

University of Manchester
School of Computer Science
Project Report April 2013

**Matchmaking Skills Using
OWL Ontologies**

Author: Iulia Andrada Ungureanu
BSc(Hons) Computer Science
with Business and Management

Supervisor: Ulrike Sattler

0.1 Abstract

Matching individuals with positions is essential in hiring and managing human resources. With online recruitment and organisational knowledge management becoming increasingly popular, there is a growing interest in good quality retrieval of people by certain skills; the need for an approach that would match all and only suitable candidates and rank them according to their offering has developed in turn. As traditional keyword search is prone to both false positives and false negatives, with thinly populated results for very specialised skill requests, we refer to OWL ontologies as an alternative.

This report investigates the suitability of using an OWL ontology to model candidate skills and relationships between them, which will be used in querying and retrieving results in a software application. Following recruitment standard procedures, we distinguish between compulsory and desirable skills, to differentiate between candidates and ensure a wider space of results, which we rank in a most-promising order.

The report outlines the design, implementation and evaluation of this approach, which resulted in the development of a rich OWL ontology of skills and individuals, a matchmaker tool that retrieves and displays the matching candidates, using a dynamic user interface. An in depth-case study to compare keyword search to our ontology is also presented, which illustrates OWL's advantages over the traditional method, as well as some disadvantages.

Project Title: Matchmaking Skills Using OWL Ontologies

Author: Iulia Andrada Ungureanu

Degree: BSc(Hons) Computer Science with Business and Management

Supervisor: Ulrike Sattler

0.2 Acknowledgements

I would like to thank everyone who has helped and supported me through my final and toughest year at university. Many and special thanks to my supervisor, Uli Sattler, who has provided invaluable help, motivation and inspiration to achieve great things. Thanks also go to my parents, who never doubted my ability to succeed in all my endeavours.

Contents

0.1	Abstract	1
0.2	Acknowledgements	2
1	Introduction	7
1.1	Motivation	7
1.2	Existing Approaches	7
1.3	Project Proposal: Aims and Objectives	8
1.4	Report Synopsis	9
2	Background and Literature Survey	10
2.1	Overview	10
2.2	Semantic Web and Ontologies	10
2.2.1	Ontologies	10
2.2.2	Description Logics	10
2.3	The Web Ontology Language OWL	11
2.3.1	Overview	11
2.3.2	OWL Ontologies	11
2.3.3	OWL Reasoners	13
2.3.4	Protégé	14
2.3.5	OWL API	14
2.4	Matchmaking Skills with OWL Ontologies	15
3	Requirements and Design	17
3.1	Overview	17
3.2	Development Approach	17
3.3	Project Requirements	17
3.3.1	Ontology Requirements	17
3.3.2	Software Requirements	18
3.4	Architecture	19
3.5	Ontology Design	20
3.5.1	Overview	20
3.5.2	Representing the Offer	21
3.5.3	Representing the Request	21
3.6	Java Application	22
3.6.1	Overview	22
3.6.2	User Interface Design	23
3.7	Summary	23

4	Implementation	24
4.1	Modelling the Ontology	24
4.1.1	Overview	24
4.1.2	Defining Classes and Their Properties	25
4.1.3	Defining Individuals	26
4.2	Java Application Implementation	27
4.2.1	Building the User Interface	27
4.2.2	Matching and Ranking Results	28
4.3	Challenges	29
4.4	Summary	30
5	Results	31
5.1	Overview	31
5.2	Functional Requirements	31
5.2.1	System Walkthrough	32
5.3	Non-functional Requirements	33
6	Testing and Evaluation	35
6.1	Overview	35
6.2	Manual Testing	35
6.3	Unit Testing	36
6.4	Performance and Scalability Testing	37
6.4.1	Motivation	37
6.4.2	Experiments and Results	37
6.5	Case Study	39
6.5.1	Overview	39
6.5.2	The Job Advert	40
6.5.3	Matching With the Ontology-based Matchmaker	40
6.5.4	Matching With DocFetcher	41
6.5.5	Comparing Results	41
6.5.6	What About Structured Documents?	42
6.5.7	Case Study Summary	43
6.6	Summary	43
7	Summary and Outlook	44
7.1	Achievements	44
7.2	Future Work	45
A	Full Ontology Hierarchy	49
B	OWL API Main Methods Used	51
C	Case Study - Experiment Queries	52
D	Case Study - Match Maker Results Breakdown	53

List of Figures

2.1	Classes, Individuals and Properties Example	12
2.2	Protégé User Interface	14
3.1	Architecture	19
3.2	Ontology Design (TBox) Overview	20
3.3	Ideal Offer Representation	21
3.4	Representing the Offer - Example	21
3.5	Representing the Request: A simplified example	22
3.6	Application Design	22
3.7	User Interface Wireframe	23
4.1	Core Ontology Superclasses	24
4.2	Example of using <code>hasPart</code>	26
4.3	Individuals in the Ontology	27
4.4	Skills Autocompletion Example	28
4.5	Proof Customisation	28
5.1	User Interface Final Version	31
5.2	Selecting Expertise Level	32
5.3	Selecting the Type of Experience and Duration	32
5.4	Adding More Skills	33
6.1	Manual Testing with the DL Query Tab	36
6.2	Application Run Time	37
6.3	Run Time by Desirable Queries - with <i>Hermit</i>	38
6.4	Run Time by Desirable Queries - with <i>Fact++</i>	38
6.5	Run Time By Compulsory Candidate Pool Coverage	39
6.6	Representing CVs for Text Matching	41
6.7	Ontology Match Maker Results	42
6.8	DocFetcher Results	42
6.9	DocFetcher Incorrect Match	42

List of Tables

3.1	Functional Ontology Requirements: Groups of Competency Questions	18
3.2	Non-Functional Ontology Requirements	18
3.3	Subclasses and Properties in the Skills Ontology	20
4.1	Distinguishing between Skills Descriptions using Object Properties	25
4.2	Object Properties in the Ontology - Domains and Ranges	25
6.1	Junior Software Developer Advert	40
6.2	User Input for the Match Maker GUI	40

Chapter 1

Introduction

1.1 Motivation

The idea behind building a tool that would retrieve people according to their skills came up from spotting a real need for such an application throughout the student's industrial placement. Whilst working in managing urgent problems with IT systems, particularly issues that were causing business disruptions, we would need to contact people with the right skills as soon as possible. Since there was no way of knowing who knew what, apart from word of mouth and personal relationships, an application to quickly suggest a person based on a skill would have saved a lot of time (and money).

Further research into the area showed us that there is a growing interest in good quality retrieval and matching of people skills, for a variety of purposes. Examples include online recruitment, which looks at matching job advertisements with resumes or skills profiles, or internal organisations needs, such as assigning individuals to tasks and teams based on skills and qualifications or finding an expert in a certain area.

The problem of matching people according to their skill set turned out to be complex: despite a very large number of skills specified by people's profiles or resumes, the space for specialised, specific requests is thinly populated, mainly due to the lack of or incomplete structuring of these skills.

1.2 Existing Approaches

There are various approaches to matchmaking skills, each with its advantages and disadvantages.

Keyword Search with Unstructured Text

The basic way of tackling the problem is by performing a keyword search through unstructured data, where the skills are viewed as simple strings, which are to be syntactically matched to a set of relevant words. The results could be based on an all-or-nothing method, retrieving only exact matches, or a more systematic, vector-based technique. The latter could be approached via simple frequency counts for each keyword, or, in more depth, by using weights for different skills to get a more relevant match (similar to the COINS matchmaker) [8]. This syntactic

approach is very fast, easy and cheap to set-up¹ and efficient to some extent, yet it completely ignores background knowledge and semantics. For instance, if we were to request candidates with programming skills and one would only name a specific one in their CV (such as C++), the candidate would not be a match, despite clearly being able to programme.

Retrieval with Taxonomies

Nonetheless, some skills may come in structured formats, such as taxonomies, where they would be hierarchically classified in superclass-subclass relationships. This knowledge would help solve the earlier issue, where, if C++ were modelled as a sub-class of Programming, the candidate would be a match. However, simple "is-A" type of relationships may lead to false positives and false negatives. For example, if someone specified Microsoft Excel as a skill, and that was modelled as a subclass of Microsoft Office, this would imply that the person was competent in the entire Microsoft Office package, which may not be accurate. Such cases show the need for a more expressive way of structuring skills and their properties, to clearly distinguish particular relationships (such as the part-of example).

Modelling and Matching with Ontologies

In the context of the Semantic Web approaches gaining momentum and starting to revolutionise the way information is provided on the Web [13], as well as our intention to not restrict our matching to simple string comparisons, we decided to use ontologies to semantically describe and match skills descriptions. Based on Description Logics (DL) formalisation and reasoning, ontologies allow for more than simple hierarchical structuring, providing the ability to describe concepts, properties and the relationships between them. In addition, DL comes with reasoning services, allowing a deduction of knowledge which has not been explicitly specified.

Through these opportunities to increase the expressivity in describing our domain, we expected to see better results with regard to the quality of retrieval, but also to be able to provide the user with a more flexible way of searching for skills.

This leads us to our project proposal and objectives, focused around an ontology-based matchmaking of skills.

1.3 Project Proposal: Aims and Objectives

Having in mind the drawbacks of text-matching techniques and the potential gains of using ontologies to model skills, this project will be focused on two main aspects:

- *Implementing an ontology-based skills matchmaking tool*
We aimed to build a software application that would retrieve candidates for a given skill set, allowing the user to distinguish between compulsory and optional skills, but also add preferences with regard to proof for the skill, competency level or experience. The skills, their properties and relationships would be modelled in an extensible ontology, with which the application would communicate in real time. The candidates found would be ranked according to how well they matched the request.
Achieving this objective involves a thorough study (and practice) of Description Logics

¹In terms of memory space, processing power, quick and easy implementation.

and its reasoning capabilities, the tool(s) required to build ontologies, an API to communicate with the ontology in the development environment, as well as knowledge and use of a programming language to build the application.

- *Investigating the benefits of the approach*

Having built the matchmaking tool, we aim on getting an insight into what we gained by using ontologies, in comparison to unstructured or other structured means (such as XML). We are interested in evaluating how our approach helps improve precision and/or recall and at what costs and disadvantages.

