.ac.be/william/phd/index.htm#adaptio.

6.3.2.3 Obtrusiveness Adaptation Layer

This layer intercepts interactions from mobile services in the Services Layer (see Figure 6-6), and adapts their obtrusiveness to suit the user’s current situation. Afterwards, it passes on the adapted interactions to the user. The layer utilizes and extends the AdaptIO system, a mobile adaptation approach that adapts service interaction obtrusiveness at runtime. AdaptIO is a model-based approach, where service designers declaratively specify the interaction adaptation behavior of their services in *knowledge models* (see section 6.3.3.1; for more information, we refer to [33]). In more detail, AdaptIO receives notifications from mobile services; chooses appropriate interaction resources (e.g., dialog, sound), based on the knowledge models and the user’s current situation; and utilizes those resources to present the interactions to the user. Below, we elaborate on the main components and the communication between them.

Firstly, AdaptIO was extended with the **User Situation Inferencer**, which determines the user’s current situation and notifies other components of changes. This component encodes the user-supplied situation definitions (see Situation Specification Interface below) as logic rules, and passes them on to the Environment Discovery and Management Layer. There, the Reasoning Engine uses them to accurately infer the user’s situation. If a new situation is inferred, the layer’s Environment Notification Service notifies this component.

The **Reconfiguration Engine** determines which high-level interaction resources should be used for each service’s interaction, based on 1/ the user’s current situation and 2/ the aforementioned *knowledge models*. If alerted by the User Situation Inferencer of a new user situation, the engine consults these knowledge models to retrieve the interaction resources that best suit the user’s new situation. The **Interaction Controller** converts these abstract interaction resources (e.g., dialog) to concrete platform-specific (e.g., Android) interaction components, thus decoupling the knowledge models from the concrete deployment platform.

The **Notification Manager** receives pro-active interactions (i.e., notifications) from mobile services and relays them, together with the service's latest interaction components obtained from the Reconfiguration Engine, to the **Service Interaction Interface**. This interface displays the notifications to the user, employing suitable interaction components.

Finally, AdaptIO is extended with a **Situation Specification Interface**. This interface allows users to expressively define their situations, utilizing the environment context data from the Environment Discovery and Management Layer. Situation definitions are passed to the User Situation Inferencer, allowing the component to infer the user's current situation. Section 6.3.3.2 elaborates on the UI.

The Reconfiguration Engine is based on MoRE [120], which we ported to Android and is based on Autonomic Computing principles [124]. To query the knowledge models at runtime, we rely on a ported version of the Eclipse Modeling Framework Model Query[128] plugin. The model-handling operations are described in [125].

6.3.3 Methodology

In this section, we elaborate on the approach methodology and detail the tasks that need to be performed by the two stakeholders, namely the service designer and user. The designer is responsible for creating the knowledge models, which capture the service's desired behavior for adapting interaction obtrusiveness. On the other hand, the user is in charge of specifying situations across which the obtrusiveness of interactions differ (e.g., in a meeting vs. free-time). Supporting the user in this task, our approach provides a user-friendly mobile interface called the Situation Specification Interface, which leverages environment context. Below, we elaborate on the designer's tasks (section 6.3.3.1). Section 6.3.3.2 discusses the Situation Specification Interface.

6.3.3.1 Service Designer: Adaptation Behavior Specification

In order to model the interaction obtrusiveness of services, we use the conceptual framework for implicit interactions presented in [126]. This framework defines two dimensions to characterize interactions: initiative and attention. Regarding the initiative factor, our approach focuses on proactive interactions (or notifications), where the system takes initiative and the user is potentially interrupted. The attention factor concerns an interaction's attentional demand, which can be represented on an axis. For our purposes, we divided the attention axis in three segments: *invisible* (user does not perceive the interaction), *slightly-appreciable* (user does not perceive the interaction, unless he makes an effort), and *user-awareness* (user is completely aware of the interaction, even while performing other tasks).

In our approach, the service's potential levels of interaction obtrusiveness correspond to the attention axis segments. It can further be observed that, depending on the user's situation, the service's currently desired obtrusiveness level will vary. To capture this information, the designer creates the first knowledge model, namely an *obtrusiveness model*, which contains a state machine.

[128] http://www.eclipse.org/modeling/emf/ (access date: 11/05/2013)

Each state corresponds to an obtrusiveness level, and the guard conditions of the state transitions reference a user situation. The services' obtrusiveness models are checked by the Reconfiguration Engine (see Figure 6-6) whenever it receives a new user situation; if any transition matches the new situation, it is fired, leading to a new obtrusiveness state for the service. In Figure 6-7, we show the state diagram of a service that displays incoming messages.

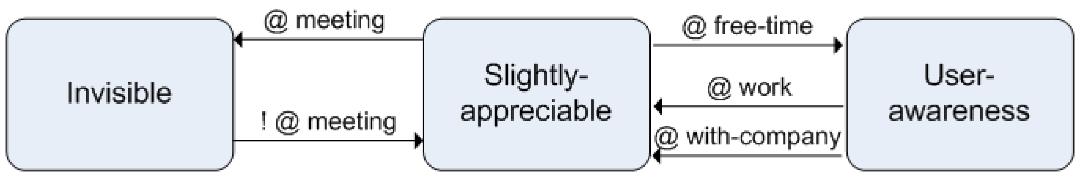


Figure 6-7. *AdaptIO:* Obtrusiveness state diagram for a messaging service.

When the user arrives at work (*@work* situation), the messaging service passes to the *slightly-appreciable* state, thus reducing notification obtrusiveness when the user is working. When the system determines that the user is no longer working (*@free-time* situation), the service goes back to the *user-awareness* state, again increasing notification obtrusiveness. In case the system determines the user is in a meeting (*@meeting*), the service passes to the *invisible* obtrusiveness state, making sure the user is not disturbed. In addition, if the user is in the company of others (*@with-company*) while the service is at the maximum obtrusiveness level (i.e., *user-awareness*), the messaging service transitions to the *slightly-appreciable* level, so the user is not overly disturbed while socializing.

Each of the obtrusiveness states is supported by the appropriate interaction resources. In the second knowledge model, the *interaction model*, the designer associates interaction resources with each obtrusiveness level. For instance, the slightly-appreciable level is associated with the following interaction resources:

- Haptic
    - Vibration
- Visual
    - Property
        - Momentary
    - Lights
    - Text
    - Image

Note that the auditory modality has been deactivated for the slightly-appreciable level, leaving only visual and haptic. Regarding the haptic modality, the vibration resource is enabled, while the lights (e.g., LED flash for Android), text and image visual resources are activated as well. The Momentary property indicates any visual resource will only be shown for a brief moment.

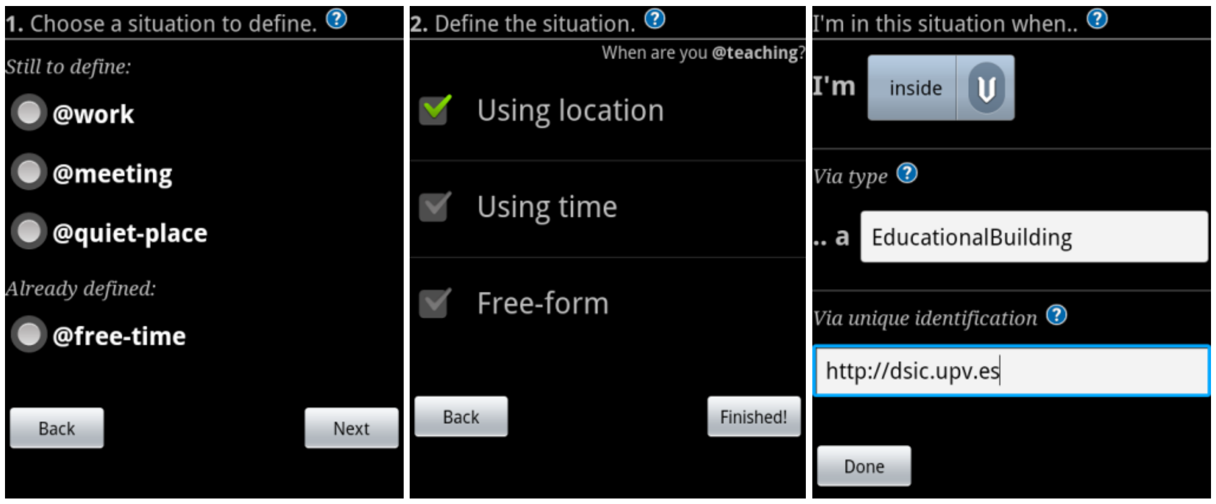
Whenever a transition fires in a service's obtrusiveness model, thus leading to a new obtrusiveness state, the Reconfiguration Engine (see Figure 6-6) consults the service's interaction model, to retrieve interaction resources suiting the new obtrusiveness level.

The knowledge models are re/presented in XML Metadata Interchange standard (XMI)[129]. The obtrusiveness and interaction model for our example messaging service can be found in Appendix C.3.2, as well as a model describing the available interaction resources (haptic, visual, and auditory). Multiple example knowledge models can be found on http://wise.vub.ac.be/william/phd/index.htm#adaptio.

6.3.3.2 *Service User: Situation Specification*

In order to guarantee accurate and unambiguous situation definitions, users are put in charge of defining their own relevant situations. We developed a mobile interface that allows users to perform this task in a generic and fine-grained way, based on environment context. To increase usability and support nomadic users in a wide range of environments, the interface also supports *capturing* user situations. Below, we first discuss how the user can manually specify situations, and then how he can use the "capture" functionality.

Manually specifying situations



EducationalBuilding

http://dsic.upv.es

(a) Situation overview (b) Specification options (c) Define via location

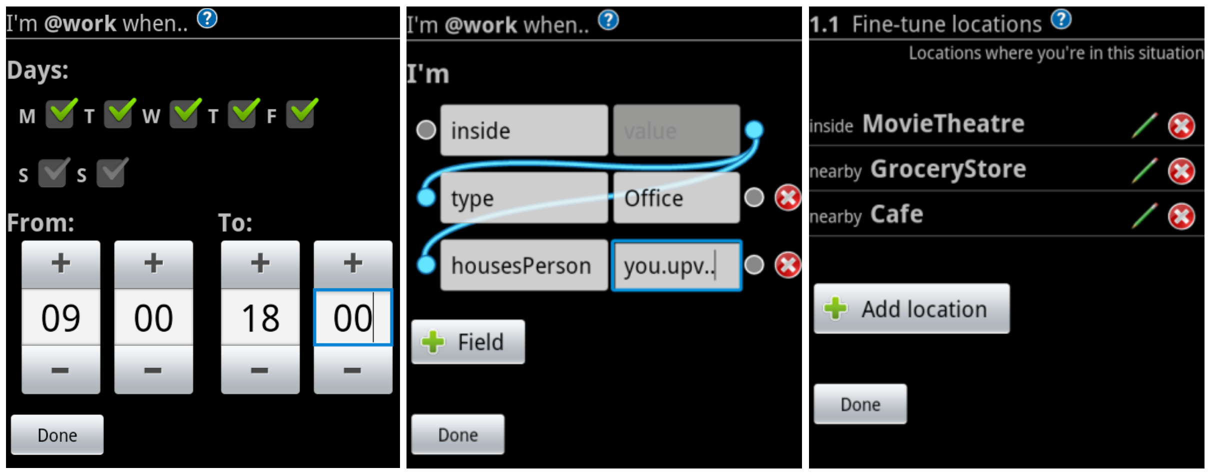
Figure 6-8. *AdaptIO*: Situation Specification Interface screenshots (1).

In the first screen (see Figure 6-8/a), the user chooses to define a still undefined situation (referenced in the deployed services' obtrusiveness models), or edit an already defined one. In the second screen (see Figure 6-1/b), he can define the chosen situation using the location and time options. A third, more advanced "free-form" option allows the user to place arbitrary constraints on his environment (see below). Using the location option, the user describes the location(s) he is in

[129] http://www.omg.org/spec/XMI (access date: 11/05/2013)

while being in the chosen situation. For each location (see Figure 6-8/c), the user specifies whether he is *inside* or *nearby* a certain place, person or thing (i.e. physical entity) in that situation, and provides a way to identify that physical entity via its type and/or unique identification (URI). The user is aided via auto-complete functions: the type field suggests terms from well-known ontologies, as well as synonyms of the ontology terms (provided by WordNet); while the URI field suggests URIs that identify physical entities the user has encountered (this information is obtained from the Environment Discovery and Management Layer). The user can also specify time intervals (i.e., days of the week and time span) during which he is in the situation (see Figure 6-9/a). We note that, in case multiple locations / time spans are specified (e.g., see Figure 6-1/c), the user will be in the particular situation in case he is in one of those locations / current time falls into one of the given time intervals. When multiple options are employed at the same time (e.g., location and time), the user will be in the specified situation only if both defined options are satisfied; for instance, when the user is at (one of) the defined location(s) *and* in (one of) the specified time interval(s).

The advanced, "free-form" option (see Figure 6-9/b) allows defining situations in a more powerful and expressive way, by placing arbitrary constraints on the user's environment context. A constraint consists of a property and a value field. A user may arbitrarily constrain a property value, either by providing a concrete string or by linking its connector to other constraints. This free-form option, with connectable components, resembles the popular Yahoo! Pipes online mashup tool[130]. In this example, the user is inside his office during the *@work* situation. To describe this in a generic way, the user specifies the *inside* property, and creates two constraints on the place he should be inside of. The first constraint states the *type* of the place should be "Office", while the second specifies the place is the user's office (via the *housesPerson* property). Using the constraint's connectors, the user connects the two new constraints to the first constraint's value field. The property and value fields are respectively backed by the same two auto-complete functions mentioned above.



(a) Define via time (b) Free-form option (c) Captured locations

Figure 6-9. *AdaptIO:* Situation Specification Interface screenshots (2).

[130] http://pipes.yahoo.com/pipes/ (access date: 11/05/2013)

Capturing situations

The “capture” option exploits the user’s current environment to quickly and easily specify situations. In this option, the user takes a snapshot of his environment, fine-tunes it, and attaches it to a situation. For example, the user is sitting in a movie theatre, and one of the services produces a loud notification. The *obtrusiveness model*, defining the service’s adaptation behavior (see section 6.3.3.1), makes sure notifications are handled at the *invisible* obtrusiveness level in an *@quiet-place* situation (e.g., classroom). However, mobile users are typically nomadic and move in a wide range of (previously unknown) environments, making it difficult even for them to foresee every situation-relevant environment (e.g., movie theatre). After quickly (and manually) turning off the device’s sound, the user selects the capture option. This option re-uses the screens from the previous “define” option (see previous section) and populates them, based on the user's current environment context. The user selects the location aspect (see Figure 6-9/c), and sees that the “inside MovieTheatre” location is present, as well as some other captured locations (e.g., “nearby Cafe”). He then removes the irrelevant locations, and also unchecks the time and free-form option, since they are not relevant in this case (e.g., see Figure 6-8/b). In the final screen, the user attaches the fine-tuned context data to the *@quiet-place* situation, ensuring the *invisible* obtrusiveness level will be utilized in movie theatres as well.

6.3.4 Evaluation

User acceptance of the general obtrusiveness adaptation mechanism has already been evaluated in previous work [33]. Therefore, this work focuses on evaluating the mobile user’s new role in our extended approach; namely, specifying his own obtrusive situations, utilizing our mobile interface and leveraging environment context. We validated the usability and expressivity of the Situation Specification Interface by means of a user evaluation, where users had to specify six situations of varying complexity via the definition and capturing options (see section 6.3.3.2). The final, most complex situation required using the more advanced free-form option. In each user session, we took five minutes to shortly explain the interface, and then let the user specify the described situations. We noted the required time, as well as any encountered difficulties and errors during their task. After performing their task, the users filled out the Post-Study System Usability Questionnaire (PSSUQ) [127]. A total of 8 subjects participated in the experiment (5 male and 3 female), between the ages of 25 to 35. All of them had a strong background in computer science, being students or researchers; they were also familiar with the use of a smartphone, and 4 out 8 own an Android smartphone similar to the one used in the experiment. For detailed information on the task situations, as well as experimental setup and results, we refer to [30].

Figure 6-10 shows a summary of the PSSUQ questionnaire results. Overall, users found the interface simple to use (questions 1, 2) and very easy to learn (7), while they also felt they could complete tasks effectively (3) and quickly become productive using the system (8). Users found the provided information more or less clear (11), easy to find (12) and understand (13), and clearly organized (15). Overall, around 80% of the users found the interface pleasant (16), and 75% liked using the interface

(17). Averaging the results, on a scale from 1 (strongly agree) to 7 (strongly disagree); overall satisfaction was 3.09, usefulness was 3.2, information quality was 3, and interface quality was 3.04.

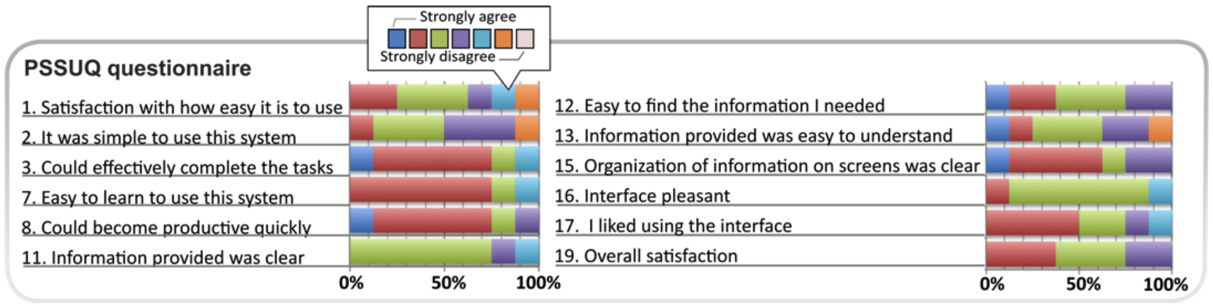


Figure 6-10. *AdaptIO*: Summarized questionnaire results.

On average, users took about 13 minutes to specify all situations. One user had initial difficulty with the location method (see section 6.3.3.2); but the other users had no problems. Six out of seven users had difficulty using the free-form method while testing the defined option; on the other hand, they found using this method easier when “capturing” situations.

In conclusion, the evaluation shows the interface to be usable and expressive, allowing users to specify non-trivial situations within a short time span. Moreover, the capture option makes it very simple to specify most situations. The free-form method, which allows for more generic and complex situation definitions, proved to be more difficult to use and have a steep learning curve. However, using this option becomes much easier during capturing, since users were able to fine-tune a given free-form specification instead of creating one from scratch.

6.4 Summary

This chapter presented three mobile applications, each of which is built on top of SCOUT and relies on its context access features. Each application hereby yields contributions in its respective domain. Below, we shortly summarize each application.

The **Person Matcher** [31] relies on SCOUT’s ability to detect other people in the vicinity of the user. More specifically, the SCOUT push-based query access is utilized, in order to be notified of nearby people and their online FOAF profiles. Based on the user’s and detected persons’ FOAF networks, it performs a detailed compatibility check and notifies the user in case interesting people are detected. This compatibility check involves crawling both FOAF networks, up to a configurable degree, and finding connections between the networks. For instance, an example connection constitutes a friend of the user working at the same research institute as the detected person. The mobile user can specify the kind of people he is interested in via configurable weighting schemes. A mobile user interface enables users to retrieve more information on found matches (e.g., connections between the networks), and allows different weighting schemes to be downloaded and plugged in.

COIN [29] is a mobile, client-side web augmentation system, which automatically injects context-aware and personalized features. Importantly, this system does not require an expensive re-engineering of the website, since the enrichment is performed on-the-fly as the user is visiting existing websites. COIN is based on the increasing availability of semantic information in websites (e.g., via RDFa, microformats), which enables COIN to obtain knowledge on the meaning of existing, third-party page content. In order to enrich websites to suit the mobile user's needs (e.g., annotate products sold by nearby shops), COIN further requires access to rich environment context. The SCOUT framework is utilized to retrieve the required context data in a push- and pull-based way. Furthermore, the non-centralized nature of SCOUT perfectly suits the client-side COIN approach, as the necessary context is directly available locally.

The generic approach to context-aware feature injection consists of three steps. First, semantic information on the website contents is extracted, both from the requested webpage and potentially pages linked by this requested page as well. In the second step, the extracted semantic information is matched to the user's context, with the goal of finding page content that can be enhanced with context-aware features. In the third step, features are injected into the identified content, by applying existing adaptation techniques. The second (matching step) and third (feature injection) steps are repeated for each crawled webpage, and in case of certain context changes.

We implemented COIN as an extensible software framework for the Android platform (version 2.2), where different components can be plugged in at each step. For instance, context-provisioning systems other than SCOUT can be utilized as well, given they provide the necessary context data in a push-based way.

The **AdaptIO** system [30] dynamically adapts the obtrusiveness of mobile interactions to suit the mobile user's situation (e.g., being in a meeting, or at lunch). In mobile and nomadic settings, obtrusiveness adaptation should be applicable in any newly discovered, heterogeneous environment, likely lacking any context-aware infrastructure. We achieve this challenge by putting the user in charge of defining obtrusive situations, and exploiting rich environment context. Moreover, the existing AdaptIO system was moved to the mobile platform, to avoid reliance on a server infrastructure. By utilizing the SCOUT framework, the necessary environment context data can be supplied in a push-based way, across heterogeneous surroundings lacking context-provisioning middleware. In order to enable interaction with environment services as well, SCOUT was extended with standards-based semantic services support.

We further developed a mobile user interface that enables users to define their own obtrusive situations, based on collected environment context. In order to support the nomadic nature of mobile users, the interface also supports capturing situations, whereby the mobile user's current context is used to quickly define new obtrusive situations. Finally, a user study was performed to evaluate the usability and expressivity of the interface.

Chapter 7

Conclusions and future work

In this final chapter, we seize the opportunity to reflect on the results of this dissertation, and look forward to possible future work and new research directions. This chapter is structured as follows. Section 7.1 summarizes the presented work. In section 7.2, we discuss the main contributions and achievements of this dissertation. Section 7.3 lists limitations of the presented work. In section 7.4, articles publishing the work presented in this dissertation are listed.Finally, section 7.5 presents future work.

7.1 Summary

In this dissertation, we presented a mobile, client-side context-provisioning framework called Semantics COntext-aware Ubiquitous scouT (SCOUT). Mobile applications can utilize this software framework to obtain rich knowledge on the user's context and physical surroundings (or environment). In order to supply integrated query access to the collected environment context, SCOUT relies on a general-purpose mobile query service, which was also introduced in this dissertation.

Below, we first summarize the SCOUT framework (section 7.1.1). Then, we shortly review the mobile query service (see section 7.1.2).

7.1.1 SCOUT framework

Each task performed by SCOUT, including detection, spatial information inferring, and data integration, is clearly separated and encapsulated into a distinct layer of the framework architecture.

This way, different technologies and mechanisms can easily be plugged in at each level, without requiring changes to the other layers. At the end of the section, we shortly recap the proof-of-concept applications built on top of SCOUT.

Below, we first summarize the different tasks performed by SCOUT:

Detection: SCOUT automatically detects physical entities in the user's vicinity, and identifies their associated online semantic data sources. For instance, using a built-in RFID reader, tags put on nearby physical entities (e.g., person) can be detected, and their contents read to obtain URLs of associated online data (e.g., FOAF profile).

For this purpose, the SCOUT framework relies on so-called detection techniques. Two kinds of detection techniques are distinguished; *direct*, which employ a hardware component (e.g., RFID reader) to detect entities and locate associated online data; and *indirect*, which perform these tasks by passing on local context (e.g., GPS position) to third-party services (e.g., online geographic directory). By utilizing multiple detection techniques interchangeably and in parallel, SCOUT can be deployed across heterogeneous environments with varying sensing support (e.g., RFID, Bluetooth; or GPS, if no tags or sensors are available).

Inferring spatial information: Based on detection and location data, high-level spatial information is inferred. Such spatial data denotes whether the user is nearby or inside other physical entities (e.g., building, person), and whether physical entities themselves are spatially related (e.g., user's hotel and nearby public transportation hubs).

By supplying such high-level spatial data, mobile applications can abstract from low-level detection and location data, such as detection distances and absolute coordinates. A flexible mechanism is in place, centered on a spatial index, to actively keep spatial information up-to-date. Importantly, this mechanism can deal with detection data of varying accuracy, thereby supporting a wide range of detection techniques (see above). In particular, we apply a custom process to convert inaccurate detection data into spatial shapes for indexing; whereby spatial shapes represent the geometric shapes and absolute coordinates of detected entities.

Data integration: Given the located online semantic data and inferred spatial information, SCOUT provides integrated query access to the collected context and environment data. Mobile applications can pose queries in a push- and pull-based way, to retrieve any piece of information on the user's environment context.

SCOUT keeps and maintains multiple models and components to realize this query access. The *Spatial Model* stores spatial relations between physical entities and the user, which reflect the spatial information passed from the Spatial Layer. The *User Model* stores information useful for personalization (e.g., device properties, user preferences). Finally, the *Environment Model* stands for an abstract, integrated view on the user's current and previous environment, and includes the two aforementioned concrete models together with the identified online semantic data sources. Mobile applications access the Environment Model in a push- and pull-based way, respectively by utilizing the Environment Notification Service and Environment Query Service components.

We discussed three proof-of-concept applications built on top of SCOUT, which rely on the aforementioned context access features to realize their own functionality. Each of these applications yield a contribution in their respective domains. The **Person Matcher** [31] pro-actively identifies nearby people of interest, based on their FOAF networks, and pushes them to the user. In order to be notified of nearby people and their online FOAF profiles, the Person Matcher relies on the push-based query access supplied by SCOUT. **COIN** [29] is a mobile, client-side web augmentation approach, which automatically injects useful context-aware features into websites. In order to determine mobile user needs, COIN utilizes the SCOUT framework to gain push- and pull-based query access to rich environment context. The **AdaptIO** system [30] dynamically adapts the obtrusiveness of mobile interactions to suit the mobile user's current situation. In order to accurately define and determine obtrusive situations (e.g., in a meeting), AdaptIO employs SCOUT to access environment context, across heterogeneous (and potentially minimally outfitted) mobile environments.

7.1.2 Mobile query service

In the SCOUT framework, the Environment Model represents an abstract, integrated view on the user's (current and previous) surroundings. An essential part of this model consists of the online sources detected by SCOUT, describing physical entities in the user's surroundings. In order to transparently query this online semantic dataset, SCOUT relies on the mobile query service presented in this dissertation. This general-purpose query service enables any client to query a currently untapped part of the Semantic Web, comprising of large amounts of relatively small online sources. Clients are hereby able to outline their relevant selection of online semantic data; for instance, data describing nearby detected entities (SCOUT), historic and cultural information (tour guides) [128], or descriptions of items to be recommended (recommender systems) [85]. Below, we discuss the challenges and requirements for the mobile query service, as well as the presented solutions and their implementations. Then, we indicate the results of our experimental validation of the query service.

A number of issues and challenges arise in our particular querying scenario. Firstly, it is not feasible to locally query the entire relevant dataset (e.g., the dataset should be kept in-memory to enable fast querying). Consequently, a first challenge is to **reduce the total query** dataset, to make local querying feasible. Furthermore, our query dataset is located online and captured in individual files. To reduce the resulting high data retrieval costs and decrease the effects of connectivity loss, a second challenge is to **minimize the number of source downloads**. Two solutions naturally present themselves to meet these challenges. By identifying online data relevant to posed queries (**Identify relevant online sources**), the final query dataset is reduced (see first challenge). Furthermore, since only sources comprising query-relevant data need to be downloaded, our second challenge is also met. Secondly, by locally caching data likely to be frequently referenced (**Locally cache data**) fewer sources need to be downloaded to serve queries (see second challenge). To further address our challenges, any software component implementing these solutions should meet two key requirements. Firstly, components should perform *fine-grained data identification and retrieval*, allowing the total query dataset to be further reduced (see first challenge), and more unnecessary downloads to be avoided (see second challenge). Secondly, components should *reduce memory usage and processing effort*, to cope with the fact that mobile hardware is still relatively limited compared to larger devices (e.g., desktop computers, laptops).

The mobile query service implements the aforementioned solutions via the following components: the **Source Index Model (SIM)**, which identifies query-relevant online sources by indexing source metadata (i.e., predicates and resource types); and cache components locally caching source data, including the **Source Cache** and **Meta Cache**, two alternatives that respectively organize the cache based on origin source and shared metadata. By developing and evaluating these two separate cache components, we investigate the impact of source metadata on data selectivity and performance. For the same purpose, we developed three separate SIM variants keeping increased amounts of source metadata. By focusing on source metadata (i.e., predicates and subject/object types), we aim to achieve a balance between fine-grained data identification retrieval and memory / processing overhead (see requirements). We note that, by additionally keeping information on previously removed data (i.e., after clearing storage space), the Meta Cache is also able to perform the source identification task. Therefore, in order to deploy both solutions, the mobile query service either uses 1/ the SIM combined with the Source Cache; or 2/ the stand-alone Meta Cache. We apply a data validity strategy to ensure the freshness of the cached data.

For the Meta Cache, we introduced a removal strategy called Least-Popular-Sources (LPS), which suits our particular setting where data is captured in online sources. We call LPS a source-based removal strategy, since it identifies cached data to be removed via their origin source. This removal strategy takes into account the “popularity” of online sources as indicated by 1/ the popularity of its source data, i.e., the number of stored cache units containing the data, and 2/ the popularity of its own metadata, i.e., how often the contained metadata is found in other sources. By considering the first factor, we reduce the probability of a cache miss; while the second factor helps to keep the number of source re-downloads per cache miss in check. In case it can be assumed that popular metadata combinations are more likely to be referenced by posed queries, this second factor may also help to reduce cache misses. Additionally, download cost can be considered, making it less likely that sources with high download times are removed.

Semantic Web technologies implement the Open World Assumption (OWA), in order to support the vision of the Semantic Web as an open, interlinked web of data. We note that, due to the OWA, any online source may specify extra statements for RDF resources. In the case of the mobile query service, this means newly encountered sources may specify extra types for already processed source data; potentially leading to indexed source metadata to become out-of-date. In order to keep query service metadata up-to-date, resource type information should be tracked across sources, and component data structures need to be updated when necessary. We call this updating process **type**

mediation. On the other hand, we note that in case such cross-source typing issues do not occur in the online dataset, or can be excluded (e.g., in case there is control over online sources, the sources can be supplemented with the missing types), type mediation can be disabled. We further note that other approaches, which integrate semantic data from multiple sources, also encounter this problem but do not consider it in their work [12, 67, 68]. Secondly, by leveraging logical axioms from OWL ontologies, source metadata retrieved from online sources (i.e., types), as well as search constraints extracted from posed queries, can be enriched. This process is called **type inferencing**. In case type inferencing is applied to enrich source metadata, additional query-relevant data can be identified and returned; when search constraints are extended, increased amounts of irrelevant data can be ruled out. The type inferencing feature may be enabled or disabled, depending on the needs of mobile applications and device capabilities (cfr. inferencing support by RDF stores).