1.4 Report Synopsis

This section will give an overview of the following chapters:

Chapter 2 will provide the relevant background information regarding the semantic web technologies used, as well as an insight into ontology-based matchmaking approaches.

Chapter 3 provides a high level description of the design of the ontology and the matchmaking tool, after presenting the system requirements.

Chapter 4 describes in depth the implementation of the key components and algorithms of the matchmaking tool, as well as the challenges encountered.

Chapter 5 illustrates the achievement of the objectives and requirements by illustrating the results, through a brief system walkthrough.

Chapter 6 provides an overview of the different types of testing performed to evaluate the application, as well as a detailed case study reviewing the advantages and drawbacks of ontology-based matchmaking.

Chapter 7 summarises the achievements of the project, sharing the key lessons and conclusions drawn from the implementation and evaluation, as well as suggesting future work and improvements.

Chapter 2

Background and Literature Survey

2.1 Overview

This chapter will provide an overview of the tools, technologies and theoretical aspects necessary to the understanding of how the objectives illustrated earlier were tackled and achieved.

2.2 Semantic Web and Ontologies

The Semantic Web is a vision for the future of the World Wide Web which looks to give explicit meaning to the hyperlinked information, such that machines can interpret the web content they process and provide users meaningful and appropriate search results [6]. The key issue arising is representing this knowledge, whilst supporting its sharing and reuse.

2.2.1 Ontologies

Ontologies are a formal means of representing knowledge about a domain of interest. Based on conceptualisation,¹ an ontology is an explicit specification of the terms (concepts) in the domain, their properties or attributes and the relationships between them [12]. This specification often includes a hierarchical structure, with parent-child, "is-a" type of relationships.

Ontologies are highly useful not to just describe, but also integrate data; the agreement on the exact definition of concepts and their relations reduces ambiguity and leads to a common understanding of a certain area [25]. This is also facilitated by the lack of data models and structures typical to databases: *"Ontologies are typically specified in languages that allow abstraction away from data structures and implementation strategies"* [24].

2.2.2 Description Logics

Description Logics is a family of knowledge representation languages based on logic, which sets the foundation for ontology languages [2].

The basic elements of DL are *concept* names, such as `Person`, `Animal`, `Mother` and *role* names, such as `hasChild`, `hasParent`, the latter representing binary relationships between elements.

¹"A conceptualisation is an abstract, simplified view of the world that we wish to represent for some purpose." [12]

Concepts and roles are used to represent and formalise the terminology of the domain of interest, which is stored in a *TBox* (Terminological Box) [20]. The TBox corresponds to the schema of a database - it contains concept expressions, which would either define concepts or impose restrictions on possible interpretations [4]. For example, the following is a concept definition for a mother:

$\text{Mother} \equiv \text{Woman} \sqcap \exists \text{hasChild}.\text{Person}$

The next expression (or axiom), however, restricts the interpretation of the domain, stating that a person can have at most one mother:

$\text{Person} \sqsubseteq (\leq 1 \text{hasMother})$

DL is also used to describe concrete situations, on an individual level; this knowledge is captured in an *ABox* (Assertion Box). The ABox corresponds to the data setting in a database, containing a finite number of individual assertions that state memberships of concepts or roles [2, 10]. For example, the assertions:

Jessica : Woman
 (Jessica, Dan) : hasChild
 Diane : $\neg(\text{Mother})$

state that Jessica is a woman, that Dan is her child and that Diane is not a mother.

2.3 The Web Ontology Language OWL

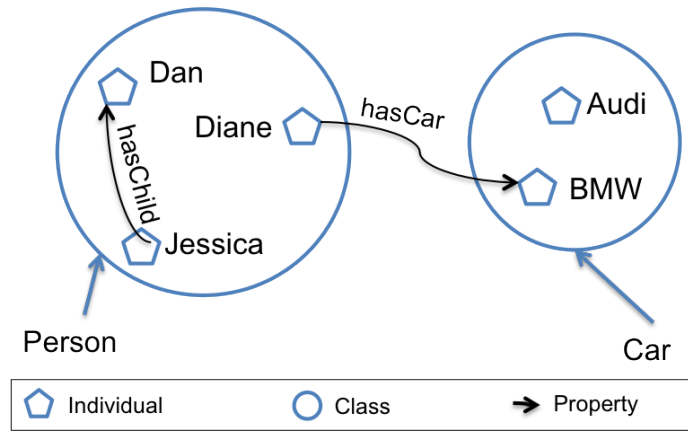
2.3.1 Overview

OWL is a semantic web language used to define and instantiate ontologies. OWL goes beyond XML, RDF, and RDF-S in expressing semantics, by adding more vocabulary to describe properties and classes in depth: relationships between classes (such as disjointness), extensive annotations, cardinality, equality, characteristics of properties (e.g. transitivity), etc. [6].

The basis of OWL lies in Description Logics, making use of its reasoning services, which help automatically derive logical consequences (not literally specified in the ontology) from the knowledge represented explicitly [1, 2].

2.3.2 OWL Ontologies

OWL ontologies closely follow the DL format, consisting of classes (corresponding to concepts), individuals (instances of classes) and properties (corresponding to roles, stating relationships between individuals). Figure 2.1 gives an example representation of these entities.

Figure 2.1: Classes, Individuals and Properties Example²

Classes

As illustrated in Figure 2.1, classes can be seen as sets of individuals with similar characteristics, that explicitly represent a real-world concept. OWL classes are typically organised in a hierarchical manner, with a superclass-subclass structure. For instance, *Woman* can be a superclass of *Mother* or of *Grandmother*, which means that any mother or grandmother is also a woman. These relationships are known as *subsumption relationships*, where subclasses are subsumed by their superclasses [14].

Apart from naming them, OWL classes can be defined using other classes and properties: for instance, a grandmother could be defined as a conjunction of *Mother* and the condition $\exists \text{hasChild.Mother}$.

Individuals

Individuals are the key entities in the domain of interest, defined and organised as members of one or more classes. OWL allows for users to explicitly declare individuals as equivalent or different, otherwise they are considered potentially equivalent, even if they have different names [14].

Properties

Object properties are OWL's version of DL roles, thus representing relationships between two individuals. For example, *hasCar* maps a *Person* (domain) to a *Car* (range).

An OWL ontology may also contain *data properties*, which specify relationships between an individual and data values. These can be used, for instance, to store the age of a *Person*, via a data property *hasAge* which would link an instance of a person to an *integer* representing their age. OWL allows for properties to have sub-properties, with subsumption relationships similar to classes. For example, *hasCar* may be modelled as a sub-property of *hasVehicle*.

Nevertheless, the conceptualisation effort in building ontologies would be useless if it were not for reasoners to automatically process and deduce knowledge from the stated properties and relationships.

²Adapted from Horridge [14]

2.3.3 OWL Reasoners

”A reasoner is a program that infers logical consequences from a set of explicitly asserted facts or axioms and typically provides automated support for reasoning tasks such as classification, debugging and querying.” [7]

According to Horrocks [15], one of the main motivations behind basing OWL on DL was the availability of reasoning systems. These help fulfil a variety of reasoning tasks, based on both the TBox and the ABox.

The main TBox reasoning services involve [3]:

- *Concept satisfiability*: checks if a class can have instances according to how the ontology is modelled;
- *Subsumption*: check whether a class subsumes another class;
- *Classification*: compute the entire ontology taxonomy entailed by the specified classes and properties.

Reasoners also perform tasks related to the ABox, usually at runtime:

- *Retrieval*: return all instances of a class or find all classes that an individual is an instance of;
- *Consistency*: check whether the set of assertions in the ABox is consistent;
- *Instance Check*: check if the assertions in the ABox entail that a certain individual is an instance of a given class.

For example, given the following Tbox and Abox:

```
Mother  $\equiv$  Woman  $\sqcap$   $\exists$ hasChild.Person
Jessica : Mother
Iulia: Woman
(Iulia, Dan): hasChild
```

An instance retrieval of the class `Mother` would return both `Jessica` and `Iulia`. The reasoner would automatically deduce that, because `Iulia` has a child and is a `Woman`, she is a `Mother`, despite this not having specifically been asserted in the Abox, as in the case of `Jessica`.

The key advantage (and guarantee) behind reasoners is that they implement decision procedures³ for the tasks mentioned above that are sound (return no wrong answers) and complete (return all the valid answers).

Reasoners can be used in software development environments, as a large number of them support the OWL API, a Java API we will be describing in a later section. Examples include: *FaCT++*⁴, *Hermit*⁵ or *Pellet*⁶.

³A decision procedure is an algorithm which terminates with a correct yes/no answer to a given decision problem [5].

⁴<http://owl.man.ac.uk/factplusplus/>

⁵<http://www.hermit-reasoner.com/>

⁶<http://clarkparsia.com/pellet/>

2.3.4 Protégé

Protégé⁷ is an open-source ontology editor based on Java, which provides capabilities for creating, visualising and manipulating ontologies. The tool is extensible, allowing for various plug-ins, such as *OWLviz* or *Ontograf*, which are used for visualisation. Figure 2.2 illustrates the Protégé user interface, showing the tabs corresponding to the main OWL ontology components (classes, object and data properties, individuals), but also to some of the plug-ins.