We performed an experimental validation of the mobile query service, where the performance of the major query service components was measured, taking different variants and configurations into account. The experiments showed that, by performing the source identification and local caching tasks, the query service is able to supply realistic query performance. Furthermore, we observed that leveraging source metadata indeed allows a balance between fine-grained data identification retrieval and performance overhead. Reflecting this observation, the Meta Cache component turned out to provide the best performance. By additionally applying the LPS removal strategy on Meta Cache, the amount of cache misses and resulting source re-downloads can be kept in check, further increasing querying performance. At the same time, we note that overall execution times are still high for the best performing Meta Cache configuration (i.e., utilizing LPS), ranging from 1,5s to 1,5m. However, most of the responsible overheads, including persistent loading times and source download times, are in the end unavoidable in a setting where memory and storage is limited; while the individual query execution times are bound by the utilized query engine. Finally, the OWA features proved useful in improving data access. However, the experiments showed that type inferencing and type mediation currently exhibit impractical performance, necessitating more advanced components or increased mobile device resources.

7.2 Contributions

In this section, we summarize the contributions resulting from the work in this dissertation. We discuss the contributions related to the SCOUT framework (section 7.2.1) and the mobile query service (section 7.2.2), as well as the individual contributions made by the mobile applications built on top of SCOUT (section 7.2.3).

7.2.1 SCOUT framework

In this section, we discuss the contributions concerning the SCOUT framework.

Mobile, client-side context-provisioning: We presented a mobile, client-side context-provisioning framework called SCOUT (Semantics-based COntext-aware Ubiquitous scouT) that runs entirely on the mobile device. Importantly, SCOUT proves that mobile, client-side context-provisioning – including the context gathering, interpretation, integration and dissemination tasks described below – is feasible on mobile devices.

Support for heterogeneous environments: SCOUT can be deployed in a range of heterogeneous environments, suiting the nomadic nature of mobile users. This is realized by utilizing multiple detection techniques (e.g., utilizing RFID, GPS coordinates) interchangeably and in parallel, to detect online semantic data describing the user's surroundings. This aspect of supporting newly discovered, heterogeneous environments via the deployment of multiple detection techniques is currently not considered in the state of the art.

Semantic Web as information platform: In contrast to most existing context-provisioning approaches, SCOUT does not rely on external context providers or proprietary servers to retrieve (static) context data on the user's environment. Instead, SCOUT leverages the Semantic Web itself as an information platform, in order to retrieve rich semantic data on the user's environment.

Inferring spatial information: We present a flexible mechanism to infer high-level spatial information, based on inaccurate raw detection and location data. Due to this spatial information, mobile applications can easily access location-related data while ignoring low-level details. Furthermore, by allowing detection data of varying accuracy, we support the range of detection techniques utilized to discover the user's environment.

Rich semantic query access: SCOUT supplies integrated, semantic query access to the collected context, in a push- and pull-based way. To our knowledge, there is no current work that presents semantic, push-based query access to integrated context.

7.2.2 Mobile query service

In this section, we elaborate on the contributions pertaining to the mobile query service.

Mobile, client-side integrated querying: We introduced a mobile query service that enables the transparent, integrated querying of large amounts of small online semantic sources. It hereby supplies configurable support for the Semantic Web's Open World Assumption, applying type inferencing and introducing a process called type mediation. The query service is deployed entirely on the mobile device, thus demonstrating the feasibility of performing indexing and caching tasks at the client-side.

Unlocking part of the Semantic Web: Currently, mobile applications need to rely on online query endpoints to access online Semantic Web data; which makes a large part of the Semantic Web, consisting of RDF files and semantically annotated websites, currently inaccessible. By providing integrated query access to such small online sources, the query service unlocks this Semantic Web segment for mobile applications.

Online source identification: We introduce an indexing component called the Source Index Model, which identifies query-relevant data sources by indexing metadata from online sources. We demonstrate that, by focusing on indexing source metadata, high source selectivity can be achieved while still guaranteeing minimal memory and performance overhead. In contrast, related approaches often index instance data (e.g., RDF stores) or resource constraints / selectivity estimates (e.g., query distribution), which takes up more space and requires a larger performance effort.

Local source data caching: We present a caching component called Meta Cache, which locally caches downloaded data and organizes the data according to shared source metadata. We show that by focusing on source metadata, a fine-grained retrieval of cached data can again be realized, without overly compromising memory usage and processing times. To our knowledge, no existing caching system organizes the cached data according to its inherent semantics (i.e., types, predicates).

Tailored removal strategy: We present the Least-Popular-Sources (LPS) removal strategy, fine-tuned to our setting where data is captured in online files. It was demonstrated that this removal strategy significantly increases the querying performance of the Meta Cache component, by reducing cache misses and resulting source re-downloads.

7.2.3 Applications using SCOUT

Other contributions worth highlighting are the ones made by approaches leveraging the SCOUT framework. We shortly summarize these contributions below:

- The **Person Matcher** [31] is a mobile application that automatically finds interesting people in the user's vicinity. We presented a matching algorithm that performs a detailed crawling of FOAF networks, in order to find connections or overlaps. To realize its functionality, the mobile application relies on the pro-active data filtering support provided by SCOUT, in the form of push-based queries.
- **COIN** [29] represents a mobile, client-side web augmentation application, which makes existing websites context-aware by injecting context-aware features on-the-fly. As an important contribution, COIN can be applied to existing websites lacking pre-engineered adaptation support, and without requiring client-side strategies hard-coded to specific website DOMs. This is done by exploiting semantic annotations in websites (e.g., via RDFa) to automatically identify suitable locations for feature injection. In order to obtain pull- and push-based access to rich environment context, and allow the approach to be deployed entirely on the client-side, COIN relies on the SCOUT framework.
- The **AdaptIO** system [30] automatically adapts the obtrusiveness of mobile interactions to suit the mobile user's situation. In contrast to existing approaches, we aimed to provide obtrusiveness adaptation support in a priori unknown, newly discovered environments, suiting mobile users' nomadic nature. For this purpose, we put the mobile user in charge of defining his own situations, and leveraged rich environment context to determine the user's current situation. This contribution was made possible via the SCOUT framework, as it allowed AdaptIO

to obtain rich environment context in heterogeneous mobile environments, and be deployed autonomously on the mobile device.

7.3 Limitations

As any research effort, this dissertation focuses on particular research problems. This requires setting clear boundaries, and accepting the resulting limitations. Below, we summarize these limitations for the work presented in this dissertation, namely the SCOUT framework and mobile query service.

7.3.1 SCOUT framework

Below, we discuss boundaries and limitations related to the SCOUT framework.

Focus on location-related data: The SCOUT context-provisioning framework automatically discovers online semantic sources describing the user's current surroundings. As such, the data discovery process is driven by the user's current location. Our focus on location-related data is further illustrated by the spatial data inferring task (see section 7.1.1), which converts detection and location information into high-level spatial data. The underlying rationale is that mobile users typically require information related to their current surroundings (e.g., which is the next train back home from this station, or where can I have a nice lunch in this neighborhood). Due to this focus, data related to other context factors is not considered at this point, for instance, profiles of people the user is meeting with, as retrieved from his calendar (also called future context) [12].

Dataset with static information: The SCOUT framework aims to supply rich environment context, describing the user's current surroundings. Currently, this context data comprises static data, retrieved from online sources such as RDF files and websites, and does not include dynamic information obtained via e.g., remote sensors. Supporting the efficient retrieval of dynamic data in mobile settings requires special consideration, which was not the goal of this work.

Dataset consisting of small online sources: SCOUT does not rely on centralized information systems, datasets or query endpoints to retrieve descriptions on the user's surroundings. Instead, SCOUT leverages the Semantic Web itself as an information platform, automatically collecting online data sources associated with the user's surroundings. This decision was made to ensure SCOUT is deployable in an autonomous way, independent of any (proprietary) server infrastructures. Furthermore, this allows content providers (e.g., place owners, tourist services) to keep only one information source, such as a semantically annotated website that is both human-readable and consumable by arbitrary software clients. At the same time however, this assumes that physical entities in the user's vicinity (people, places and things) are indeed described by (small) dedicated online semantic sources. Another assumption is that these online sources need to be identifiable, via tags or online services. We however note that, due to the increased deployment of RFID tags and QR codes in daily life, this assumption is in line with current developments.

7.3.2 Mobile query service

In this section, we elaborate on limitations with regards to the mobile query service.

Query dataset consisting of small online sources: The goal of the mobile query service is to supply transparent query access to a currently untapped Semantic Web segment, comprising small online RDF sources (e.g., RDF files, semantically annotated websites). Because of this reason, including large online semantic datasets (e.g., LinkedGeoData, DBPedia) during querying has not been considered.

Other Semantic Web issues: The mobile query service supplies configurable features to support the Semantic Web's Open World Assumption. However, other problems arise as well when exploring data in the Semantic Web. In Linked Data, an important guideline is to re-use the same URI for identical resources, so they can be identified across different online sources. Furthermore, well-known domain-specific ontologies should be utilized to describe RDF resources. However, in practice, content authors usually introduce new URIs to describe existing resources, and create new ontologies to annotate the RDF resources. Therefore, mechanisms should be in place to identify identical RDF resources and align different ontologies describing the same domain. Resolving these issues is beyond the scope of this dissertation. Currently, it is the responsibility of the application developer to resolve these issues manually in the posed queries (e.g., indicate alternative resource URIs and ontology terms in UNION or FILTER clauses).

Limitations of the experimental validation: During the experimental validation, we evaluated the presented Least-Popular-Sources (LPS) removal strategy. As a main goal, these experiments investigated the usefulness of LPS in keeping the number of cache misses and resulting source re-downloads in check for the Meta Cache component. In order to evaluate the performance of LPS, Least-Recently-Used (LRU) was utilized as a reference strategy. In this vein, the goal of the experiments was not to extensively evaluate existing removal strategies, for instance including LRU, Least-Frequently-Used (LFU) or Furthest-Away-Removal (FAR). The suitability of such cache removal strategies depends on the locality of reference, exhibited by the particular experiment queries. For instance, in case a temporal locality is exhibited, LRU and LFU will perform well; in case cached data in the user's vicinity and walking direction is often required, FAR will be more suitable. Therefore, to evaluate such strategies, a large and representative set of queries should be extracted from different types of real-world applications, and used in the experiments. However, comprehensively comparing the LPS strategy to the range of currently existing strategies, utilizing such a large amount of queries from existing mobile applications, is not in the scope of this dissertation.

7.4 Publications

This section lists articles publishing the work presented in this dissertation, respectively concerning the SCOUT context-provisioning framework (section 7.4.1.1), the mobile query service (section 7.4.1.2) and applications using SCOUT (section 7.4.1.3).

7.4.1.1 SCOUT context-provisioning framework

Below, we list publications pertaining to the SCOUT framework.

1) **Publication**: Van Woensel, W., Casteleyn, S., De Troyer, O.: "A Framework for Decentralized, Context-Aware Mobile Applications Using Semantic Web technology", Proceedings of On the Move to Meaningful Internet Systems: OTM 2009 Workshops, Confederated International Workshops and Posters, LNCS 5872, pp. 88-97, Eds. Robert Meersman, Pilar Herrero, Tharam S. Dillon, Publ. Springer Verlag, ISBN 978-3-642-05289-7, Vilamoura, Portugal (2009)

   Summary: This article detailed the layered architecture of the SCOUT context-provisioning framework.

2) **Publication**: Van Woensel, W., Casteleyn, S., De Troyer, O.: "SCOUT: A Framework for Personalized Context-Aware Mobile Applications. ", Proceedings of the ICWE 2009 Doctoral Consortium, Eds. Gustavo Rossi, Publ. CEUR Workshop Proceedings, online http://ceur-ws.org/Vol-484, ISBN 1613-0073, San Sebastian, Spain (2009)

   Summary: This doctoral consortium article presented a short summary of the SCOUT architecture, and discussed the current status and challenges.

7.4.1.2 Mobile query service

In this section, we list publications concerning the mobile query service.

1) **Publication**: Van Woensel, W., Casteleyn, S., Paret, E., De Troyer, O.: "Mobile Querying of Online Semantic Web Data for Context-Aware Applications", IEEE Internet Computing Special Issue (Semantics in Location-Based Services), Vol. 15, N° 6, pp. 32-39, Eds. Sergio Ilarri, Arantza Illarramendi, Eduardo Mena, Amit Sheth, ISBN-ISSN: 1089-7801 (2011)

   Summary: This article presented a newer version of the SCOUT framework architecture, and introduced an initial version of the mobile query service supporting source metadata indexing (featuring type inferencing at the query side).

2) **Publication**: Van Woensel, W., Casteleyn, S., Paret, E., De Troyer, O.: "Transparent Mobile Querying of online RDF sources using Semantic Indexing and Caching", Proceedings of the 12th International Conference on Web Information System Engineering (WISE 2011), pp. 185-198, Eds. Athman Bouguettaya, Manfred Hauswirth, Ling Liu, Sydney, Australia (2011).

   Summary: This article further elaborated on the mobile query service, introducing the Meta Cache and detailing the Source Index Model.

   This paper received the best paper award.

3) **Publication**: Paret, E., Van Woensel, W., Casteleyn, S., Signer, B. and De Troyer, O.: "Efficient Querying of Distributed RDF Sources in Mobile Settings based on a Source Index Model",

Proceedings of the 8th International Conference on Mobile Web Information Systems (MobiWIS 2011), pp. 554-561, Eds. Elhandi Shakshuki, Muhammad Younas, Niagara Falls, Canada (2011)

Summary: This article discussed the Source Index Model of the mobile query service in detail.

This paper received the best paper award.

4) **Publication**: Paret, E., Van Woensel, W., Casteleyn S., Signer, B. and De Troyer, O.: "Efficient Mobile Querying of Distributed RDF Sources", Proceedings of the 8th Extended Semantic Web Conference (ESWC 2011) (Poster), Heraklion Greece (2011)

   Summary: This poster introduced the Source Index Model of the mobile query service.

7.4.1.3 Applications using SCOUT

This section lists publications concerning the proof-of-concept mobile applications built on top of SCOUT.

1) **Publication**: Adapting the obtrusiveness of service interactions in dynamically discovered environments", Proceedings of the 9th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 2012), pp. 250 - 262, Beijing, China (2012)

   Summary: In this article, we discussed how the AdaptIO system was extended to deal with a priori unknown, dynamically discovered environments.

2) **Publication**: Van Woensel, W., Casteleyn, S., De Troyer, O.: "A Generic Approach for On-The-Fly Adding of Context-aware Features to Existing Websites", Proceedings of the 22nd ACM Conference on Hypertext and Hypermedia (HT'11), pp. 143-152, Eds. Paul De Bra, Kaj Grønbæk, ACM 2011, ISBN-ISSN 978-1-4503-0256-2, Eindhoven, Netherlands (2011).

   Summary: This article published an elaborated, generic and conceptual framework for the COIN approach.

3) **Publication**: Casteleyn, S., Van Woensel, W., De Troyer, O.: "Assisting Mobile Web Users: Client-Side Injection of Context-Sensitive Cues into Websites", Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services (iiWAS2010), pp. 441-448, ISBN 978-1-4503-0421-4, Paris, France (2010)

   Summary: In this article, we introduced the initial COIN approach, together with a prototype implementation.

4) **Publication:** Van Woensel, W., Casteleyn, S., De Troyer, O.: "Applying Semantic Web Technology in a Mobile Setting: The Person Matcher", Proceedings of the 10th International Conference of Web Engineering (ICWE 2010) (Demo), pp. 507-510, Eds. Benatallah et al., Publ. LNCS 6189, Springer-Verlag Berlin Heidelberg, ISBN 978-3-642-13910-9, Vienna, Austria (2010)

Summary: This demo article detailed the Person Matcher application.

7.5 Future work

The research presented in this dissertation allows multiple opportunities for future work, some of which are related to the limitations discussed in the previous section. These opportunities range from extensions and improvements to the SCOUT framework and mobile query service, to more elaborate future work, where we extend our work into other research domains. In addition, we mention future work for the mobile approaches utilizing the SCOUT framework.

7.5.1 SCOUT framework

In this section, we discuss future work regarding the mobile SCOUT framework.

In the SCOUT architecture, the **Spatial Layer** is responsible for supplying and maintaining high-level spatial information. The layer is centered on a spatial index, which stores the spatial shapes of detected physical entities and the user. The spatial index is populated based on detection and location information from the Detection Layer, and is utilized to determine nearness and containment between physical entities. Multiple improvements and extensions can be envisioned for this layer, as we discuss below.

Indicate degree of uncertainty: Depending on the detection information supplied by the employed detection techniques, it is possible the locations of detected physical entities need to be approximated. For instance, in case only the detected range is known (e.g., in case of RFID), the detected entity can only be assumed to be located on the edge of a circle, with as radius the detected range. Depending on these approximations, the inferred spatial relations between entities will have a certain level of certainty. By explicitly indicating this degree of certainty, mobile applications may obtain more accurate knowledge on the user's surroundings.

Multiple spatial shapes per entity: Each physical entity has an associated spatial shape in the spatial index. However, some physical entities can include multiple spatial shapes; for instance, an institute spanning multiple buildings.

Location-based garbage-collection of spatial index: It is clear that the spatial index will collect a large amount of entries after a while. To avoid the spatial index getting too large, spatial shapes related to entities currently not in the user's vicinity could be stored persistently or removed.

The SCOUT **Environment Layer** provides mobile applications with an abstract, integrated view of the user and their environment, called the Environment Model. The implementation underlying the Environment Model can be extended in multiple ways, as we discuss below.

Location-based data source retrieval: An essential part of the Environment Model consists of the online semantic dataset describing the user's current environment. Currently, the SCOUT framework utilizes the mobile query service, presented in this dissertation, to gain transparent query access to this online dataset. This general-purpose query service does not make any assumptions on which kind of queries are being fired, and can thus be re-used across different settings (e.g., by recommender systems [85]). However, in a context-aware setting, mobile applications typically require information on nearby physical entities. Consequently, another option is to continuously and pro-actively retrieve detected online sources associated with currently nearby physical entities. These dynamically collected, location-related sources are then utilized as query dataset. Similar pro-active approaches are discussed in [12, 62]. For instance, this location-related dataset can be used in combination with the existing query service; whereby the query service retrieves non location-specific data when referenced.

External (P2P) access to the Environment Model: Currently, applications running on the mobile device can query the Environment Model to obtain knowledge on any piece of the user's current environment. However, external applications (i.e., not deployed on the user's device) could also benefit from the data in this model. For instance, a social networking site can track the route and visited places of a tourist, and share this information with family and friends. Moreover, a particular instance of SCOUT (i.e., running on a certain user's device) could benefit from the knowledge captured in the Environment Models of other users; e.g., to recommend or share certain places or routes through the city, or to obtain data on an environment the user has not yet visited (e.g., other part of the city). In order to support external applications (e.g., online websites) in accessing the Environment Model, the model could for instance be replicated on a WWW server, thus relieving the mobile device from serving all of the information requests. Furthermore, P2P communication between different SCOUT instances could occur opportunistically, for instance to obtain information on the current environment from devices with more sensing capabilities. In either case, it is clear that solid privacy policies need to be put in place. This extension would take place into the domain of P2P communication and security / privacy.

Support efficient services communication: SCOUT already has limited support for interacting with mobile environment services (as described in Chapter 6). Mobile applications can register a service discovery query with the Environment Notification Service, to be alerted when specific services become nearby. Via the Service Invoker component, mobile applications can then interact with discovered services, for instance to enable actuators (e.g., switch on light) or obtain dynamic sensor readings (e.g., light intensity). This means the main communication work is delegated towards the mobile application. Ideally, mobile applications should be able to query the Environment Model to retrieve any piece of information on the user's environment, including dynamic sensor data. Behind the scenes, such queries could be analyzed, enabling SCOUT to contact related services and retrieve the dynamic information on-the-fly. Based on data transiency, some of the data could potentially also be cached. This extension of the SCOUT framework would involve working in the domain of services communication.

Real-world validation of SCOUT: Several mobile systems rely on SCOUT to realize their functionality, each with their own contributions in their respective domains (see section 7.2). While these systems can be seen as proof-of-concepts regarding the supplied functionality of SCOUT, the mobile applications were not tested by actual users in real-world settings (e.g., when walking around). Before SCOUT can be deployed in the real world, it should be validated by evaluating the real-world performance of these applications. This involves setting up extensive user experiments.

7.5.2 Mobile query service

This section discusses future work concerning the mobile query service.

Estimate memory usage at runtime: Currently, we make a rough estimation of the cached payload, since the Android platform does not supply tools to efficiently and accurately measure object memory usage at runtime (such as e.g., the Java Instrumentation API). This estimated size considers the concrete payload, such as character arrays and integers, but not object overheads. More advanced heuristics could be employed to accurately and efficiently measure the entire cached payload. This work would not be specific to the mobile query service, but re-usable by other data-intensive Android applications as well.

Persistent swapping of core indices: As observed in the experiments, encountering a very large amount of sources could lead to the Source Index Model (SIM) or Meta Cache exceeding the available memory. Regarding the cache, this problem could be partially resolved by having a more accurate memory estimate at runtime, so the kept in-memory payload does not exceed the specified memory limit (see above, *Estimate memory usage at runtime*). At any rate however, the internal indices of either component would eventually lead to out-of-memory errors. To resolve this problem, parts of the SIM and Meta Cache indices could be swapped to persistent storage or removed entirely. Determining which parts should be stored or removed could be done via an existing removal strategy, depending on the observed locality of reference. For instance, in case often referenced metadata is likely to be referenced again, LFU can be utilized. It can be noted that our current implementation already contains a multi-level index solution, where parts can be swapped to persistent storage or removed. This implementation is currently used to index resource information during the type mediation process. However, before this solution can be re-used for the core indices of the main query service components, a major effort needs to be undertaken to optimize the implementation (see below, *Efficient multi-level index implementation*).

Efficient multi-level index implementation: Our implementation includes a persistent multi-level index solution, where parts are automatically swapped to persistent storage or removed. Currently, this solution is being utilized to index resource information during the type mediation process. In addition, this multi-level index could be used to improve the memory efficiency of the core indices of the core query service components (see above, *Persistent swapping of core indices*). However, as shown in our experiments, the persistent multi-level index yields a significant performance overhead. Therefore, future work involves optimizing the multi-level index implementation.

Optimize type inferencing: As shown in our experimental validation, type inferencing yields an exceedingly large performance overhead, both during source processing and query resolution. In particular, during the query phase, type inferencing needs to be re-applied to previously processed sources, since online sources cannot be updated with the types already inferred during source processing. In order to avoid this query-time overhead, inferred types can be stored locally after source processing, ruling out the necessity for type inferencing during query resolution (i.e., trading performance for extra storage). However, the fact remains that applying type inferencing in general incurs a huge overhead. In order to make type inferencing feasible in our mobile query service, either a more advanced type inferencing component needs to be devised or plugged in; or more powerful mobile devices need to be employed.

More efficient cache maintenance: Our experimental validation also showed that source-based removal strategies (such as LPS) incur large cache maintenance costs, due to the course-graininess of the cache units. It should be investigated in detail how such maintenance, which involve moving large amounts of cached data to persistent storage or removing persistent data, can be optimized.

Automatic configuration of LPS factors: In our experimental validation, we manually chose a factor weighting for the Least-Popular-Sources (LPS) removal strategy that resulted in the best performance for our specific dataset and experiment queries. However, the optimal factor weighting is likely determined by the particular composition of the online semantic dataset. This is suggested by differences between our current experimental validation and previous experiments [86]. In these previous experiments, a partially synthetic dataset was employed, with fewer metadata combinations that were more evenly spread across the data. As a result, larger numbers of source re-downloads per cache miss were incurred, since the missing metadata combinations typically occurred in many more online sources. In order to deal with different dataset compositions, a mechanism should be developed to infer optimal factor configurations.

Applying query distribution to integrate large datasets: The mobile query service aims to provide transparent, integrated mobile query access to a currently untapped part of the Semantic Web, consisting of small online RDF sources. However, knowledge contained in large online datasets (accessible via online query endpoints, e.g., LinkedGeoData, DBPedia) could also be very useful to mobile applications, for instance to enrich information retrieved from small online RDF sources. To realize a fully integrated query access, mobile applications currently need to query these online datasets separately, and then manually integrate the retrieved data with the query results from our mobile query service. In the query distribution domain, various approaches [67, 68] already exist to efficiently distribute the execution of a query across online query endpoints. An interesting research effort would consist of aligning query distribution with the mobile query service. This work would take place in the domain of query distribution.

Data exploration issues: Multiple issues occur when exploring data across different Semantic Web sources. Firstly, an important Linked Data guideline is to re-use the same URI for identical resources, so they can be identified across different sources. However, RDF content authors usually do not check other sources for existing URIs that already identify the resources in question. This fact is reflected by the sets of interlinks made available between major online datasets (e.g., DBPedia, LinkedGeoData), i.e., by statements specifying equivalence between different resource URIs via the owl:sameAs property. Secondly, well-known, domain-specific ontologies should be utilized to describe RDF resources, in order to facilitate data integration and query authoring. In practice however, semantic sources often use different domain-specific ontologies, or employ their own custom ontology. Furthermore, different schemas and conventions are often utilized to represent the same content data. For instance, the WGS84 [81] and geoFeatures[131] ontologies both describe geographical data, defining different schema terms to represent the same concepts and rely on different conventions to represent geographical coordinates.

The mobile query service already supports data exploration in the Semantic Web to an extent, by supplying configurable features to deal with the Semantic Web's Open World Assumption. However, in order to provide fully robust exploration support, the mobile query service should also deal with the above mentioned issues. For instance, manually specified interlinks could be leveraged to deal with resource identification issues (although these are usually only available for large datasets that are part of the Linked Data cloud[132]), and existing ontology matching and alignment approaches to deal with heterogeneous ontologies. Furthermore, query-rewriting techniques could be applied, whereby queries are relaxed based on user preferences and domain knowledge [129]. For instance, regarding domain knowledge, subclass/subproperty relations as well as thesauri for retrieving word synonyms can be leveraged to replace query terms by more general (or alternative) terms, in order to ensure results will be returned.

Considering other mobility-related issues Matters such as battery consumption and the impact of network delays have not yet been considered in the mobile query service. Future work could consist of dealing with such mobility-related issues.

7.5.3 Applications using SCOUT

In this section, we shortly mention future work regarding the mobile applications utilizing the SCOUT framework.

7.5.3.1 The Person Matcher

Leverage social networks: Currently, the Person Matcher requires each encountered person, as well as the user, to have their own FOAF profile. Future work could involve leveraging social network data on encountered persons (e.g., via the Facebook Graph API) to construct (or extend) the (FOAF) networks of connected people.

[131] http://www.mindswap.org/2004/geo/geoOntologies.shtml (access date: 08/03/2013)
[132] http://linkeddata.org/ (access date: 04/06/2013)

7.5.3.2 COIN

Extra annotation languages: By providing support for other annotation languages (e.g., microdata[133], which is currently pushed by major search providers) in the semantic information extraction step, the applicability of the COIN approach could be increased.

Optimize semantics extraction step: Since the semantics extraction step is quite resource intensive on mobile devices, we aim to investigate methods of optimizing this step. For instance, this could be done by sharing extracted content semantics between clients.

Mobile user interface: Current work-in-progress is the development of a mobile user interface, which allows end-users to easily specify matching strategies and configure adaptation techniques.

Feedback mechanisms: In addition, we are currently developing feedback mechanisms for injected features, which would enable custom matching strategies to fine-tune their process logic according to user feedback.

7.5.3.3 AdaptIO

Fine-tune mobile user interface: The mobile interface should be fine-tuned based on the user feedback. Specifically, this involves improving the usability and learnability of the free-form option.

More extensive user tests: More extensive user tests are still required to fully assess the usability of the interface, preferably with a focus on users without computer science backgrounds.

End-user adaptation behavior specification: The largest remaining challenge is to empower end-users to specify the behavior of interaction adaptation, a task now reserved for the service designer. This way, the user could express custom adaptation behavior not initially foreseen by the designer. For instance, regarding the messaging service example (see Chapter 6, section 6.3.3.1), some people may prefer being made fully aware of any notifications when being in company of others; while others only want to be slightly-aware in case certain people are nearby (e.g., thesis promoter).

[133] http://schema.org/ (access date: 19/07/2013)

Appendix A

SCOUT Ontologies

A.1 Spatial Model ontology

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix owl2xml: <http://www.w3.org/2006/12/owl2-xml#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://wise.vub.ac.be/namespaces/spatial-model#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix spatial-model: <http://wise.vub.ac.be/namespaces/spatial-model#> .
@base <http://wise.vub.ac.be/namespaces/spatial-model> .

<http://wise.vub.ac.be/namespaces/spatial-model> rdf:type owl:Ontology ;
      rdfs:comment "This ontology is used by the Spatial Model in the SCOUT
framework." .


owl:sameAs rdf:type owl:AnnotationProperty .


:contained rdf:type owl:ObjectProperty ;
      rdfs:comment "This property states that the subject resource contained the
object resource in the past." ;
      owl:inverseOf :wasContainedIn .


:containedIn rdf:type owl:ObjectProperty ;
      rdfs:comment "This property states that the subject resource is currently
contained in the object resource." .


:contains rdf:type owl:ObjectProperty ;
```

```
      rdfs:comment "This property states that the subject resource currently
contains the object resource." ;
      owl:inverseOf :containedIn .


:isNearby rdf:type owl:ObjectProperty ;
      rdfs:comment "This property states that the subject resource is currently
nearby the object resource." .


:lastKnownLocation rdf:type owl:ObjectProperty ;
      rdfs:comment "The object resource of this property specifies the last known
location of the subject resource. The absolute coordinates of the location resource
can for instance be specified using the geoFeatures vocabulary (as is done
currently in the SCOUT SpatialManager)." .


:wasContainedIn rdf:type owl:ObjectProperty ;
      rdfs:comment "This property states that the subject resource was contained in
the object resource in the past." ;
      rdfs:subPropertyOf owl:topObjectProperty .


:wasNearby rdf:type owl:ObjectProperty ;
      rdfs:comment "This property states that the subject resource was nearby the
object resource in the past." .


:from rdf:type owl:DatatypeProperty ;
      rdfs:comment "This property states that the spatial relation (e.g.,
indicating nearness, containment) is / was valid from the specified timestamp
onward. The subject of this property should be an rdf:Statement utilizing one of
the spatial relation properties (e.g., isNearby, contains, ..)." .


:until rdf:type owl:DatatypeProperty ;
      rdfs:comment "This property states that the spatial relation (e.g.,
indicating nearness, containment) is / was valid up until the specified timestamp.
The subject of this property should be an rdf:Statement utilizing one of the
spatial relation properties (e.g., isNearby, contains, ..)." .


owl:Thing rdf:type owl:Class .


:currentlyNearby rdf:type owl:NamedIndividual ,
                          owl:Thing ;
      owl:sameAs :isNearby .


:isNearby rdf:type owl:NamedIndividual ,
                   owl:Thing ;
      rdfs:comment "This property states that the subject resource is currently
nearby the object resource." .
```

A.2 Example Spatial Model instance

```
<7b9e9850-4e26-4dfd-a0be-3c97e44fecec>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wilma.vub.ac.be/~wvwoense/metadata/healthcity.rdf> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#wasNearby> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wise.vub.ac.be/members/william> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675767319" ;
      <http://wise.vub.ac.be/namespaces/spatial-model#until>
              "1337675770458" .