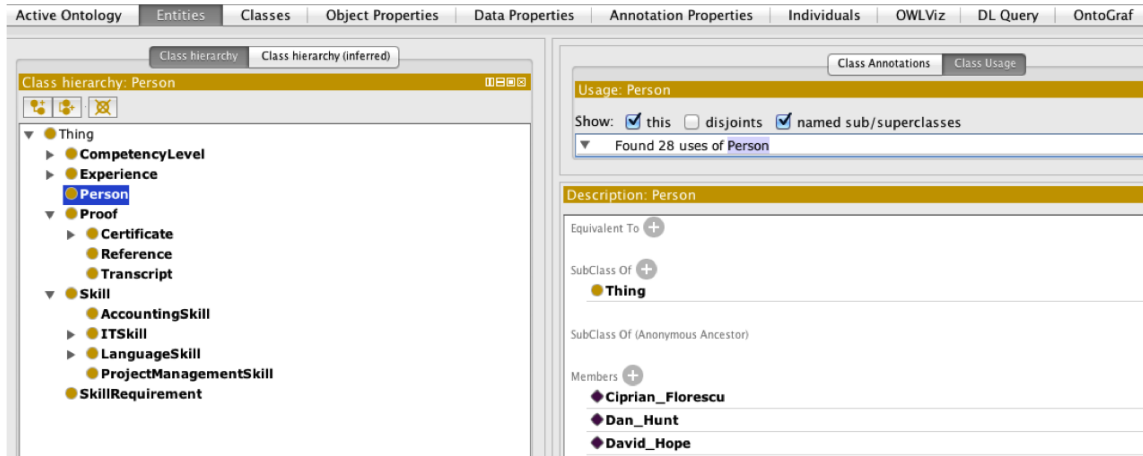


Figure 2.2: Protégé User Interface

Protégé uses OWL reasoners to give instant feedback to the user on the logical consequences of their ontology design.⁸

The user can also formulate DL queries (or class expressions) in Protégé, which uses the reasoners to retrieve classes, subclasses or individuals that satisfy the query. This feature is available in the *DL Query Tab*, which will be illustrated later in our testing examples (Figure 6.1).

Class expressions in Protégé are described in the *Manchester OWL Syntax*, a syntax which is quick and easy to write and read, even for users with limited or no background in DL.⁹ In the Manchester OWL Syntax, boolean class constructors are replaced with English language keywords (such as *and*, *or* and *not*), as well as the existential constructors (*some* for \exists , *only* for \forall). For example, the earlier definition for a mother would be represented as:

Woman and hasChild some Person

Protégé's features are mostly based on the use of the OWL API, presented next.

2.3.5 OWL API

The OWL API¹⁰ is a Java Application Programming Interface (API), used to create and manipulate OWL 2 ontologies. The API supports parsing and writing in OWL 2 syntaxes (such as RDF/XML or OWL/XML), as well as interfaces for working with reasoners (such as FaCT++,

⁷<http://protege.stanford.edu/>

⁸Notice the *Class Hierarchy Inferred* tab - this would show the results of a classification of the ontology done by the reasoner.

⁹Full guide to using the Manchester OWL Syntax with Protégé available at <http://www.code.org/downloads/manchesterowlsyntaxeditor/EditorGuide.pdf>

HermiT or Pellet). The reasoner interfaces of the OWL API facilitate incremental reasoning support, by allowing a reasoner to listen for ontology changes and process them immediately [7]. This allows the Java application to provide real time information and results from an ontology.

2.4 Matchmaking Skills with OWL Ontologies

We will be using the terms *offer* (O) and *request* (R) throughout the paper to refer to the candidate skill offering, respectively the user skills and preferences request, which we are interested in matching.

Significant research and academic literature has been dedicated to the idea of using ontologies for matchmaking problems, for skills as well as for other popular searches, such as houses or flats for rent [10], or product matching in online marketplaces [9].

In their argument for ontology based matchmaking, Du et al. [10] emphasise the disadvantages of keyword search in making little use of background knowledge and semantics, missing matches or returning incorrect ones. Di Noia et al. [8] further discuss the purely probabilistic nature of unstructured approaches, which leads to these false matches. For example, if the user was requesting a candidate with experience in working with agile methodologies, one may rank very well if he or she had the word "experience" many times in the CV, regardless of whether this experience is related to agile methodologies or not. Nonetheless, different approaches, which use ontologies, exist for the problem of matchmaking.

One approach models the offer and the request as concepts in an ontology, and relies on calculating distances between concepts to retrieve their compatibility [11]. The concepts with the shortest paths from the request would be seen as better matches. The technique does, however, come with various issues, that may go against the semantics of the ontology. For example, an offer that is subsumed by the request will have a distance different from zero, despite it being a complete match. This will result in the offer ranking lower than a literal (string) match, when it should have the same ranking.

Other approaches use DL inference methods to find matches for a given request. The most popular one involves *concept subsumption checking* ($O \sqsubseteq R$), corresponding to the offer fulfilling all the requirements of R . This method excludes the case when the request is inconsistent or incomparable to the offer, thus resulting in a limited space of results. Di Noia et al. [18] suggest the use of two other DL inference services for this problem, namely *concept abduction* and *concept contraction*, which would help review and rank counteroffers for the exact matches. Concept contraction, on the one hand, introduces the possibility to relax some of the constraints of the request when they are conflicting with the offer. Concept abduction, on the other hand, is used to refine the offer to make it more compatible with the request. An offer would be considered a match if it were subsumed by the request after abduction (adding further details to the offer) and contraction (removing some constraints from the request). Du et al. [10] suggest treating the matchmaking problem as a conjunctive query answering problem when performing abduction and contraction, to reduce human efforts in formalising requests. Their suggestions involves expressing the offer as individual assertions in the ontology and the request as a set of conjunctive queries expressed in terms of the offer. The match would simply be the answer to the request expressing the offer.

However, none of these approaches distinguishes between hard and soft constraints concerning

¹⁰<http://owlapi.sourceforge.net/>

the request. It is a standard procedure in recruitment to differentiate between mandatory and optional skill requirements, where the optional skills help distinguish between candidates and rank them accordingly.

The approaches presented and their analysis led to the design and development approach of our matchmaking tool, described in the next chapters.

Chapter 3

Requirements and Design

3.1 Overview

This chapter will cover the design of the system, with an overview of the overall architecture and in-depth descriptions of the main components: the skills ontology and the Java application. As the design closely followed the development approach and it was adapted to the project's requirements which, at times, changed, we will start the chapter with reviewing these first.

3.2 Development Approach

Knowing that requirements were bound to change as working code was being produced and more knowledge of the domain was gained, we decided to follow an iterative and evolutionary approach in building both the ontology and the software. The development was organised in short, mostly weekly iterations, where we would review the requirements and design, implement certain functionalities and test that they are working as expected. In doing so, we were open to changes in requirements, despite it potentially affecting the design and implementation.

3.3 Project Requirements

As the stakeholders were also the developers of this project, the requirements were treated in a flexible manner, with unplanned changes arising at times. These changes would occur when there was a possibility for more or better functionalities or when it seemed that the main objectives would be reached more effectively through a different approach. Yet a longer path would also be chosen when it would provide the developer, new to the field, a chance to explore and learn about different areas of Description Logics.¹

3.3.1 Ontology Requirements

Functional Requirements

Using the technique suggested by Suárez-Figueroa et al. [23], the ontology's functional requirements were specified in the form of competency questions and their answers, which would help

¹For example, the requirement involving the number of years of experience was added in a later iteration, as it gave the developer the chance to explore data properties in ontologies.

identify the key elements of the domain of interest. They are asked from the point of view of the candidate seeker, containing commonly looked for attributes in people's CVs. Feedback from using the tool and further research in the area would lead to more questions being added throughout the iterations, resulting in changing requirements. Table 3.1 illustrates the final list of the questions for the ontology.

CQ1	What is the person's name?	Jack Jones; Jessica May
CQ2	What is the person's nationality?	British; Romanian;
CQ3	What is the person's address?	1 Brownslow Walk Manchester, 3 Wadeson Road Stockport
CQ4	What is the person's date of birth?	12/03/1978; 15/01/1968
CQ5	What is the person's contact number?	07827693461; 07782341213
CQ6	What skills does the person have?	Excel; Programming; English
CQ7	What is the competency level of their skill?	Beginner; Intermediate; Advanced
CQ8	What proof do they have for the mentioned skill?	IELTS; Transcript; Reference
CQ9	What kind of experience do they have using the skill?	Professional Experience; Academic Experience
CQ10	How many years of experience do they have using the skill?	3 years; 1 year; 10 years;

Table 3.1: Functional Ontology Requirements: Groups of Competency Questions

Non-Functional Requirements

The non-functional ontology requirements follow properties specified in Stellman and Greene's check list [22], as illustrated in Table 3.2.

Flexibility and Reusability	The ontology should be extensible to other skills areas (apart from IT) should it be necessary.
Portability	The ontology should work correctly on any tool that supports OWL 2.
Performance	Querying the ontology should have a fast response time. ²
Usability	The class hierarchy should be simple and predictable.
Reliability	The ontology should be consistent.
Scalability	Querying the ontology should not take significantly longer when the number of assertions increases. ³

Table 3.2: Non-Functional Ontology Requirements

3.3.2 Software Requirements

Functional Requirements

The application's functional requirements were formulated according to input and output needs.

In terms of *input*, the requirements can be summarised to:

- FR1. The application must allow a selection and querying of multiple skills.
- FR2. The application must differentiate between compulsory and desirable skills.
- FR3. The application must display the existing skills in the ontology to the user as suggestions.
- FR4. The user must be able to request that the candidates have proof for their skills.
- FR5. The application must suggest the types of proof available for a given skill.

²This would depend on the reasoner used.

³This will be evaluated in Chapter 6.

FR6. The user must be able to request that the candidates have a certain experience using the skill.

FR7. The user must be able to request that the candidates have a certain level of expertise in using the skill.

As per the *output*, the following functional requirements were considered:

FR8. The application must query the ontology to retrieve matches to the user's request.

FR9. The application must return only candidates that match the compulsory requirements (when these exist).

FR10. When no compulsory skills are specified, the application must return all the candidates that fulfil at least one of the given desirable skills.

FR11. The application must rank the results according to how well the candidates fulfil the request.

Non-Functional Requirements

As the ontology is linked to the software application and it was also the main focus of the project, the non-functional requirements of the tool very much rely on the fulfilment of the ones for the ontology (Table 3.2). We aimed for the tool to be robust, flexible and reusable, in line with the extensibility of the ontology. The performance and scalability of the application were under thorough scrutiny, as analysed in one of the following chapters (Section 6.2).