<http://wilma.vub.ac.be/~wvwoense/metadata/healthcity.rdf>
      <http://wise.vub.ac.be/namespaces/spatial-model#lastKnownLocation>
              [
<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#hasCoordinat
eSystem>

<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#WGS1984> ;

<http://www.mindswap.org/2003/owl/geo/geoFeatures20040307.owl#xyCoordinates>
                        "50.824501037597656,4.394495487213135
50.82426834106445,4.394777297973633 50.82423400878906,4.394863128662109
50.824302673339844,4.395013332366943 50.82432556152344,4.3949875831604
50.82451248168945,4.395369529724121 50.824771881103516,4.395047664642334
50.824501037597656,4.394495487213135"
              ] .

<28e5a0e6-66b7-4571-b43c-75ff9e357249>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wise.vub.ac.be/members/william> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#contains> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wilma.vub.ac.be/~wvwoense/metadata/kk.rdf> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675770458" .

<http://wilma.vub.ac.be/~wvwoense/metadata/denker_in_alle_staten.rdf>
      <http://wise.vub.ac.be/namespaces/spatial-model#lastKnownLocation>
              [
<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#hasCoordinat
eSystem>
```

```
<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#WGS1984> ;

<http://www.mindswap.org/2003/owl/geo/geoFeatures20040307.owl#xyCoordinates>
                        "50.82242202758789,4.393936634063721"
              ] .

<2b02c20d-6d4b-4f36-8116-59b27b621816>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wise.vub.ac.be/members/william> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#contained> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wilma.vub.ac.be/~wvwoense/metadata/atletiekpiste.rdf> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675765052" ;
      <http://wise.vub.ac.be/namespaces/spatial-model#until>
              "1337675767319" .

<d618aca0-0189-4766-a7d4-a1279ad841bb>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wise.vub.ac.be/members/william> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#wasNearby> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wilma.vub.ac.be/~wvwoense/metadata/denker_in_alle_staten.rdf>
;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675759401" ;
      <http://wise.vub.ac.be/namespaces/spatial-model#until>
              "1337675761151" .

<a85f444f-d0aa-49e4-a030-b8ec5e9992c4>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wise.vub.ac.be/members/william> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#wasNearby> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wilma.vub.ac.be/~wvwoense/metadata/healthcity.rdf> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675767319" ;
      <http://wise.vub.ac.be/namespaces/spatial-model#until>
              "1337675770458" .

<http://wilma.vub.ac.be/~wvwoense/metadata/atletiekpiste.rdf>
      <http://wise.vub.ac.be/namespaces/spatial-model#lastKnownLocation>
              [
<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#hasCoordinat
eSystem>
```

```
<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#WGS1984> ;

<http://www.mindswap.org/2003/owl/geo/geoFeatures20040307.owl#xyCoordinates>
                        "50.822837829589844,4.393952369689941
50.823795318603516,4.3938775062561035 50.82400894165039,4.3944220542907715
50.82379913330078,4.394934177398682 50.822872161865234,4.394920825958252
50.822696685791016,4.394435405731201 50.822837829589844,4.393952369689941"
              ] .

<ade96b4f-1c24-47c2-b68d-7a9193686947>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wise.vub.ac.be/members/william> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#wasNearby> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wilma.vub.ac.be/~wvwoense/metadata/student_restaurant.rdf> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675761151" ;
      <http://wise.vub.ac.be/namespaces/spatial-model#until>
              "1337675765052" .

<http://wise.vub.ac.be/members/william>
      a       <http://wise.vub.ac.be/namespaces/user-model#User> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#lastKnownLocation>
              [
<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#hasCoordinat
eSystem>

<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#WGS1984> ;

<http://www.mindswap.org/2003/owl/geo/geoFeatures20040307.owl#xyCoordinates>
                        "50.82326864795351,4.398212742613295
50.823273144561526,4.398212742613295 50.82327314456066,4.398226978817168
50.82326864795262,4.398226978817168"
              ] .

<6c48ca71-9f71-4b19-aa50-970350411d56>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wilma.vub.ac.be/~wvwoense/metadata/denker_in_alle_staten.rdf>
;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#wasNearby> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wise.vub.ac.be/members/william> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675759401" ;
      <http://wise.vub.ac.be/namespaces/spatial-model#until>
              "1337675761151" .
```

```
<2f83241e-f836-4f80-a459-d8009db460cf>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wilma.vub.ac.be/~wvwoense/metadata/kk.rdf> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#containedIn> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wise.vub.ac.be/members/william> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675770458" .

<f9b4f345-f07e-47ec-a79d-c83662cfe003>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wilma.vub.ac.be/~wvwoense/metadata/tcomplex.rdf> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#isNearby> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wilma.vub.ac.be/~wvwoense/metadata/healthcity.rdf> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675759401" .

<873c08ae-1108-4eaf-991a-29dd103596b6>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wilma.vub.ac.be/~wvwoense/metadata/tcomplex.rdf> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#wasNearby> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wise.vub.ac.be/members/william> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675767319" ;
      <http://wise.vub.ac.be/namespaces/spatial-model#until>
              "1337675770458" .

<04b7d7ac-fbfc-422a-aabe-90bc931b73ca>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wilma.vub.ac.be/~wvwoense/metadata/atletiekpiste.rdf> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#wasContainedIn> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wise.vub.ac.be/members/william> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675765052" ;
      <http://wise.vub.ac.be/namespaces/spatial-model#until>
              "1337675767319" .

<http://wilma.vub.ac.be/~wvwoense/metadata/student_restaurant.rdf>
      <http://wise.vub.ac.be/namespaces/spatial-model#lastKnownLocation>
```

```
              [
<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#hasCoordinat
eSystem>

<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#WGS1984> ;

<http://www.mindswap.org/2003/owl/geo/geoFeatures20040307.owl#xyCoordinates>
                        "50.8216438293457,4.396157264709473
50.82173156738281,4.3962082862854 50.82177734375,4.396084785461426
50.82194137573242,4.396157264709473 50.8220100402832,4.395858287811279
50.821903228759766,4.3957977294921875 50.821956634521484,4.395531177520752
50.821800231933594,4.395445346832275 50.82182312011719,4.395264148712158
50.82171630859375,4.395196914672852 50.821685791015625,4.395368576049805
50.82173156738281,4.395409107208252 50.821624755859375,4.395921230316162
50.82167053222656,4.395970821380615 50.8216438293457,4.396157264709473"
              ] .

<http://wilma.vub.ac.be/~wvwoense/metadata/kk.rdf>
      <http://wise.vub.ac.be/namespaces/spatial-model#lastKnownLocation>
              [
<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#hasCoordinat
eSystem>

<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#WGS1984> ;

<http://www.mindswap.org/2003/owl/geo/geoFeatures20040307.owl#xyCoordinates>
                        "50.8232307434082,4.398026943206787
50.823360443115234,4.39824914932251 50.82332992553711,4.398293495178223
50.82344436645508,4.398494720458984 50.82337188720703,4.3986005783081055
50.82319259643555,4.398293495178223 50.82316970825195,4.398319244384766
50.82310104370117,4.398210525512695 50.8232307434082,4.398026943206787"
              ] .

<c6586351-19b2-4b2a-9d84-153856d19dcc>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wilma.vub.ac.be/~wvwoense/metadata/healthcity.rdf> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#isNearby> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wilma.vub.ac.be/~wvwoense/metadata/tcomplex.rdf> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675759401" .

<http://wilma.vub.ac.be/~wvwoense/metadata/tcomplex.rdf>
      <http://wise.vub.ac.be/namespaces/spatial-model#lastKnownLocation>
              [
<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#hasCoordinat
eSystem>

<http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems20040307.owl#WGS1984> ;
<http://www.mindswap.org/2003/owl/geo/geoFeatures20040307.owl#xyCoordinates>
                        "50.82421112060547,4.395654201507568
50.824249267578125,4.39560079574585 50.82422637939453,4.395543098449707
50.82417297363281,4.395526885986328 50.82417297363281,4.395458698272705
50.82430648803711,4.395485401153564 50.824344635009766,4.395580768585205
50.82439422607422,4.395513534545898 50.824462890625,4.395663738250732
50.824310302734375,4.395848751068115 50.82421112060547,4.395654201507568"
              ] .

<464bb8f4-3abf-464b-8232-b5832802b7d4>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wilma.vub.ac.be/~wvwoense/metadata/student_restaurant.rdf> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#wasNearby> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wise.vub.ac.be/members/william> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675761151" ;
      <http://wise.vub.ac.be/namespaces/spatial-model#until>
              "1337675765052" .

<f55b2060-de64-4d01-9fdd-ff80f692202b>
      a       <http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#object>
              <http://wise.vub.ac.be/members/william> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
              <http://wise.vub.ac.be/namespaces/spatial-model#wasNearby> ;
      <http://www.w3.org/1999/02/22-rdf-syntax-ns#subject>
              <http://wilma.vub.ac.be/~wvwoense/metadata/tcomplex.rdf> ;
      <http://wise.vub.ac.be/namespaces/spatial-model#from>
              "1337675767319" ;
      <http://wise.vub.ac.be/namespaces/spatial-model#until>
              "1337675770458" .
```

A.3 User Model ontology

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix restaurant: <http://gaia.fdi.ucm.es/ontologies/restaurant.owl#> .
@prefix owl2xml: <http://www.w3.org/2006/12/owl2-xml#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://wise.vub.ac.be/namespaces/user-model#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix space: <http://frot.org/space/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://wise.vub.ac.be/namespaces/user-model> .

<http://wise.vub.ac.be/namespaces/user-model> rdf:type owl:Ontology ;
      rdfs:comment "This ontology contains some user model properties, and defines
the User class that is used in the SCOUT user and spatial model. Note that
```

```
properties from well-known ontologies such as FOAF, vCard, .. should be preferred
over these properties, should they be available." .


:prefersCuisine rdf:type owl:ObjectProperty ;
       rdfs:comment "This property states that the user prefers this cuisine." ;
       rdfs:range restaurant:Cuisine ;
       rdfs:domain :User .


:stayingAt rdf:type owl:ObjectProperty ;
       rdfs:comment "This property states that the user is currently staying at this
hotel." ;
       rdfs:range space:Hotel ;
       rdfs:domain :User .


:allowance rdf:type owl:DatatypeProperty ;
       rdfs:comment "The amount of money the user is willing or able to pay." ;
       rdfs:domain :User .


space:Hotel rdf:type owl:Class ;
       rdfs:comment "A hotel." .


restaurant:Cuisine rdf:type owl:Class ;
       rdfs:comment "A cuisine (e.g., Italian, Chinese, ..)" .


:User rdf:type owl:Class ;
       rdfs:comment "A resource of this type represents the user about whom the user
model stores information. In SCOUT, this type is required to denote the user
resource in the user and spatial model." .
```

A.4 Example User Model instance

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix project:  <http://ebiquity.umbc.edu/ontology/project.owl#> .
@prefix person:  <http://ebiquity.umbc.edu/ontology/person.owl#> .
@prefix milo:    <http://reliant.teknowledge.com/DAML/Mid-level-ontology.owl#> .
@prefix foaf:    <http://xmlns.com/foaf/0.1/> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix um:      <http://wise.vub.ac.be/namespaces/user-model#> .
@prefix resto:   <http://gaia.fdi.ucm.es/ontologies/restaurant.owl#> .
@prefix space:   <http://frot.org/space/0.1/> .
@prefix research:  <http://ebiquity.umbc.edu/ontology/research.owl#> .
@prefix publication:  <http://ebiquity.umbc.edu/ontology/publication.owl#> .

<http://wise.vub.ac.be/members/william/foaf_william.rdf>
      a       foaf:PersonalProfileDocument ;
      foaf:maker <http://wise.vub.ac.be/members/william> ;
      foaf:primaryTopic <http://wise.vub.ac.be/members/william> .

<http://wise.vub.ac.be/members/william>
      a       foaf:Person , um:User ;
      um:prefersCuisine resto:MexicanTexMexCuisine ;
      um:stayingAt <http://www.queeneindhoven.nl/> ;
      foaf:family_name "Van Woensel" ;
      foaf:givenname "William" ;
      foaf:homepage <http://wise.vub.ac.be/members/william> ;
      foaf:interest <http://en.wikipedia.org/wiki/Mobile_computing> ,
<http://semanticweb.org> , milo:Guitar ;
      foaf:knows <http://wise.vub.ac.be/members/bram> ,
<http://wise.vub.ac.be/members/beat> , <http://wise.vub.ac.be/members/sven> ,
<http://wise.vub.ac.be/members/olga> ;
      foaf:mbox_sha1sum "5a88cf44b22fec69f5ce52745a5f10777fb70373" ;
      foaf:name "William Van Woensel" ;
      foaf:nick "darth_willy" ;
      foaf:schoolHomepage <http://www.vub.ac.be> ;
      foaf:title "Mr" ;
      foaf:workInfoHomepage
              <http://wise.vub.ac.be/members/william> ;
      foaf:workplaceHomepage
              <http://wise.vub.ac.be> .

resto:MexicanTexMexCuisine
      a       resto:Cuisine .

<http://www.queeneindhoven.nl/>
      a       space:Hotel .

<http://semanticweb.org>
      a       foaf:Document ;
      foaf:topic <http://www.w3.org/RDF/> , <http://www.w3.org/TR/rdf-sparql-
query/> , <http://www.w3.org/TR/owl-features/> , research:SemanticWeb ,
<http://www.w3.org/2001/sw/> .

<http://wise.vub.ac.be/members/sven>
      a       foaf:Person ;
      rdfs:seeAlso <http://wise.vub.ac.be/members/william/foaf_sven.rdf> .

<http://wise.vub.ac.be/members/olga>
      a       foaf:Person ;
      rdfs:seeAlso <http://wise.vub.ac.be/members/william/foaf_olga.rdf> .

<http://en.wikipedia.org/wiki/Mobile_computing>
      a       foaf:Document ;
      foaf:topic research:MobileComputing .

<http://wise.vub.ac.be/members/bram>
      a       foaf:Person ;
      rdfs:seeAlso <http://wise.vub.ac.be/members/william/foaf_bram.rdf> .

<http://wise.vub.ac.be/members/beat>
      a       foaf:Person .
```

Appendix B

Mobile applications

B.1 Experimental validation

B.1.1 Queries

The first query returns all shopping centres together with their names, absolute coordinates, town and (optionally) photos:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wgs: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX fb: <http://rdf.freebase.com/ns/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?shoppingCentre ?name ?lat ?long ?town ?photos
WHERE {
  ?shoppingCentre rdf:type fb:location.location .
  ?shoppingCentre rdfs:label ?name .

  ?shoppingCentre wgs:lat ?lat ;
    wgs:long ?long ;
    fb:business.shopping_center.address ?address .
  ?address fb:location.mailing_address.citytown ?town .

  OPTIONAL {
    ?shoppingCentre dbp:hasPhotoCollection ?photos .
  }
}
```

The second query selects all persons and the groups they are member of, optionally with images depicting these persons and their online chat accounts:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?person ?group ?img ?account
WHERE {
  ?group rdf:type foaf:Group .
  ?group foaf:member ?person .
  ?person rdf:type foaf:Person .

  OPTIONAL {
    ?img foaf:depicts ?person .
  }
  OPTIONAL {
    ?person foaf:holdsAccount ?account .
    ?account rdf:type foaf:OnlineChatAccount .
  }
}
```

The third query returns all airports, together with their absolute coordinates:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.geonames.org/ontology#>
PREFIX sch: <http://schema.org/>
PREFIX wgs: <http://www.w3.org/2003/01/geo/wgs84_pos#>
SELECT DISTINCT ?airport ?lat ?long
WHERE {
  ?airport rdf:type sch:Airport .

  ?airport wgs:geometry ?shape ;
    wgs:lat ?lat ;
    wgs:long ?long .
}
```

The fourth query finds all exhibitions, together with their names, start- and end-dates, venues, displayed pieces and their names:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX am: <http://purl.org/collections/nl/am/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?exhibition ?name ?from ?to ?venue ?piece ?pieceName
WHERE {
  ?exhibition rdf:type am:Exhibition .
  ?exhibition rdfs:label ?name ;
    am:exhibitionDateStart ?from ;
    am:exhibitionDateEnd ?to ;
    am:exhibitionVenue ?venue .

  ?piece am:exhibition ?exhibition ;
    am:title ?pieceName .
}
```

The fifth and final query selects products below 20 dollars, their price, manufacturer, name and comments:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX gr: <http://purl.org/goodrelations/v1#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?offering ?price ?manufacturer ?name ?comment
WHERE {
  ?offering gr:hasPriceSpecification ?priceSpec ;
    gr:includesObject ?object .

  ?priceSpec gr:hasCurrency "USD" ;
    gr:hasCurrencyValue ?price .
  FILTER (xsd:double(?price) < 20)

  ?object gr:typeOfGood ?goodType .
  ?goodType gr:hasMakeAndModel ?model .
  ?model rdf:type gr:ProductOrServiceModel .

  ?model gr:hasManufacturer ?manufacturer .

  ?model rdfs:label ?name ;
    rdfs:comment ?comment .
};
```

Appendix C

Mobile applications

C.1 Person Matcher

C.1.1 Weighting schemes

C.1.1.1 Friends scheme

```
@prefix match: <http://wise.vub.ac.be/match#> .
@prefix match_friend: <http://wise.vub.ac.be/match_friend#> .
@prefix dcmi: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema-datatypes#> .

match_friend:FriendMatcher rdf:type match:WeightingScheme ;
	rdfs:label "Friend Matcher" ;
	dcmi:description "A weighting scheme for finding new friends" ;

	match:hasMatchRelation [
		a match:LinkMatchRelation ;

		rdfs:label "knows" ;
		dcmi:description "denotes rdf:type generic 'knows' relation" ;

		match:weight "1.0"^^xsd:float ;

		match:hasMatchProperty [
			match:propID "knows" ;
			match:property foaf:knows ;
			match:isSymmetric "true"^^xsd:boolean
		]
```

```
        ] ;
        match:hasMatchRelation [
              a match:LinkMatchRelation ;

              rdfs:label "in group together with" ;
              dcmi:description "match people who are in the same groups together" ;

              match:weight "0.8"^^xsd:float ;

              match:hasMatchPropertySequence [
                    match:hasMatchProperties [
                          a rdf:Seq ;
                          rdf:li match_friend:member1 ;
                          rdf:li match_friend:member2
                    ]
              ]
        ] ;
        match:hasMatchRelation [
              a match:ValueMatchRelation ;
              match:valueArity match:arityMany ;

              rdfs:label "share interest topics with" ;
              dcmi:description "match via topics of explicit user interests" ;

              match:weight "0.9" ;

              match:hasMatchPropertySequence [
                    match:hasMatchProperties [
                          a rdf:Seq ;
                          rdf:li match_friend:interest ;
                          rdf:li match_friend:topic
                    ]
              ]
        ] .


match_friend:interest rdf:type match:MatchProperty ;
        match:propID "interest" ;
        match:property foaf:interest ;
        match:isSymmetric "false" .

match_friend:topic rdf:type match:MatchProperty ;
        match:propID "topic" ;
        match:property foaf:topic ;
        match:isSymmetric "false" .

match_friend:made1 rdf:type match:MatchProperty ;
        match:propID "made_thing" ;
        match:property foaf:made ;
        match:isSymmetric "false" ;
        match:inverseProperty foaf:maker .
```

```
match_friend:made2 rdf:type match:MatchProperty ;
	match:propID "made_thing" ;
	match:property foaf:maker ;
	match:isSymmetric "false" ;
	match:inverseProperty foaf:made .

match_friend:member1 rdf:type match:MatchProperty ;
	match:propID "member" ;
	match:property foaf:member ;
	match:isSymmetric "false" ;

match_friend:member2 rdf:type match:MatchProperty ;
	match:propID "member" ;
	match:property foaf:member ;
	match:isSymmetric "false" ;
```

C.1.1.2 Colleagues scheme

```
@prefix match: <http://wise.vub.ac.be/match#> .
@prefix match_friend: <http://wise.vub.ac.be/match_friend#> .
@prefix dcmi: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema-datatypes#> .

match_friend:ColleagueMatcher rdf:type match:WeightingScheme ;
	rdfs:label "Colleague Matcher" ;
	dcmi:description "A weighting scheme for finding new people to collaborate
with" ;

	match:hasMatchRelation [
		a match:LinkMatchRelation ;

		rdfs:label "knows" ;
		dcmi:description "denotes rdf:type generic 'knows' relation" ;

		match:weight "1.0"^^xsd:float ;

		match:hasMatchProperty [
			match:propID "knows" ;
			match:property foaf:knows ;
			match:isSymmetric "true"^^xsd:boolean
		]
	] ;
	match:hasMatchRelation [
		a match:LinkMatchRelation ;

		rdfs:label "made document with" ;
		dcmi:description "match people who have made documents together" ;
```

```
            match:weight "0.8"^^xsd:float ;

            match:hasMatchPropertySequence [
                  match:hasMatchProperties [
                        a rdf:Seq ;
                        rdf:li match_friend:made1 ;
                        rdf:li match_friend:made2
                  ]
            ]
      ] ;
      match:hasMatchRelation [
            a match:LinkMatchRelation ;

            rdfs:label "in group together with" ;
            dcmi:description "match people who are in the same groups together" ;

            match:weight "0.7"^^xsd:float ;

            match:hasMatchPropertySequence [
                  match:hasMatchProperties [
                        a rdf:Seq ;
                        rdf:li match_friend:member1 ;
                        rdf:li match_friend:member2
                  ]
            ]
      ] ;
      match:hasMatchRelation [
            a match:ValueMatchRelation ;
            match:valueArity match:arityMany ;

            rdfs:label "share interest topics with" ;
            dcmi:description "match via topics of explicit user interests" ;

            match:weight "0.7" ;

            match:hasMatchPropertySequence [
                  match:hasMatchProperties [
                        a rdf:Seq ;
                        rdf:li match_friend:interest ;
                        rdf:li match_friend:topic
                  ]
            ]
      ] .


match_friend:interest rdf:type match:MatchProperty ;
      match:propID "interest" ;
      match:property foaf:interest ;
      match:isSymmetric "false" .

match_friend:topic rdf:type match:MatchProperty ;
      match:propID "topic" ;
      match:property foaf:topic ;
      match:isSymmetric "false" .

match_friend:made1 rdf:type match:MatchProperty ;
      match:propID "made_thing" ;
      match:property foaf:made ;
      match:isSymmetric "false" ;
      match:inverseProperty foaf:maker .

match_friend:made2 rdf:type match:MatchProperty ;
      match:propID "made_thing" ;
      match:property foaf:maker ;
      match:isSymmetric "false" ;
      match:inverseProperty foaf:made .

match_friend:member1 rdf:type match:MatchProperty ;
      match:propID "member" ;
      match:property foaf:member ;
      match:isSymmetric "false" ;

match_friend:member2 rdf:type match:MatchProperty ;
      match:propID "member" ;
      match:property foaf:member ;
      match:isSymmetric "false" ;
```

C.2 COIN

C.2.1 Queries

C.2.1.1 Page Model query

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX gr: <http://purl.org/goodrelations/v1#>
PREFIX pm: <http://wise.vub.ac.be/namespaces/webscape/page_model#>
SELECT ?page_product_id ?page_manufacturer ?page_element
WHERE {
  ?triple1 rdf:subject ?page_product ;
     rdf:predicate gr:hasMPN ;
     rdf:object ?page_product_id .

  ?triple2 rdf:subject ?page_product ;
     rdf:predicate gr:hasManufacturer ;
     rdf:object ?page_manufacturer .

  ?triple1 pm:relatedElement ?page_element .
}
```

C.2.1.2 Context query

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX um: <http://wise.vub.ac.be/namespaces/user-model#>
PREFIX geo: <http://www.mindswap.org/2003/owl/geo/geoFeatures20040307.owl#>
PREFIX region: <http://wise.vub.ac.be/region/>
PREFIX gr: <http://purl.org/goodrelations/v1#>
PREFIX sumo: <http://www.ontologyportal.org/SUMO.owl#>
PREFIX sm: <http://wise.vub.ac.be/namespaces/spatial-model#>
SELECT ?interest ?user_coords ?entity_coords ?shop_product_id
WHERE {
  ?user rdf:type um:User ;
     um:manufacturerInterest ?interest ;
     sm:lastKnownLocation ?user_pos .
  ?user_pos geo:xyCoordinates ?user_coords .

  ?stat rdf:subject ?user ;
     rdf:predicate sm:isNearby ;
     rdf:object ?entity .

  ?entity rdf:type sumo:RetailShop ;
     region:sells ?shop_product ;
     sm:lastKnownLocation ?entity_pos .
  ?entity_pos geo:xyCoordinates ?entity_coords .
  ?shop_product gr:hasMPN ?shop_product_id .
}
```

C.3 AdaptIO

C.3.1 Service discovery scenarios

C.3.1.1 Tourism application

A local tourism application enables remote tourism services to provide the user with information on good nearby hotel deals, and nearby points-of-interest. To discover these kinds of remote services, the application registers a discovery query with the Environment Notification Service, from the Environment Discovery and Management Layer.

This query looks for nearby tourism services (msm:Service, es:TourismService), which provides an operation (msm:hasOperation) to get points-of-interest (es:GetPointsOfInterest) or hotel deals (es:getHotelDeals), and is currently nearby (sm:isNearby) the user (um:User). It further obtains the contact information for the service and its operations.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX msm: <http://cms-wg.sti2.org/ns/minimal-service-model#>
PREFIX es: <http://wise.vub.ac.be/namespaces/environment-services#>
PREFIX um: <http://wise.vub.ac.be/namespaces/user-model#>
PREFIX sm: <http://wise.vub.ac.be/namespaces/spatial-model#>
SELECT ?service ?operation ?operationType ?port ?targetNS ?address ?transport
WHERE {
  ?service a msm:Service ;
    a es:TourismService ;
    msm:hasOperation ?operation .
  ?operation a ?operationType .
  FILTER (sameTerm(?operationType, es:GetPointsOfInterest) ||
sameTerm(?operationType, es:getHotelDeals))

  ?user a um:User .
  ?stat rdf:subject ?user ;
    rdf:predicate sm:isNearby ;
    rdf:object ?service .

  // contact info
  ?service msm:hasPort ?port ;
    msm:hasTargetNamespace ?targetNS .

  ?port msm:hasBindType ;
    msm:hasLocation ?address ;
    msm:hasTransport ?transport .
}
```

In case a relevant tourism service is encountered, the application is notified, which utilizes the Service Invoker for remote communication. Based on the received data from one of the operations, the application provides notifications, informing the user of good nearby deals and points-of-interest.

C.3.1.2 Movie application

A local movie application enables remote movie ticket services to notify the user, in case the movie theatre plays a movie on the user's "to-watch" list. To discover these kinds of services, the application registers a discovery query with the Environment Notification Service, from the Environment Discovery and Management layer.

This query looks for movie ticket services (msm:Service, es:MovieTicketService), which provides an operation (msm:hasOperation) to obtain information on available movie tickets (es:getMovieTicketInformation), and is located inside (sm:containedIn) a movie theatre (sumo:MovieTheatre) that is currently nearby (sm:isNearby) the user (um:User).

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX msm: <http://cms-wg.sti2.org/ns/minimal-service-model#>
PREFIX es: <http://wise.vub.ac.be/namespaces/environment-services#>
PREFIX sm: <http://wise.vub.ac.be/namespaces/spatial-model#>
PREFIX um: <http://wise.vub.ac.be/namespaces/user-model#>
SELECT ?service ?operation ?port ?targetNS ?address ?transport
WHERE {
  ?service a msm:Service ;
    a es:MovieTicketService ;
    msm:hasOperation ?operation .
  ?operation a es:GetMovieTicketInformation .

  ?stat rdf:subject ?service ;
    rdf:predicate sm:containedIn ;
    rdf:object ?movieTheatre .
```

```
        ?movieTheatre a sumo:MovieTheatre .

        ?user a um:User .
        ?stat rdf:subject ?user ;
          rdf:predicate sm:isNearby ;
          rdf:object ?movieTheatre .

        // contact info
        ?service msm:hasPort ?port ;
          msm:hasTargetNamespace ?targetNS .

        ?port msm:hasBindType ;
          msm:hasLocation ?address ;
          msm:hasTransport ?transport .
      }
```

In case a relevant ticket service is encountered, the application is notified. The application utilizes the Service Invoker to communicate remotely with the service. Based on the response received from the es:GetMovieTicketInformation operation, the application determines whether the service sells tickets for movies "to watch". If so, the application notifies the user, informing him of the movie and the movie theatre, and potentially passing on the service's link to the webpage where the movie ticket can be purchased.

C.3.2 Knowledge models

C.3.2.1 Obtrusiveness model

```
<?xml version="1.0" encoding="UTF-8"?>
<obtrusivenessmodel:ConsiderateSystem xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:obtrusivenessmodel="http://pros.upv.es.ObtrusivenessSpace/" id="">
  <containerOfObtrusivenessSpaces>
    <obtrusivenessSpace name="">
      <initiativeLevels name="Proactive"
obtrusivenessLevels="//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace/@atten
tionLevels.0/@obtrusivenessLevels.0
//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace/@attentionLevels.1/@obtrusi
venessLevels.0
//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace/@attentionLevels.2/@obtrusi
venessLevels.0"/>
      <attentionLevels name="Invisible">
        <obtrusivenessLevels name="Invisible-Proactive"
initiativeLevel="//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace/@initiativ
eLevels.0">
          <obtrusivenessLevelServices name="messaging-insivible"
obtrusivenessServiceService="//@services.0">
            <transitions name="!@meeting"
targetService="//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace/@attentionLe
vels.1/@obtrusivenessLevels.0/@obtrusivenessLevelServices.0"/>
```

```
            <configurationModel href="InteractionFeatures-
InvisibleProactive.cm4spl#/"/>
          </obtrusivenessLevelServices>
        </obtrusivenessLevels>
      </attentionLevels>
      <attentionLevels name="Slightly-appreciable">
        <obtrusivenessLevels name="Slightly-Proactive"
initiativeLevel="//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace/@initiativ
eLevels.0">
          <obtrusivenessLevelServices name="messaging-slightly"
obtrusivenessServiceService="//@services.0">
            <transitions name="@meeting"
targetService="//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace/@attentionLe
vels.0/@obtrusivenessLevels.0/@obtrusivenessLevelServices.0"/>
            <transitions name="@free-time"
targetService="//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace/@attentionLe
vels.2/@obtrusivenessLevels.0/@obtrusivenessLevelServices.0"/>
            <configurationModel href="InteractionFeatures-
SlightlyProactiveDefault.cm4spl#/"/>
          </obtrusivenessLevelServices>
        </obtrusivenessLevels>
      </attentionLevels>
      <attentionLevels name="User-awareness">
        <obtrusivenessLevels name="Aware-Proactive"
initiativeLevel="//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace/@initiativ
eLevels.0">
          <obtrusivenessLevelServices name="messaging-aware"
obtrusivenessServiceService="//@services.0">
            <transitions name="@work"
targetService="//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace/@attentionLe
vels.1/@obtrusivenessLevels.0/@obtrusivenessLevelServices.0"/>
            <transitions name="@with-company"
targetService="//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace/@attentionLe
vels.1/@obtrusivenessLevels.0/@obtrusivenessLevelServices.0"/>
            <configurationModel href="InteractionFeatures-
AwareProactive.cm4spl#/"/>
          </obtrusivenessLevelServices>
        </obtrusivenessLevels>
      </attentionLevels>
    </obtrusivenessSpace>
  </containerOfObtrusivenessSpaces>
  <services name="Messaging"
obtrusivenessSpacesService="//@containerOfObtrusivenessSpaces.0/@obtrusivenessSpace
"/>
  <featureModel href="InteractionFeatures.fm4spl#/"/>
</obtrusivenessmodel:ConsiderateSystem>
```