3.4 Architecture

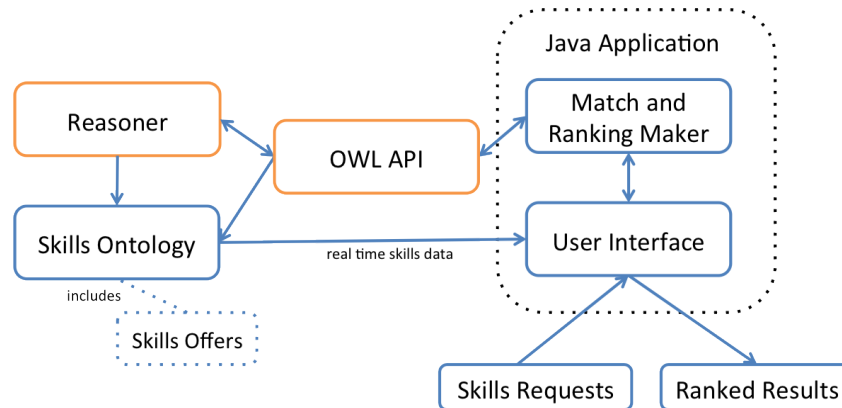


Figure 3.1: Architecture⁴

Figure 3.1 illustrates the main components of the system and how these interact. The user makes a skills request via the user interface, which feeds into the skills ontology to display real time skills data. The matchmaker of the Java application picks up the request and uses the OWL API (together with a reasoner) to retrieve matches from the ontology, which is where

⁴Blue is used for self-built components, while orange is used for imported ones.

the offer lies. The candidate matches are ranked in the Java application according to how well they fulfil the request and the ranked results are displayed through the Java user interface. Each component is described in more depth in the following sections.

3.5 Ontology Design

3.5.1 Overview

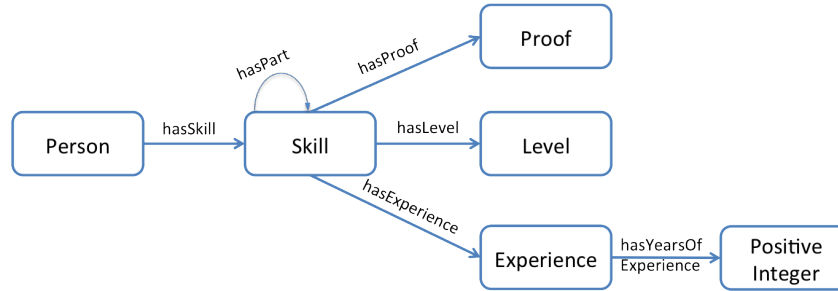


Figure 3.2: Ontology Design (TBox) Overview

Figure 3.2 gives a high level view of the design of the skills ontology, which closely follows the functional requirements and the scope of the project, as previously listed in Section 3.3.1.

The `Person` and `Skill` classes were easily identified as the key elements of the requirements. Further analysis into the domain of interest (job adverts, CVs, etc.), particularly into sources of skills, assessment and standards, led to refining requirements throughout the project’s lifecycle and ultimately adding `Proof`, `Level` and `Experience` as classes in the ontology. This further proves that an iterative approach to the development of our project was an appropriate decision.

As per Table 3.3, each of the illustrated classes has numerous subclasses, who may have further subclasses of their own. Not surprisingly, `Skill` is the class with the largest number of descendants, broken down into different types of IT skills, such as `Programming`, `Scripting`, `WebDesign`, etc., each with their own subclasses. The clear hierarchy (fully represented in Appendix A) makes it easy for anyone to build on the ontology, by placing the new skill under the right superclasses.⁵

Superclass	No of Subclasses	Example of Subclasses	Object and Data Properties
Skill	83	ProgrammingSkill, English, Java	hasProof, hasExperience, hasLevel, hasPart
Proof	21	Certificate, Transcript, IELTS	-
Level	3	Beginner, Advanced	-
Experience	1	Professional Experience	hasYearsOfExperience
Person	-	-	name, nationality, address, contactNo, dateOfBirth; hasSkill

Table 3.3: Subclasses and Properties in the Skills Ontology

⁵Notice the use of plural in *superclasses*, as a skill can be a subclass of more classes; e.g.: Java is both a `ObjectOrientedProgramming` skill and also a `ProceduralProgramming` skill, so it would have at least these two superclasses.

3.5.2 Representing the Offer

Realistically, people's skills are represented in separate text files, which would need to be described in OWL formatting. Ideally, this conversion would be done automatically, to then be imported in an ontology (see Figure 3.3). Nonetheless, this was not feasible for the current project, given its scope, requirements and time constraints.

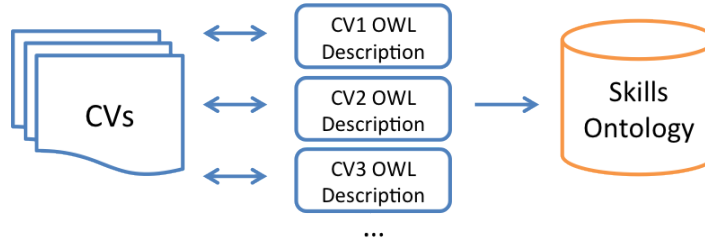


Figure 3.3: Ideal Offer Representation

As an alternative, we have decided to represent individuals and their skills as assertions in the ABox of the skills ontology [10]. The offer is therefore part of the ontology (hold in separate files and imported into the class-level ontology). Figure 3.4 gives a brief example of how Jessica, who is advanced in Java and has a transcript as proof for the skill, is represented in the ABox of the ontology.

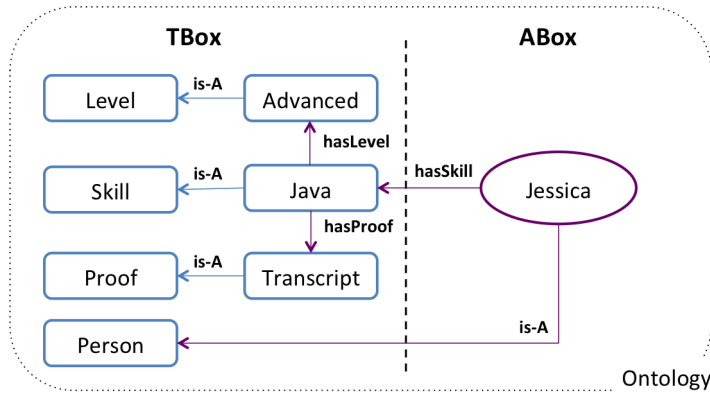


Figure 3.4: Representing the Offer - Example

3.5.3 Representing the Request

The request is built as a conjunctive query in the Java application (in the form of an OWL class expression), using the skills and further properties selected in the user interface (Figure 3.5).



Figure 3.5: Representing the Request: A simplified example

3.6 Java Application

3.6.1 Overview

We found that the most appropriate application programming interface for our tool was the OWL API for Java and hence we used Java to build the application. The two other options we had were Jena API and OWLLink. The Jena API does not support standard reasoners - which we needed for inferring properties that were not explicitly modelled, thus it was not a good match. OWLLink is simply an extension of the OWL API, whose further capabilities are around remote reasoners, outside of the scope of our application.

In designing the Java application, we followed the model-view separation principle, avoiding to include application logic in the user interface object [17]. Three main classes emerged (Figure 3.6):

- a GUI class;
- a Description Logics Query Manager class to handle all the methods connecting to the ontology (via the OWL API).
- a Controller class to manage the input from the GUI class and process it, by querying the ontology through the Query Manager and feeding back the output to the GUI class.

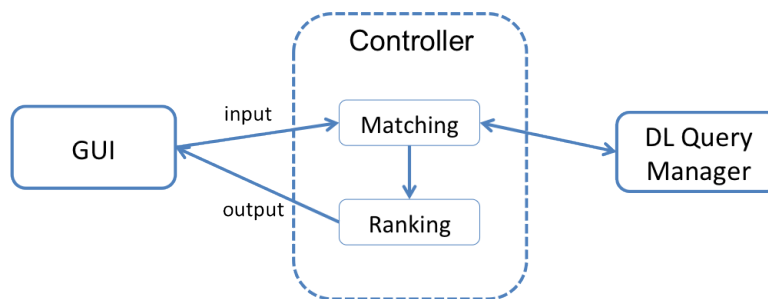


Figure 3.6: Application Design

This design helped fulfil flexibility and reusability requirements, as the separation of concerns makes it easy for a developer to perhaps build a new GUI and link it to the application, or reuse the existing one for a different matching approach.

3.6.2 User Interface Design

Despite the user interface not being a key focus, we strived for it to be intuitive and user-friendly, whilst also closely adhering to the input and output functionality requirements specified earlier (Section 3.3.2). Figure 3.7 illustrates an early wireframe of the interface layout. We aimed to include components that were simple and familiar to the user, consistent to the GUIs that would be used daily (based on the *Star User Interface* [21] principles of *familiarity*, *simplicity*, *consistency* and *what you see is what you get*).

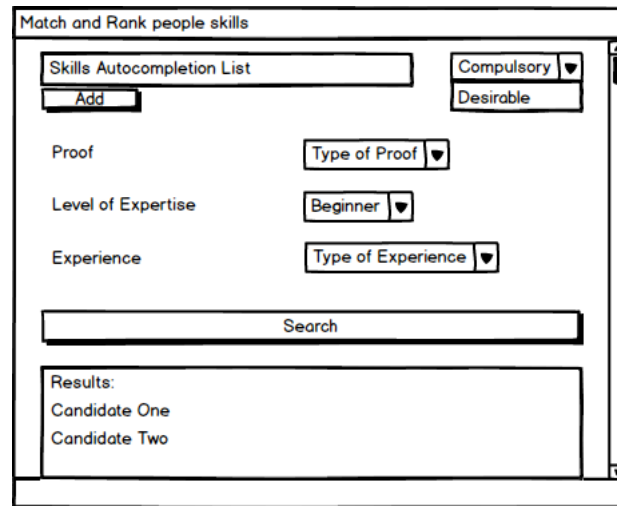


Figure 3.7: User Interface Wireframe

3.7 Summary

This chapter introduced our iterative development approach, aiming to refine the requirements periodically for a better end-product. This would ultimately affect the design of the main components - the skills ontology and the Java application - which were also illustrated and detailed. The functional and non-functional requirements described earlier were key objectives for the design aspect, but also for implementation, as shown in the next section.