C.3.2.2 Interaction model – invisible

```
<?xml version="1.0" encoding="UTF-8"?>
<ConfigurationModelPackage:ConfigurationModel xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
```

```
xmlns:ConfigurationModelPackage="http://es.gvcase.ConfigurationModelPackage/"
name="InvisibleProactive_Configuration">
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.0"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.2"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.3"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.4"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.5"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.6"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.7"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.8"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.13"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.10"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.1"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.9"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.14"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.15"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.16"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.11"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.12"/>
  </FeatureStates>
  <featureModel
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#/"/>
</ConfigurationModelPackage:ConfigurationModel>
```

C.3.2.3 Interaction model – slightly-appreciable

```
<?xml version="1.0" encoding="UTF-8"?>
<ConfigurationModelPackage:ConfigurationModel xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:ConfigurationModelPackage="http://es.gvcase.ConfigurationModelPackage/"
name="SlightlyProactiveDefault_Configuration">
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.0"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.2"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.3"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.4"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.5"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.6"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.7"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.8"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.13"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.10"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.1"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
```

```
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.9"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.14"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.15"/>
  </FeatureStates>
  <FeatureStates state="DEACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.16"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.11"/>
  </FeatureStates>
  <FeatureStates state="ACTIVE">
    <Feature
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#//@Features.12"/>
  </FeatureStates>
  <featureModel
href="file:/Users/migipas/Desktop/eclipseModeling/Eclipse.app/Contents/MacOS/runtim
eObtrusiveness/CaseStudyModels/InteractionFeatures.fm4spl#/"/>
</ConfigurationModelPackage:ConfigurationModel>
```

C.3.2.4 Interaction resources model

```
<?xml version="1.0" encoding="UTF-8"?>
<FeatureModelPackage:FeatureModel xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:FeatureModelPackage="http://es.gvcase.featuremodelpackage"
Name="InteractionFeatures">
  <Features Name="InteractionModalities">
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Optional"
UpperBound="1" To="//@Features.3" From="//@Features.0"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Optional"
UpperBound="1" To="//@Features.2" From="//@Features.0"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Optional"
UpperBound="1" To="//@Features.1" From="//@Features.0"/>
  </Features>
  <Features Name="Visual">
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Optional"
UpperBound="1" To="//@Features.9" From="//@Features.1"/>
```

```
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Mandatory"
LowerBound="1" UpperBound="1" To="//@Features.10" From="//@Features.1"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:OR"
LowerBound="1" UpperBound="-1"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Regular"
UpperBound="1" To="//@Features.12"
From="//@Features.1/@CardinalityBased_Relationships.2"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Regular"
UpperBound="1" To="//@Features.11"
From="//@Features.1/@CardinalityBased_Relationships.2"/>
  </Features>
  <Features Name="Auditory">
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Alternative"
LowerBound="1" UpperBound="1"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Regular"
UpperBound="1" To="//@Features.5"
From="//@Features.2/@CardinalityBased_Relationships.0"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Regular"
UpperBound="1" To="//@Features.6"
From="//@Features.2/@CardinalityBased_Relationships.0"/>
  </Features>
  <Features Name="Haptic">
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Mandatory"
LowerBound="1" UpperBound="1" To="//@Features.4" From="//@Features.3"/>
  </Features>
  <Features Name="Vibration"/>
  <Features Name="Sound">
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Alternative"
LowerBound="1" UpperBound="1"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Regular"
UpperBound="1" To="//@Features.7"
From="//@Features.5/@CardinalityBased_Relationships.0"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Regular"
UpperBound="1" To="//@Features.8"
From="//@Features.5/@CardinalityBased_Relationships.0"/>
  </Features>
  <Features Name="Speech"/>
  <Features Name="Soft"/>
  <Features Name="Loud"/>
  <Features Name="Lights"/>
  <Features Name="Property">
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Alternative"
LowerBound="1" UpperBound="1"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Regular"
UpperBound="1" To="//@Features.16"
From="//@Features.10/@CardinalityBased_Relationships.0"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Regular"
UpperBound="1" To="//@Features.15"
From="//@Features.10/@CardinalityBased_Relationships.0"/>
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Regular"
UpperBound="1" To="//@Features.14"
From="//@Features.10/@CardinalityBased_Relationships.0"/>
```

```
    <CardinalityBased_Relationships xsi:type="FeatureModelPackage:Regular"
UpperBound="1" To="//@Features.13"
From="//@Features.10/@CardinalityBased_Relationships.0"/>
  </Features>
  <Features Name="Text"/>
  <Features Name="Image"/>
  <Features Name="QuickView"/>
  <Features Name="Highlighted"/>
  <Features Name="Momentary"/>
  <Features Name="Iconic"/>
</FeatureModelPackage:FeatureModel>
```

Bibliography

1. Ruiz, J., Li, Y., Lank, E.: User-defined motion gestures for mobile interaction. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 197–206. ACM, New York, NY, USA (2011).

2. Scholliers, C., Hoste, L., Signer, B., De Meuter, W.: Midas: a declarative multi-touch interaction framework. Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction. pp. 49–56. ACM, New York, NY, USA (2011).

3. Chiti, S., Leporini, B.: Accessibility of android-based mobile devices: a prototype to investigate interaction with blind users. Proceedings of the 13th international conference on Computers Helping People with Special Needs - Volume Part II. pp. 607–614. Springer-Verlag, Berlin, Heidelberg (2012).

4. Kane, S.K., Jayant, C., Wobbrock, J.O., Ladner, R.E.: Freedom to roam: a study of mobile device adoption and accessibility for people with visual and motor disabilities. Proceedings of the 11th international ACM SIGACCESS conference on Computers and accessibility. pp. 115–122. ACM, New York, NY, USA (2009).

5. Gil, M., Giner, P., Pelechano, V.: Service obtrusiveness adaptation. Proceedings of the First international joint conference on Ambient intelligence. pp. 11–20. Springer-Verlag, Berlin, Heidelberg (2010).

6. Chen, H., Black, J.P.: A quantitative approach to non-intrusive computing. Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services. pp. 44:1–44:10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium (2008).

7. Abowd, G.D., Dey, A.K., Brown, P.J., Davies, N., Smith, M., Steggles, P.: Towards a Better Understanding of Context and Context-Awareness. Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing. pp. 304–307. Springer-Verlag, London, UK, UK (1999).

8. Abowd, G.D., Atkeson, C.G., Hong, J., Long, S., Kooper, R., Pinkerton, M.: Cyberguide: a mobile context-aware tour guide. Wirel. Netw. 3, 421–433 (1997).

9. Cheverst, K., Mitchell, K., Davies, N.: The role of adaptive hypermedia in a context-aware tourist GUIDE. Communications of the ACM. 45, 47–51 (2002).

10. Judd, G., Steenkiste, P.: Providing Contextual Information to Pervasive Computing Applications. Proceedings of the First IEEE International Conference on Pervasive Computing and Communications. p. 133. IEEE Computer Society, Washington, DC, USA (2003).

11. Chen, H., Finin, T., Joshi, A.: An ontology for context-aware pervasive computing environments. Knowl. Eng. Rev. 18, 197–207 (2003).

12. Zander, S., Schandl, B.: A framework for context-driven RDF data replication on mobile devices. Proceedings of the 6th International Conference on Semantic Systems. pp. 22:1–22:5. ACM, New York, NY, USA (2010).

13. Weiß, D., Duchon, M., Fuchs, F., Linnhoff-Popien, C.: Context-aware personalization for mobile multimedia services. Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia. pp. 267–271. ACM, New York, NY, USA (2008).

14. Kindberg, T., Barton, J.J.: A Web-based nomadic computing system. Computer Networks. 35, 443–456 (2001).

15. Bolchini, C., Curino, C., Schreiber, F.A., Tanca, L.: Context Integration for Mobile Data Tailoring. Proceedings of the 7th International Conference on Mobile Data Management. p. 5. IEEE Computer Society, Washington, DC, USA (2006).

16. Euzenat, Jé., Pierson, Jé., Ramparany, F.: Dynamic context management for pervasive applications. Knowl. Eng. Rev. 23, 21–49 (2008).

17. Reynolds, V., Hausenblas, M., Polleres, A., Hauswirth, M., Hegde, V.: Exploiting linked open data for mobile augmented reality. W3C Workshop: Augmented Reality on the Web (2010).

18. Zander, S., Chiu, C., Sageder, G.: A computational model for the integration of linked data in mobile augmented reality applications. Proceedings of the 8th International Conference on Semantic Systems. pp. 133–140. ACM, New York, NY, USA (2012).

19. Holleis, P., Wagner, M., Böhm, S., Koolwaaij, J.: Studying mobile context-aware social services in the wild. Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries. pp. 207–216. ACM, New York, NY, USA (2010).

20. Adida, B., Herman, I., Sporny, M., Birbeck, M.: RDFa 1.1 Primer, http://www.w3.org/TR/xhtml-rdfa-primer/.

21. Hickson, I.: HTML Microdata, http://dev.w3.org/html5/md-LC/.

22. DCMI Metadata Terms, http://dublincore.org/documents/dcmi-terms/.

23. Brickley, D., Miller, L.: FOAF Vocabulary Specification 0.98, http://xmlns.com/foaf/spec/.

24. Xue, W., Pung, H., Palmes, P.P., Gu, T.: Schema matching for context-aware computing. Proceedings of the 10th international conference on Ubiquitous computing. pp. 292–301. ACM, New York, NY, USA (2008).

25. Tummala, H., Jones, J.: Developing spatially-aware content management systems for dynamic, location-specific information in mobile environments. Proceedings of the 3rd ACM international workshop on Wireless mobile applications and services on WLAN hotspots. pp. 14–22. ACM, New York, NY, USA (2005).

26. Challiol, C., Rossi, G., Gordillo, S., De Cristófolo, V.: Designing and Implementing Physical Hypermedia Applications. In: Gavrilova, M., Gervasi, O., Kumar, V., Tan, C., Taniar, D., Laganà, A., Mun, Y., and Choo, H. (eds.) Computational Science and Its Applications - ICCSA 2006. pp. 148–157. Springer Berlin / Heidelberg (2006).

27. Challiol, C., Fortier, A., Gordillo, S.E., Rossi, G.: Model-based concerns mashups for mobile hypermedia. Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia. pp. 170–177. ACM, New York, NY, USA (2008).

28. Le-Phuoc, D., Parreira, J.X., Reynolds, V., Hauswirth, M.: RDF On the Go: An RDF Storage and Query Processor for Mobile Devices. 9th International Semantic Web Conference (ISWC2010) (2010).

29. Van Woensel, W., Casteleyn, S., De Troyer, O.: A generic approach for on-the-fly adding of context-aware features to existing websites. Proceedings of the 22nd ACM conference on Hypertext and hypermedia. pp. 143–152. ACM, New York, NY, USA (2011).

30. Van Woensel, W., Gil, M., Casteleyn, S., Serral, E., Pelechano, V.: Adapting the obtrusiveness of service interactions in dynamically discovered environments. 9th International Conference on Mobile and Ubiquitous Systems. , Beijing, China (2012).

31. Van Woensel, W., Casteleyn, S., De Troyer, O.: Applying semantic web technology in a mobile setting: the person matcher. Proceedings of the 10th international conference on Web engineering. pp. 506–509. Springer-Verlag, Berlin, Heidelberg (2010).

32. Casteleyn, S., Van Woensel, W., De Troyer, O.: Assisting mobile web users: client-side injection of context-sensitive cues into websites. Proceedings of the 12th International Conference on Information Integration and Web-based Applications &#38; Services. pp. 443–450. ACM, New York, NY, USA (2010).

33. Gil, M., Giner, P., Pelechano, V.: Personalization for unobtrusive service interaction. Personal Ubiquitous Comput. 16, 543–561 (2012).

34. Chaari, T., Laforest, F., Celentano, A.: Adaptation in context-aware pervasive information systems: the SECAS project. International Journal of Pervasive Computing and Communications. 3, 400–425 (2008).

35. Schilit, B., Theimer, M.: Disseminating Active Map Information to Mobile Hosts. IEEE Network. 8, 22–32 (1994).

36. Salber, D., Dey, A.K., Abowd, G.D.: The context toolkit: aiding the development of context-enabled applications. Proceedings of the SIGCHI conference on Human Factors in Computing Systems. pp. 434–441. ACM, New York, NY, USA (1999).

37. Pascoe, J.: Adding generic contextual capabilities to wearable computers. Wearable Computers, 1998. Digest of Papers. Second International Symposium on. pp. 92–99 (1998).

38. Siewiorek, D., Smailagic, A., Furukawa, J., Krause, A., Moraveji, N., Reiger, K., Shaffer, J., Wong, F.L.: SenSay: A Context-Aware Mobile Phone. Proceedings of the 7th IEEE International Symposium on Wearable Computers. p. 248. IEEE Computer Society, Washington, DC, USA (2003).

39. Espinoza, F., Persson, P., S, A., Nystrom, H., Cacciatore, E., Bylund, M.: GeoNotes: Social and Navigational Aspects of Location-Based Information Systems. Presented at the (2001).

40. Lopez-De-Ipina, D., Vazquez, J.I., Abaitua, J.: A context-aware mobile mash-up platform for ubiquitous web. Intelligent Environments, 2007. IE 07. 3rd IET International Conference on. pp. 116–123 (2007).

41. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context&#45;aware systems. Int. J. Ad Hoc Ubiquitous Comput. 2, 263–277 (2007).

42. Lee, S., Chang, J., Lee, S.: Survey and trend analysis of context-aware systems. Information-An International Interdisciplinary Journal. 14, 527–548 (2011).

43. Dey, A.K., Abowd, G.D., Salber, D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Hum.-Comput. Interact. 16, 97–166 (2001).

44. Barton, J., Kindberg, T.: The Challenges and Opportunities of Integrating the Physical World and Networked Systems, (2001).

45. Barton, J., Barton, J.J., Kindberg, T.: The CoolTown User Experience. (2001).

46. Debaty, P., Caswell, D.: Uniform Web presence architecture for people, places, and things. Personal Communications, IEEE. 8, 46–51 (2001).

47. Debaty, P., Goddi, P., Vorbau, A.: Integrating the physical world with the web to enable context-enhanced mobile services. Mob. Netw. Appl. 10, 385–394 (2005).

48. Wellner, P.: Interacting with paper on the DigitalDesk. Commun. ACM. 36, 87–96 (1993).

49. Lepetit, V.: On Computer Vision for Augmented Reality. Ubiquitous Virtual Reality, 2008. ISUVR 2008. International Symposium on. pp. 13–16 (2008).

50. Savidis, A., Zidianakis, M., Kazepis, N., Dubulakis, S., Gramenos, D., Stephanidis, C.: An Integrated Platform for the Management of Mobile Location-Aware Information Systems. Proceedings of the 6th International Conference on Pervasive Computing. pp. 128–145. Springer-Verlag, Berlin, Heidelberg (2008).