Chapter 4

Implementation

4.1 Modelling the Ontology

4.1.1 Overview

In modelling the ontology, we used what McGuinness and Noy in [19] call a *combination development process*, involving both top-down and bottom-up definition of classes. This involves starting with first identifying noticeable classes and then either generalising them or specialising them accordingly. For example, we may have identified top-level classes such as `ProgrammingSkills` and bottom-level classes such as `Java` or `C`. To relate these, we had to find middle-level classes, such as `ObjectOrientedProgramming` and `ProceduralProgramming`, which we would later specialise further, by adding new classes such as `Ruby`, `Scala` or `C++`. The main superclasses are illustrated in Figure 4.1, with a full hierarchy available in Appendix A.



Figure 4.1: Core Ontology Superclasses (Protégé Screenshot)

4.1.2 Defining Classes and Their Properties

The classes identified for the ontology were created manually using Protégé 4.2.0 (presented in Section 2.3.4). A class could be either added as a superclass, subclass or sibling of a current class, only needing to give it a name to start with. To complete the definition of the class, equivalent classes, members of the class and disjoint classes would be added to its description.

Annotations

Annotations were a key feature used in describing the `Proof` sub-classes. They would be utilised to associate certain types of `Proof` to their corresponding skill. This would be later identified by the OWL API and suggested accordingly to the user in the GUI. For example, `IELTS` and `UETESOL` (certificates proving a person's competency in using the English language) would have `English` as an annotation; when the user would query `English` as a skill, these specific types of proof would show up as options due to their annotation.

Object Properties

In order to express relationships between the defined classes, object properties need to be defined (recall Figure 3.2). These enable the user to create the complex queries that we aim for in the functional requirements. Using these queries to structure our skills description is what makes our semantic web approach more powerful than a simple text based one. The table below provides an example of how object properties are used to distinguish simple skills descriptions that would be problematic in text-based retrieval, due to their word similarity.

Skill Natural Language Description	OWL Class Expression
Advanced in Java and intermediate C skills	<code>hasSkill some (Java and hasLevel some Advanced) and hasSkill some (C and hasLevel some Intermediate)</code>
Intermediate Java skills and advanced in using C	<code>hasSkill some (Java and hasLevel some Intermediate) and hasSkill some (C and hasLevel some Advanced)</code>

Table 4.1: Distinguishing between Skills Descriptions using Object Properties

Defining an object property involves establishing a domain and a range for the property, which can be easily added in Protégé. The table below describes the domain and ranges for the skills ontology's object properties. Not surprisingly, most of these relate to the `Skills` class, linking it to the `Proof`, `Level` and `Experience` classes.

Object Property	Domain	Range
<code>hasSkill</code>	<code>Person</code>	<code>Skill</code>
<code>hasProof</code>	<code>Skill</code>	<code>Proof</code>
<code>hasLevel</code>	<code>Skill</code>	<code>Level</code>
<code>hasExperience</code>	<code>Skill</code>	<code>Experience</code>
<code>hasPart</code>	<code>Skill</code>	<code>Skill</code>

Table 4.2: Object Properties in the Ontology - Domains and Ranges

Notice that the `hasPart` object property links a type of `Skill` class to another `Skill` class. `hasPart` is also included in the following sub-property, linking it to `hasSkill`:

`hasSkill o hasPart SubProperty of hasSkill`

The `hasPart` property allows breaking down a skill in different parts (also skills on their own), whilst the sub-property tells the reasoner that a person having the more general skill also has competencies in the parts that make up the skill (but not the other way around). This is an important property to link skills where a superclass-subclass type of relationship would not be appropriate.

For example, `Powerpoint` could not be a `MicrosoftOffice` subclass, as this would imply that a person who knows `PowerPoint` also knows `MicrosoftOffice`, which may not be the case (one may not be familiar with the other packages: `Access`, `Word` etc.). Instead, the `hasPart` property under `MicrosoftOffice` (Figure 4.2) tells the reasoner that any individual that knows `MicrosoftOffice` also knows any of the `PowerPoint`, `Access`, `Word` or `Excel`, but in order for a person to have `MicrosoftOffice` as a skill, they would need to be competent in each of its mentioned parts.

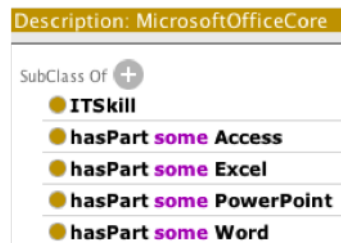


Figure 4.2: Example of using `hasPart` (Protégé Screenshot)

Data Properties

The description of classes in the skills ontology is further enriched through the use of data properties (recall Table 3.3). Here, the domain is represented by the corresponding class, whilst the range is the data type for the property (e.g. `String`, `Integer`). The most significant data property used in our skills ontology is `hasYearsOfExperience`,¹ which enables us to specify the time length of the experience of a `Person` using a skill and, later in the application, it allows the user to specify preferences in terms of this property.

4.1.3 Defining Individuals

After all the classes and properties have been defined, individuals can be created in the ontology based on these. As mentioned earlier, candidates are represented as individual assertions that are imported in the Abox of the ontology, also with the help of Protégé.

¹For `hasYearsOfExperience`, the domain is `Experience` and the range is `nonNegativeInteger`

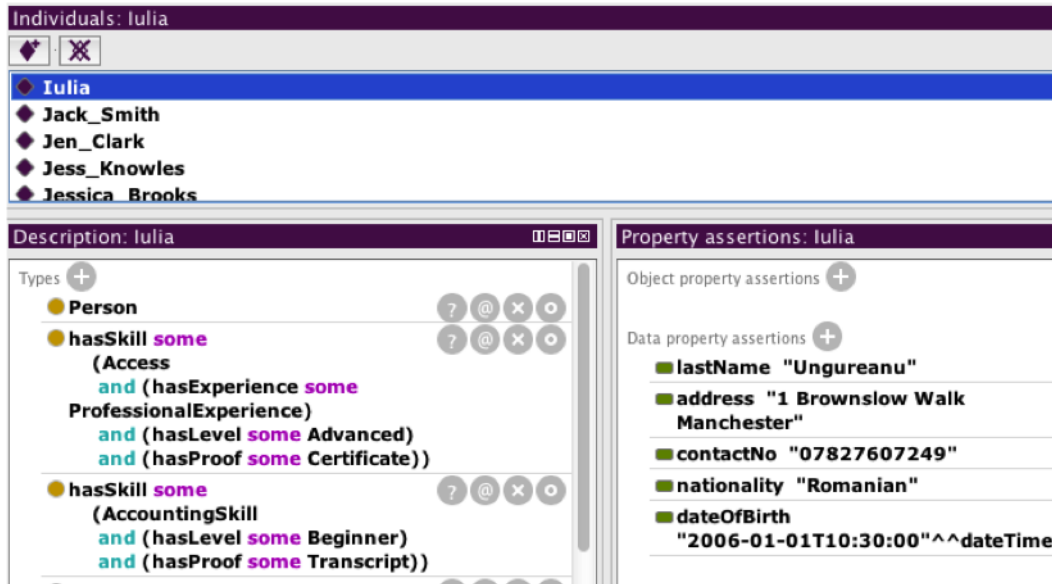


Figure 4.3: Representing Individuals in Protégé

Figure 4.3 illustrates how *Iulia* is represented in Protégé. To specify the skills, proof, level and experience for an individual, class expressions are inserted in the *Description* tab, in the Manchester OWL Syntax. Having specified domains and ranges for the object properties, Protégé checks each class expression such that invalid assertions are not made. Further details of the individual can be added in the *Property assertions* tab, using the appropriate data types.

4.2 Java Application Implementation

4.2.1 Building the User Interface

We used Java to build the application's interface, which allowed us to dynamically retrieve and display data from the skills ontology via the OWL API. Should the ontology change, these changes would be automatically displayed in the GUI.

To enhance the user experience and also avoid getting invalid data, we used a `JSuggestField` for the input of skills (Figure 4.4). This is a type of autocompletion box that gives the user skills suggestions as he or she types in. The suggestions are refreshed and displayed every time the application is ran, using the OWL API to retrieve the `Skill` sub-classes from the ontology. As the *Add more skills* button is pressed, a new `JSuggestField` is created for the new skill, while all the other components are simply restored to default, in order to avoid cluttering the GUI.²

Another element that is driven by the ontology is the *Type of Proof* drop-down list, which picks up and displays the specific proof types for the current skill (Figure 4.5). This is once again achieved through the use of the OWL API, with a method looping through the annotations of the `Proof` sub-classes and finding the corresponding ones (recall annotations from

²Information from the previously skill is stored first, so nothing is lost.

Section 4.1.2).

Figure 4.4: Skills Autocompletion Example

Figure 4.5: Proof Customisation³

4.2.2 Matching and Ranking Results

Matching Algorithm

As mentioned earlier (recall the functional requirements presented in Section 3.3.2), we have decided from the start that the results of our matchmaker would not consider candidates that do not match the given compulsory skills. This provided an opportunity for creating an efficient algorithm, which would only look for desirable skills in the candidates that match the compulsory skills, rather than all the individuals in the ontology.⁴ This considerably reduces the candidate pool to a manageable size.

The decision determined the matching algorithm to be split in two main steps:

1. Building a conjunctive OWL class expression corresponding to the compulsory skills request;
2. Retrieving the compulsory skills matches and checking them for desirable skills
 - If no compulsory skills were requested, retrieve all individuals in the ontology.

As the user presses the *Search* button, the *Controller* class picks up the input from all the fields and starts building the DL queries in the Manchester OWL Syntax (*Step 1*).

A query for the compulsory skills is built first to retrieve the definite candidates. The query is constructed incrementally as a conjunction of queries, each subquery belonging to one skill (recall Figure 3.5). A parser in the OWL API is used to check whether the query is correct in the Manchester OWL Syntax, before the query is ran. The candidates that meet the compulsory requirements are then retrieved. Desirable queries are afterwards built to verify each of the conditions of the ranking approach (*Step 2*).

³Notice the Italian-specific types of proof: *CLIDA* and *CELI*

⁴If the user only specifies desirable skills, the entire individual pool will be queried.

Ranking Approach and Heuristic

Our approach to ranking the candidates involves creating a points system. Each candidate⁵ starts with a score of 0, and is then awarded an extra a point each for fulfilling any of the following conditions:

- the candidate has the desirable skill;
- the candidate has the desirable skill and the desirable proof (if requested);
- the candidate has the desirable skill and the desirable level of expertise (if requested);
- the candidate has the desirable skill and the desirable type of experience (if requested);
- the candidate has the desirable skill and the desirable length of experience (if requested).

These conditions are contracted versions of the complete query representing the desirable skill request (recall *concept contraction* from Section 2.4). Using them helps achieve flexibility of queries and descriptive results.

After all the desirable skills have been reviewed, the candidates are ranked in descending order according to their accumulated points, with equal ranks for equal number of points.

This approach is thus based on the assumption that each extra preference is of the same importance (weighs the same) for the user. Nonetheless, the application has been designed such that weights (points) for each type of preference can be easily changed, should it be necessary.

4.3 Challenges

Implementing the described approaches was not an easy task, with various challenges arising across iterations. Some of the issues were related to the technologies used:

- incompatibility of Protégé with the latest version of Java;
- OWL API integrated reasoner (Fact++) not handling complex queries;
- data types incompatibilities from Protégé to the OWL API;
- reasoner performance issues.⁶

These were successfully overcome by finding working alternatives and solutions, yet caused delays on the development efforts.

Changing requirements were also challenging, resulting in modifications in both design and implementation. For example, *Experience* was initially represented as a type of *Proof*

⁵As just mentioned, this is a selected pool of individuals which fulfil the compulsory skill requirements. If no compulsory skills are requested, all the individuals in the ontology are added to the score table with an initial score of 0.

⁶These are tackled in more depth in Chapter 6.

rather than a separate requirement. The decision to give the class a more significant role in the application and also add the ability to query for the length of the experience resulted in changes in:

- the ontology's hierarchy and object and data properties;
- the structure of the DL queries built in the Java application;
- the ranking of candidates.

4.4 Summary

The chapter reviewed the main approaches in building the ontology and the Java application, both parts being simultaneously and incrementally developed throughout the iterations, around the functional and non-functional requirements.

A variety of Protégé features were used in building the ontology, in order to make the latter highly expressive: data properties, object properties, annotations and individual assertions. These would be queried and retrieved in the Java application with the use of the OWL API. Appendix B provides a list of the main OWL API methods used to manipulate the ontology.

The decisions and assumptions that shaped the matching and ranking algorithm were also explained.

Challenges relating to DL technologies and changing requirements were faced and overcome throughout the project lifecycle.

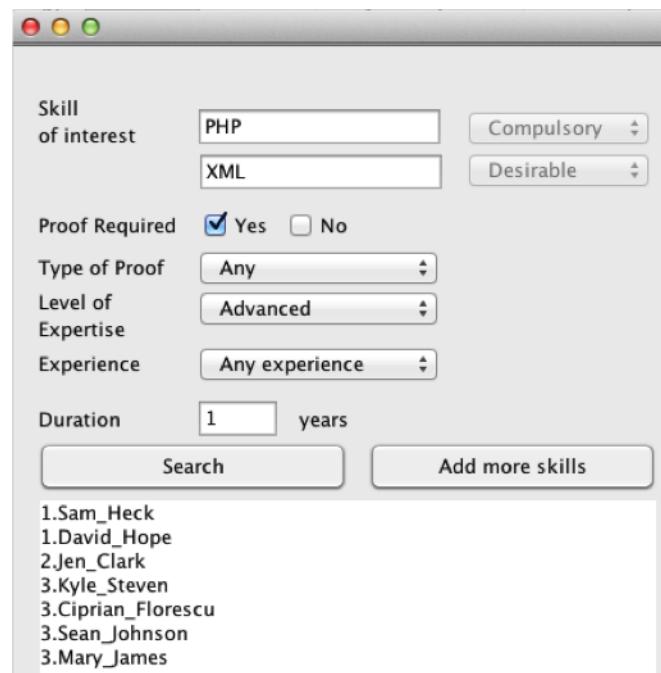
Chapter 5

Results

5.1 Overview

This chapter will demonstrate the achievements of the project, by illustrating the accomplishment of the functional and non-functional requirements through the final product.

5.2 Functional Requirements



The screenshot displays a user interface window with a title bar containing three colored buttons (red, yellow, green). The main content area is a light gray panel with the following elements:

- Skill of interest:** Two text input fields. The first contains "PHP" and is followed by a "Compulsory" dropdown menu. The second contains "XML" and is followed by a "Desirable" dropdown menu.
- Proof Required:** A label followed by a checked "Yes" radio button and an unchecked "No" radio button.
- Type of Proof:** A dropdown menu with "Any" selected.
- Level of Expertise:** A dropdown menu with "Advanced" selected.
- Experience:** A dropdown menu with "Any experience" selected.
- Duration:** A text input field containing "1" followed by the word "years".
- Buttons:** Two buttons at the bottom: "Search" and "Add more skills".
- Results List:** A scrollable list below the buttons containing the following entries:
 - 1.Sam_Heck
 - 1.David_Hope
 - 2.Jen_Clark
 - 3.Kyle_Steven
 - 3.Ciprian_Florescu
 - 3.Sean_Johnson
 - 3.Mary_James

Figure 5.1: User Interface - Final Version

Through the User Interface (Figure 5.1), the application allows the user to specify all the skill preferences mentioned in the functional requirements (recall Section 3.3.2): multiple skills (FR1), compulsory or desirable (FR2), with or without existing proof (FR4), certain levels of expertise (FR7), experience and a specific number of years of experience (FR6).

After the user presses the *Search* button, the matching candidates from the ontology are retrieved and displayed (FR8, FR9, FR10), ranked according to what they offer in terms of the skills request (FR11). The user can then make a business decision according to the results, such as calling the top 2 candidates for an interview.

5.2.1 System Walkthrough

To emphasise the accomplishment of the functional requirements referenced above, this section will briefly present how a user would interact with the application interface (represented earlier in Figure 5.1) and the results he or she would observe.

- The user would start inserting the requirements in the GUI one skill at a time, with all the fields being optional apart from the skill name and type of skill (compulsory or desirable). Leaving all other fields as by default would consequently result in the application returning results to queries written in the form `hasSkill some <Skill>`.
- Nonetheless, after selecting a skill from the autosuggestion list (recall Figure 4.4), the user may want to require a proof for the skill. By ticking the *Yes* box for *Proof Required*, the *Type of Proof* drop-down list is made available, providing options from the ontology that are specific to the current skill (recall Figure 4.5). The proof details would be added to the class expression being built in the background (and `hasProof some <Proof>`).
- Furthermore, the user can specify the level of expertise he or she is looking for in a candidate (Figure 5.2 below), adding and `hasLevel some <CompetencyLevel>` to the current class expression.
- Lastly, the user may be interested in candidates with experience in using the skill, so a type of experience may be specified, along with the required duration in years (Figure 5.3 below) - adding and `hasExperience some (<Experience> and hasYearsOfExperience value <X>)` to the class expression.

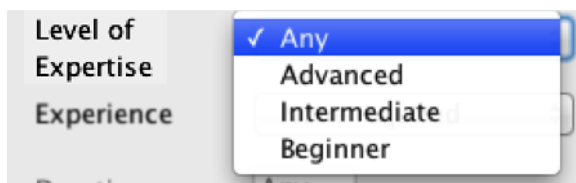


Figure 5.2: Selecting Expertise Level

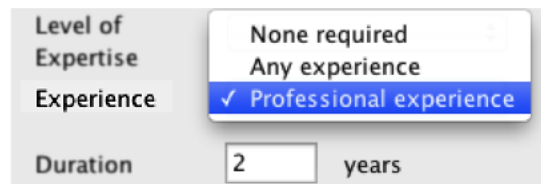


Figure 5.3: Selecting the Type of Experience and Duration

- The user can continue adding new skills by pressing the corresponding button in the GUI, or press *Search* to see the results of the current query.

Figure 5.4: Adding More Skills

- If the user decides to add a new skill, as soon as the *Add more skills* button is pressed, a separate autosuggestion box for the new skill appears (Figure 5.4 above), while all other components are restored to default, awaiting new input for the new skill. The user can then choose properties different to the previous skill, if any: other types of proof (such as `hasProof some ECDL1` or `hasProof some Reference`), level or experience. The class expression built for the new skill is added to the one created for the previous skill, in a conjunctive manner (`<Expression1> AND <Expression2>`).
- The results display the matches for the compulsory skills query, ranked according to their offering in terms of desirable skills (as per Figure 5.1). If no compulsory skills are specified, the application displays all the candidates that have at least one of the desirable skills. If only compulsory skills are requested, the corresponding candidates retrieved will all have the same rank.²

5.3 Non-functional Requirements

The achievement of non-functional requirements of the skills matchmaker and the ontology (recall Table 3.2) is less obvious to the user, but equally important to the functional requirements.

Flexibility and **reusability** have been achieved through the design of the ontology as an easily extensible representation of skills, which could be re-used in different projects. The Java application has also been built with these requirements in mind, as the separation of application logic from the GUI allows both the interface and the matchmaker to be re-used on their own, if necessary. Moreover, differentiating between compulsory and desirable skills and properties provides the user with flexibility in querying the ontology, whilst also delivering expressive and meaningful results.

Using the latest version of the OWL API and developing using Java guaranteed the **portability** of the application in any environment and of the ontology to any tool supporting OWL

¹European Computer Driving Licence

²All the candidates will have a rank of 1, as there are no desirable skills to distinguish between them.

2.

The clear, straightforward design of the ontology (recall Figure 4.1) achieves **usability**, by making it easy for any ontology developer to understand, learn, use the ontology and also build on it further. The application's user interface was also built to enhance the user experience, with autosuggestion boxes, familiar components and consistency throughout the interaction.

In terms of **performance** and **scalability**, empirical tests (expanded on in the next chapter) proved that the application performs relatively well (in a matter of seconds with Fact++) when the number of candidates is increased up to 1000 and for different number and complexities of queries.