51. Priyantha, N.B., Chakraborty, A., Balakrishnan, H.: The Cricket location-support system. Proceedings of the 6th annual international conference on Mobile computing and networking. pp. 32–43. ACM, New York, NY, USA (2000).

52. Prehofer, C., van Gurp, J., di Flora, C.: Towards the web as a platform for ubiquitous applications in smart spaces. Second Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI), at Ubicomp (2007).

53. Bouvin, N.O., Christensen, B., Grønbaek, K., Hansen, F.A.: HyCon: a framework for context-aware mobile hypermedia. New Review of Hypermedia and Multimedia. 9, 59–88 (2003).

54. Niu, W., Kay, J.: Location Conflict Resolution with an Ontology. Proceedings of the 6th International Conference on Pervasive Computing. pp. 162–179. Springer-Verlag, Berlin, Heidelberg (2008).

55. Cappiello, C., Comuzzi, M., Mussi, E., Pernici, B.: Context Management for Adaptive Information Systems. Electron. Notes Theor. Comput. Sci. 146, 69–84 (2006).

56. Frkovic, F., Podobnik, V., Trzec, K., Jezic, G.: Agent-Based User Personalization Using Context-Aware Semantic Reasoning. Proceedings of the 12th international conference on Knowledge-Based Intelligent Information and Engineering Systems, Part I. pp. 166–173. Springer-Verlag, Berlin, Heidelberg (2008).

57. Gu, T., Pung, H.K., Zhang, D.Q.: A middleware for building context-aware mobile services. Proceedings of IEEE Vehicular Technology Conference (VTC). , Milan, Italy (2004).

58. Weiss, C., Bernstein, A., Boccuzzo, S.: i-MoCo: Mobile Conference Guide Storing and querying huge amounts of Semantic Web data on the iPhone-iPod Touch. Semantic Web Challenge 2008 (2008).

59. Hofer, T., Schwinger, W., Pichler, M., Leonhartsberger, G., Altmann, J., Retschitzegger, W.: Context-Awareness on Mobile Devices - the Hydrogen Approach. Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS’03) - Track 9 - Volume 9. p. 292. IEEE Computer Society, Washington, DC, USA (2003).

60. Roduner, C., Langheinrich, M.: Publishing and discovering information and services for tagged products. Proceedings of the 19th international conference on Advanced information systems engineering. pp. 501–515. Springer-Verlag, Berlin, Heidelberg (2007).

61. Harper, S., Goble, C.A., Pettitt, S.: proXimity: Walking the Link. J. Digit. Inf. -1–1 (2004).

62. Bolchini, C., Quintarelli, E.: Filtering mobile data by means of context: a methodology. Springer-Verlag, LNCS 4278. pp. 1986–1995 (2006).

63. Bolchini, C., Curino, C.A., Orsi, G., Quintarelli, E., Schreiber, F.A., Tanca, L.: CADD: a tool for context modeling and data tailoring. Proc. IEEE Intl. Conf. on Mobile Data Management (MDM). pp. 221–223 (2007).

64. Bolchini, C., Curino, C.A., Orsi, G., Quintarelli, E., Rossato, R., Schreiber, F.A., Tanca, L.: And what can context do for data? Commun. ACM. 52, 136–140 (2009).

65. Wilson, M., Russell, A., Smith, D.A., Owens, A., Schraefel, M.C.: mSpace Mobile: A Mobile Application for the Semantic Web. User Semantic Web Workshop, ISWC2005 (2005).

66. Euzenat, J.: Alignment infrastructure for ontology mediation and other applications. Proceedings of the First International workshop on Mediation in semantic web services. pp. 81–95 (2005).

67. Quilitz, B., Leser, U.: Querying distributed RDF data sources with SPARQL. ESWC'08: Proceedings of the 5th European semantic web conference on The semantic web. pp. 524–538. Springer-Verlag, Berlin, Heidelberg (2008).

68. Langegger, A., Wöß, W., Blöchl, M.: A semantic web middleware for virtual data integration on the web. Proceedings of the 5th European semantic web conference on The semantic web: research and applications. pp. 493–507. Springer-Verlag, Berlin, Heidelberg (2008).

69. Lynden, S., Kojima, I., Matono, A., Tanimura, Y.: Adaptive Integration of Distributed Semantic Web Data. Databases in Networked Information Systems. 5999, 174–193 (2010).

70. Stuckenschmidt, H., Vdovjak, R., Broekstra, J., Houben, G.: Towards distributed processing of RDF path queries. Int. J. Web Eng. Technol. 2, 207–230 (2005).

71. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. Presented at the (2005).

72. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. VLDB Endow. 1, 1008–1019 (2008).

73. Dar, S., Franklin, M.J., Jónsson, B.T., Srivastava, D., Tan, M.: Semantic Data Caching and Replacement. VLDB ’96: Proceedings of the 22th International Conference on Very Large Data Bases. pp. 330–341. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996).

74. Jónsson, B.T., Arinbjarnar, M., Thórsson, B., Franklin, M.J., Srivastava, D.: Performance and overhead of semantic cache management. ACM Trans. Internet Technol. 6, 302–331 (2006).

75. Ren, Q., Dunham, M.H., Kumar, V.: Semantic Caching and Query Processing. IEEE Trans. on Knowl. and Data Eng. 15, 192–210 (2003).

76. Ren, Q., Dunham, M.H.: Using semantic caching to manage location dependent data in mobile computing. MobiCom ’00: Proceedings of the 6th annual international conference on Mobile computing and networking. pp. 210–221. ACM, New York, NY, USA (2000).

77. Cui, Y., Roto, V.: How people use the web on mobile devices. Proceedings of the 17th international conference on World Wide Web. pp. 905–914. ACM, New York, NY, USA (2008).

78. Setten, M. Van, Pokraev, S., Koolwaaij, J., Instituut, T.: Context-aware recommendations in the mobile tourist application COMPASS. In Nejdl, W. & De Bra, P. (Eds.). AH 2004, LNCS 3137. pp. 235–244. Springer-Verlag (2004).

79. Metzger, C., Ilic, A., Bourquin, P., Michahelles, F., Fleisch, E.: Distance-sensitive High Frequency RFID Systems. Pervasive Computing and Applications, 2008. ICPCA 2008. Third International Conference on. pp. 729–734.

80. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995).

81. Brickley, D.: Basic Geo (WGS84 lat/long) Vocabulary, http://www.w3.org/2003/01/geo/.

82. OpenGIS® Implementation Specification: Coordinate Transformation Services. (2001).

83. Silberschatz, A., Korth, H., Sudarshan, S.: Database System Concepts. (2010).

84. Díaz, O., Arellano, C., Azanza, M.: A language for end-user web augmentation: Caring for producers and consumers alike. ACM Trans. Web. 7, 9:1–9:51 (2013).

85. Ziegler, C.: Semantic web recommender systems. In Proceedings of the Joint ICDE/EDBT Ph.D. Workshop 2004 (Heraklion. pp. 78–89. Springer-Verlag (2004).

86. Van Woensel, W., Casteleyn, S., Paret, E., De Troyer, O.: Transparent mobile querying of online RDF sources using semantic indexing and caching. Proceedings of the 12th international conference on Web information system engineering. pp. 185–198. Springer-Verlag, Berlin, Heidelberg (2011).

87. Safa, H., Artail, H., Nahhas, M.: A cache invalidation strategy for mobile networks. J. Netw. Comput. Appl. 33, 168–182 (2010).

88. Xu, J., Tang, X., Lee, D.L.: Performance Analysis of Location-Dependent Cache Invalidation Schemes for Mobile Environments. IEEE Trans. on Knowl. and Data Eng. 15, 474–488 (2003).

89. Zheng, B., Lee, W.-C., Lee, D.L.: On semantic caching and query scheduling for mobile nearest-neighbor search. Wirel. Netw. 10, 653–664 (2004).

90. Patel-Schneider, P.F., Horrocks, I.: A comparison of two modelling paradigms in the Semantic Web. Web Semant. 5, 240–250 (2007).

91. Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space. (2011).

92. Paret, E., Van Woensel, W., Casteleyn, S., Signer, B., De Troyer, O.: Efficient Querying of Distributed RDF Sources in Mobile Settings based on a Source Index Model. In: Shakshuki, E. and Younas, M. (eds.) Proceedings of the 8th International Conference on Mobile Web Information Systems (MobiWIS 2011). pp. 554–561 (2011).

93. Van Woensel, W., Casteleyn, S., Paret, E., De Troyer, O.: Mobile Querying of Online Semantic Web Data for Context-Aware Applications. Internet Computing, IEEE. 15, 32–39.

94. Biddulph, M.: Semantic web crawling. XML Europe (2004).

95. Dobbs, L.: Slug: A Semantic Web Crawler, http://www.ldodds.com/projects/slug/slug-a-semantic-web-crawler.pdf.

96. Ding, L., Finin, T., Joshi, A., Pan, R., Cost, R.S., Peng, Y., Reddivari, P., Doshi, V., Sachs, J.: Swoogle: a search and metadata engine for the semantic web. Proceedings of the thirteenth ACM international conference on Information and knowledge management. pp. 652–659. ACM, New York, NY, USA (2004).

97. Chen, Y., Xie, X., Ma, W.-Y., Zhang, H.-J.: Adapting Web Pages for Small-Screen Devices. IEEE Internet Computing. 9, 50–56 (2005).

98. Paternò, F., Zichittella, G.: Desktop-to-Mobile Web Adaptation through Customizable Two-Dimensional Semantic Redesign. In: Bernhaupt, R., Forbrig, P., Gulliksen, J., and Lárusdóttir, M. (eds.) HCSE. pp. 79–94. Springer (2010).

99. Malandrino, D., Mazzoni, F., Riboni, D., Bettini, C., Colajanni, M., Scarano, V.: MIMOSA: context-aware adaptation for ubiquitous web access. Personal and Ubiquitous Computing. 14, 301–320 (2010).

100. Murugesan, S., Venkatakrishnan, B.A.: Addressing the Challenges of Web Applications on Mobile Handheld Devices. ICMB '05: Proceedings of the International Conference on Mobile Business. pp. 199–205. IEEE Computer Society, Washington, DC, USA (2005).

101. Van der Sluijs, K., Houben, G.-J., Broekstra, J., Casteleyn, S.: Hera-S: web design using sesame. Proceedings of the 6th international conference on Web engineering. pp. 337–344. ACM, New York, NY, USA (2006).

102. Nebeling, M., Grossniklaus, M., Leone, S., Norrie, M.C.: Domain-specific language for context-aware web applications. Proceedings of the 11th international conference on Web information systems engineering. pp. 471–479. Springer-Verlag, Berlin, Heidelberg (2010).

103. Ceri, S., Dolog, P., Matera, M., Nejdl, W.: Adding client-side adaptation to the conceptual design of e-learning web applications. J. Web Eng. 4, 21–37 (2005).

104. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002).

105. Dolog, P., Nejdl, W.: Using UML and XMI for Generating Adaptive Navigation Sequences in Web-Based Systems. Proc. of UML 2003: The Unified Modeling Language. Model Languages and Applications. 6th Intl. Conference, volume 2863 of LNCS, 205-219. pp. 205–219. Springer (2003).

106. Ceri, S., Daniel, F., Facca, F.M., Matera, M.: Model-driven Engineering of Active Context-awareness. World Wide Web. 10, 387–413 (2007).

107. Putzinger, A.: Towards Asynchronous Adaptive Hypermedia: An Unobtrusive Generic Help System. LWA. pp. 383–388 (2007).

108. Chang, C.-H., Kayed, M., Girgis, M.R., Shaalan, K.F.: A Survey of Web Information Extraction Systems. IEEE Trans. on Knowl. and Data Eng. 18, 1411–1428 (2006).

109. Chang, C.-H., Hsu, C.-N., Lui, S.-C.: Automatic information extraction from semi-structured Web pages by pattern discovery. Decis. Support Syst. 35, 129–147 (2003).

110. Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., Riboni, D.: A survey of context modelling and reasoning techniques. Pervasive Mob. Comput. 6, 161–180 (2010).

111. Knutov, E., De Bra, P., Pechenizkiy, M.: AH 12 years later: a comprehensive survey of adaptive hypermedia methods and techniques. New Rev. Hypermedia Multimedia. 15, 5–38 (2009).

112. Brusilovsky, P.: Adaptive Hypermedia. User Modeling and User-Adapted Interaction. 11, 87–110 (2001).

113. Chittaro, L.: Distinctive aspects of mobile interaction and their implications forÂ theÂ design of multimodal interfaces. Journal on Multimodal User Interfaces. 3, 1783–7677 (2010).

114. Ho, J., Intille, S.S.: Using context-aware computing to reduce the perceived burden of interruptions from mobile devices. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 909–918. ACM, New York, NY, USA (2005).

115. Valtonen, M., Vainio, A.-M., Vanhala, J.: Proactive and adaptive fuzzy profile control for mobile phones. Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications. pp. 1–3. IEEE Computer Society, Washington, DC, USA (2009).

116. Serral, E., Valderas, P., Pelechano, V.: Improving the Cold-Start Problem in User Task Automation by Using Models at Runtime. In: Pokorny, J., Repa, V., Richta, K., Wojtkowski, W., Linger, H., Barry, C., and Lang, M. (eds.) Information Systems Development. pp. 671–683. Springer New York (2011).

117. Pan, G., Xu, Y., Wu, Z., Li, S., Yang, L., Lin, M., Liu, Z.: TaskShadow: Toward Seamless Task Migration across Smart Environments. Intelligent Systems, IEEE. 26, 50–57 (2011).

118. Assad, M., Carmichael, D.J., Kay, J., Kummerfeld, B.: PersonisAD: distributed, active, scrutable model framework for context-aware services. Proceedings of the 5th international conference on Pervasive computing. pp. 55–72. Springer-Verlag, Berlin, Heidelberg (2007).

119. Ramchurn, S.D., Deitch, B., Thompson, M.K., De Roure, D.C., Jennings, N.R., Luck, M.: Minimising intrusiveness in pervasive computing environments using multi-agent negotiation. Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on. pp. 364–371 (2004).

120. Toninelli, A., Pantsar-Syväniemi, S., Bellavista, P., Ovaska, E.: Supporting context awareness in smart environments: a scalable approach to information interoperability. Proceedings of the International Workshop on Middleware for Pervasive Mobile and Embedded Computing. pp. 5:1–5:4. ACM, New York, NY, USA (2009).

121. Retkowitz, D., Armac, I., Nagl, M.: Towards Mobility Support in Smart Environments. SEKE. pp. 603–608. Knowledge Systems Institute Graduate School (2009).

122. Hadim, S., Al-Jaroodi, J., Mohamed, N.: Trends in Middleware for Mobile Ad Hoc Networks. JCM. 1, 11–21 (2006).

123. Meier, R., Cahill, V.: STEAM: Event-Based Middleware for Wireless Ad Hoc Network. Proceedings of the 22nd International Conference on Distributed Computing Systems. pp. 639–644. IEEE Computer Society, Washington, DC, USA (2002).

124. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer. 36, 41–50 (2003).

125. Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. Computer. 42, 37–43 (2009).

126. Ju, W., Leifer, L.: The Design of Implicit Interactions: Making Interactive Systems Less Obnoxious. Design Issues. 24, 72–84 (2008).

127. Lewis, J.R.: IBM computer usability satisfaction questionnaires: psychometric evaluation and instructions for use. Int. J. Hum.-Comput. Interact. 7, 57–78 (1995).

128. El-Sofany, H.F., El-Seoud, S.A.: Mobile Tourist Guide - An Intelligent Wireless System to Improve Tourism, using Semantic Web. iJIM. 5, 4–10 (2011).

129. Dolog, P., Stuckenschmidt, H., Wache, H., Diederich, J.: Relaxing RDF queries based on user and domain preferences. J. Intell. Inf. Syst. 33, 239–260 (2009).