The application has proven **reliability**, with consistency between the ontology and the user interface, as the latter is automatically updated when changes in the ontology occur. The consistency in the structure of class expressions from the application to the ontology was also at the core of the project's implementation, avoiding eventual failures due to incorrect syntax or semantic incompatibilities.

Chapter 6

Testing and Evaluation

6.1 Overview

Testing the application and the ontology was crucial in evaluating the end-product and the approach taken to solve the given problem. Functionality testing was performed iteratively through the project lifecycle, along with development, to tackle changing requirements and ensure good progress. Significant performance and scalability testing was undertaken post core development, which revealed external causes of delay and instability, such as the reasoners used.

This chapter will give an overview of each type of testing performed, as well as an in-depth case study, where our semantic web approach is compared to a text-based one for evaluation purposes.

6.2 Manual Testing

The *DL Query* tab in Protégé was a simple and effective way of manually testing the quality of the ontology all throughout the project lifecycle, in particular the way it models skills. It helped check whether subclasses and superclasses were correctly defined, but also if the expected individuals are retrieved when running various queries. It was also extremely useful in the developer's learning of how to build and structure class expressions to achieve the wanted results.

The feature was also further used to test the candidate results of the Java application, by checking the corresponding DL query for the user's request in Protégé and comparing the results. For example, if a user requested candidates that knew *English* and could prove it with a *Certificate*, running the corresponding query in the *DL Query* tab (Figure 6.1) would retrieve the correct candidates and these could be checked with the Java application's output.

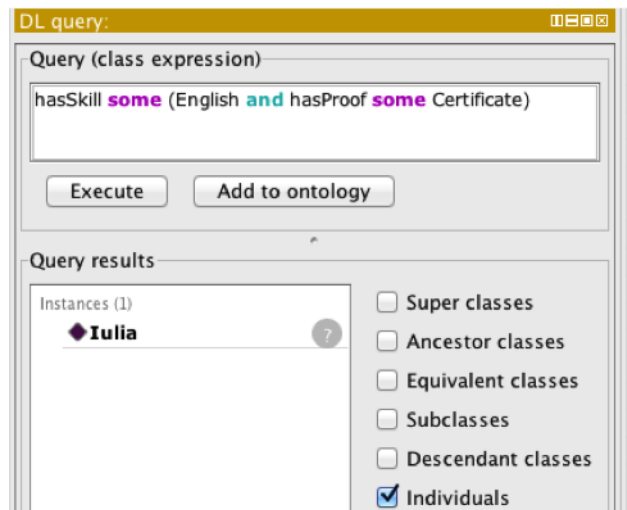


Figure 6.1: Manual Testing with the DL Query Tab (Protégé Screenshot)

6.3 Unit Testing

JUnit Tests were used to automatically test the functionality of the core features behind the Java application. They would be built as soon as a feature was implemented and they would be ran at each iteration, testing and ensuring that new changes and additions to the application have not affected existing working code. The main JUnit Tests implemented were the following:

- *Skill sort unit test*: this would test the method sorting the given skills, with compulsory skills first and desirable last.
- *Candidate sort unit test*: after the all the candidates were given scores according to how well they matched the request, the map storing their accumulated points needed sorting in descending order, in preparation for the ranking.¹ A unit test would check the method covering this functionality.
- *Manipulate class expression test*: this would review whether the class expressions were structured correctly, by checking their results. This test was essential across iterations, unveiling various bugs.
- *Ranking test*: this would review whether the ranking of candidates was performed correctly. The test helped reveal an important case we had overlooked: we were not tackling ranking when there were no desirable skills required (only compulsory skills). This resulted in the application displaying null values next to the candidate's name. Having spotted the issue with the unit test, it was changed to display all candidates with an equal rank of 1.

¹Not a straightforward implementation, as candidates and their scores were stored in a hashmap - sorting required the use of a Comparator.

6.4 Performance and Scalability Testing

6.4.1 Motivation

We found it was important to undertake performance and scalability testing, first of all as a proof of concept for our approach to matchmaking. Despite significant progress in the area, with highly optimised inference algorithms for DL being developed, reasoning can still be computationally expensive [16]. This is especially claimed for instance retrieval from large ABoxes [3], a case highly relevant to our application, which may hold large number of skills assertions, corresponding to people's CVs.

6.4.2 Experiments and Results

All the experiments were executed on a machine running Mac OS X 10.8.3 and Java 7, on an Intel Core i7 CPU at 2.8GHz. Version 3.4.3 of the OWL API was used to load the ontology and interface with the reasoners. The reasoners used are FaCT++ 1.6.2 and HermiT 1.3.7. The queries ran in each experiment can be found in Appendix C.

The starting point for testing the performance of the application was reviewing its run time when the number of candidates in the ontology would increase. The test involved running a single desirable skill query (`hasSkill some Access and has Proof some Proof`), which would imply, by design, that the application would be looking at the entire pool of candidates, rather than a selected pool according to compulsory skills.

Initial tests with the application using HermiT as reasoner were slightly worrying, as the run time would quickly escalate the more individuals were included. Nonetheless, later tests with Fact++ had much better results, proving that it was the reasoner causing a slow performance in the earlier tests, rather than the application logic. The graph below illustrates the results of these experiments.

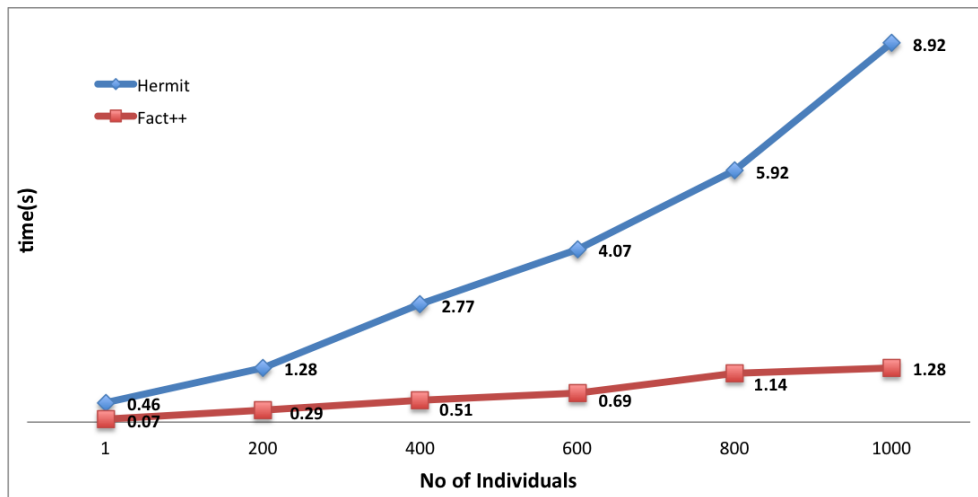


Figure 6.2: Application Run Time (in seconds) ²

²The data was generated by cloning existing individuals in the ontology, using the `<SameIndividual>` OWL property.

After testing how the application handles larger sets of candidates,³ we experimented with changes in the number of requests coming from the user. Based on the assumption that a user would not be stating preferences in more than 10-15 skills, we first tested how the application would perform when the number of desirable skills increased. Since the application is built to review the desirable skills one by one (recall the ranking approach in Section 4.2.2), we expected to see the run time to double as the number of the desirable requests doubled. Our expectations were met when testing with *Hermit*, as per the figure below:

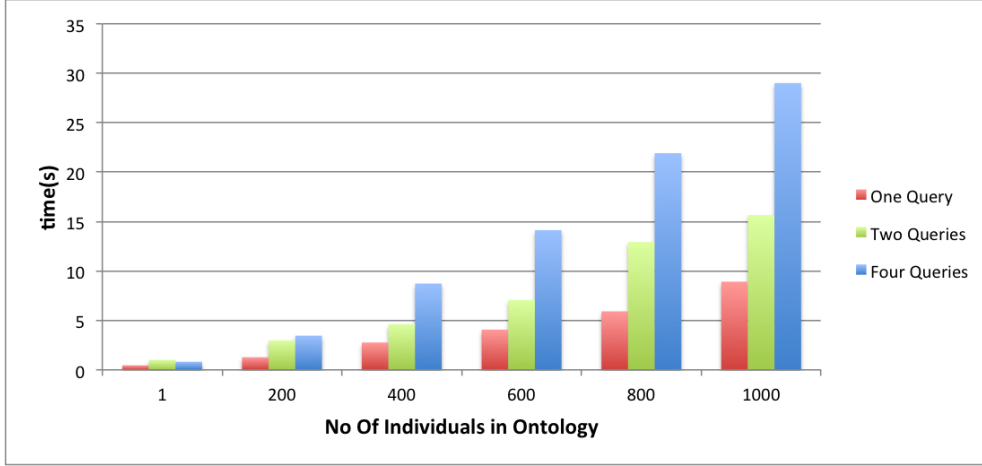


Figure 6.3: Application Run Time by No. of Desirable Queries - with *Hermit*

However, *Fact++* proved to be much faster, with only a small increase in run time as the desirable skills requests increased:

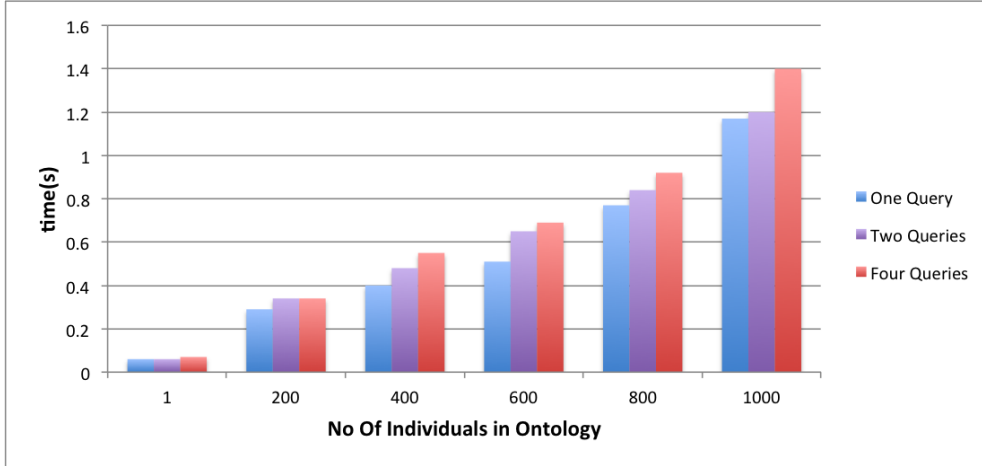


Figure 6.4: Application Run Time by No. of Desirable Queries - with *Fact++*

Now knowing that a change in the number of desirable queries does not significantly affect the performance of the application (when using *Fact++*), we can further review the run times for different compulsory skills requests in terms of their representation in the ABox. Since our matching algorithm first retrieves the candidates that comply to the compulsory skills, and

³Caveat: as per note 2, these test candidates were produced artificially, through the cloning of a series of individuals. Clever reasoners can pick up on the cloning and perform better than others simply on this basis.

only then breaks down the desirable skills and checks for matches with the selected candidates, the next step is to analyse how performance changes when this selection (as % of the entire candidate pool) increases.⁴

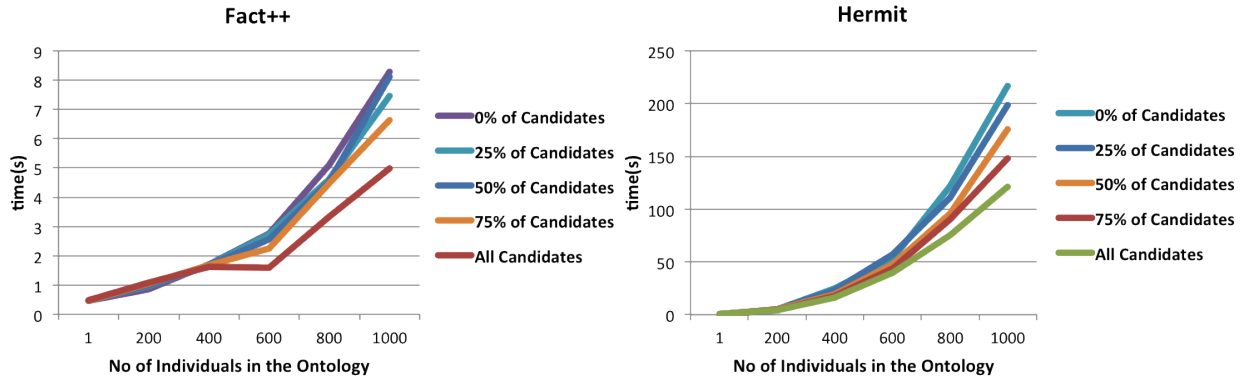


Figure 6.5: Run Time By Compulsory Candidate Pool Coverage ⁵

For smaller ontologies (up to 400 - 600 individuals), the runtime is mostly not affected by the selectivity of the query. However, for larger ABoxes, the results differ and are surprising at a first glance: it seems that the application performs better when there is a higher coverage of candidates that fulfil the compulsory skills. Nonetheless, it is reasonable that verifying that all individuals are instances of a general concept (the desirable skill which all candidates ontology fulfil) is much faster (easier for the reasoner) than verifying that they are not instances of a more specific one (the compulsory skill).⁶ We have run further tests to confirm this hypothesis, and when testing with no skills for non-matching candidates⁷, therefore reducing the run time to (almost) only reviewing the candidates fulfilling the compulsory skills, the results are as expected for our approach: the application runs faster the smaller the compulsory candidate pool coverage.

6.5 Case Study

6.5.1 Overview

This section will review a case study of using our skills matchmaking tool to retrieve candidates for a job advertisement and compare its results to the ones of a text-based retrieval application (DocFetcher⁸). The aim is to understand both advantages and disadvantages of using our tool and ontologies in general to represent and match people skills over a text matching approach.

⁴For example, for a pool of 200 of individuals, we would be looking at how the performance of the application changes when none of them fulfils the compulsory skills, to when 50 of them match the compulsory skills, to 100 of them, 150 of them and the entire pool.

⁵Notice the slightly higher run-times for these experiments; unlike earlier tests where we used the `<SameIndividual>` property, making it easier for the reasoner, we cloned different, separate individuals this time.

⁶Determining that x is not an instance of A is, in general, more difficult: the reasoner has to verify if the union $\{x:\text{not}(A)\}$ is satisfiable. If it is satisfiable, then x is not an instance of A , otherwise it is.

⁷We created the non-matching candidates such that they have no skills that would need to be evaluated by the reasoner, to avoid the earlier impact on performance and focus on solely the compulsory candidates.

⁸<http://docfetcher.sourceforge.net/en/index.html>

6.5.2 The Job Advert

We used *The University of Manchester Careers Services*⁹ website to retrieve the following job description¹⁰:

Junior Software Developer

Skills:

- Able to demonstrate coding project or thesis.
 - Some experience using C++ and/or Java in a Linux environment would be an advantage.
 - Experience in software development, as an intern/student or other - a major advantage.
 - Theoretical knowledge of development process and good engineering practice; Ability to write functional code with guidance from more experienced engineers.
 - Experience with network environment, tools and protocols, specifically the TCP/IP suite, an advantage.
 - Agile/Scrum development method understanding, an advantage.
-

Table 6.1: Junior Software Developer Advert

We will be using this description as the input for both the ontology and text-based applications and analyse their results.

6.5.3 Matching With the Ontology-based Matchmaker

The following table describes the user input for the ontology based application, transcribed from the *Junior Software Developer* advert:

Skill	Desirable/Compulsory	Proof Required	Type of Proof	Level of Expertise	Experience	Duration
ProgrammingSkill	Compulsory	No	-	Any	Any	Any
C++	Desirable	No	-	Any	Any	Any
Java	Desirable	No	-	Any	Any	Any
Linux	Desirable	No	-	Any	Any	Any
SoftwareDevelopment	Desirable	No	-	Any	Any	Any
SoftwareDevelopment	Compulsory	No	-	Any	None	-
SoftwareEngineering	Compulsory	No	-	Any	None	-
Networking	Desirable	No	-	Any	Any	Any
TCP/IP	Desirable	No	-	Any	Any	Any
Agile	Desirable	No	-	Any	None	None
Scrum	Desirable	No	-	Any	None	None

Table 6.2: User Input for the Match Maker GUI

The example emphasises the expressiveness of the class expressions in our ontology, allowing the modelling of important details, starting with desirable skills ("... would be an advantage") and ending with experience and type of experience ("... as an intern/student"); these details further help distinguish candidates, leading to descriptive rankings.

Nonetheless, inserting the requests for the job in our GUI also helps pinpoint issues with the job description itself. For example, the *Junior Software Developer* is confusing in terms of the software development request, by first stating that experience using the skill would be an advantage (hence first setting *SoftwareDevelopment* as desirable in the query), yet then clearly stating that knowledge of the software development process is mandatory (thus adding *SoftwareDevelopment* as a compulsory request as well). Our structured approach to skills

⁹<http://www.careers.manchester.ac.uk/careerslink/>

¹⁰The search included the following filters: key word *software*, *Organisation sector* = *IT*, *Occupational Area* = *Information Technology*. The first advert was selected for this case study.

may therefore draw the attention of employers to such issues and lead to better job descriptions. It would also be straightforward to extend the matchmaker and its user interface such that, when a request is entered, it checks whether there are desirable skills that are subsumed by mandatory ones - thereby supporting the user to formulate a meaningful request.

As by design, the individuals are expressed as assertions in the ABox of the ontologies. These were artificially produced to match numerous job adverts from the *Careers Services* website (including the *Junior Software Developer* advert), such that the ontology is of sufficient coverage and depth for testing.

6.5.4 Matching With DocFetcher

We have prepared a series of mini-CVs as individual text files for possible candidates for the job advert. The brief CVs are natural language descriptions of the same individuals with assertions in the skills ontology, built through straightforward transliterations of these assertions. Figure 6.6 shows an example of how Dan Hunt was represented in the ontology and then in a simple text format for DocFetcher.

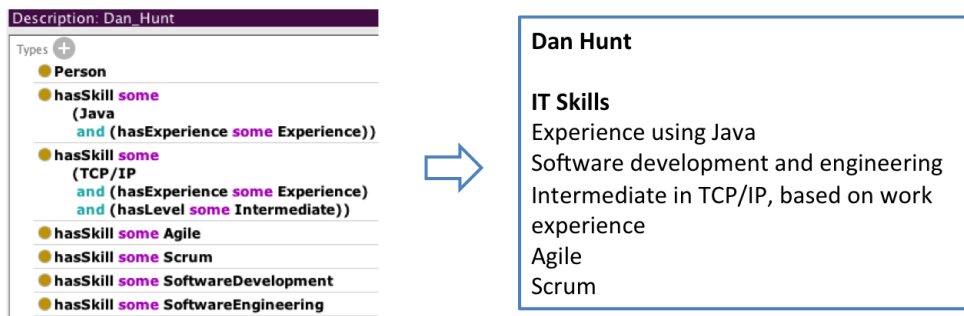


Figure 6.6: Representing CVs for Text Matching

The input for DocFetcher includes the key words of the job description illustrated in Table 6.1.

6.5.5 Comparing Results

The results of our ontology-based matchmaker are illustrated in Figure 6.7 (see Appendix D for the complete reasoning), while the results of DocFetcher are represented in Figure 6.8.

1.Dan_Hunt
2.Jack_Smith
3.Emily_Brown
4.Jessica_Brooks
5.Michelle_Rivero

Figure 6.7: Ontology Match Maker Results

Title	Score [%]
W Dan Hunt	57
W Jessica Brooks	46
W Michelle Rivero	46
W Jack Smith	45
W Emily Brown	43
W John	11
W Sam Heck	11
W David Hope	11
W Ciprian Florescu	7
W Jess Knowles	6
W Ken Dean	5
W Sean Johnson	4
W Mary James	3
W Jen Clark	2
W Uli	2
W Iulia	2
W Michael Pryor	1

Figure 6.8: DocFetcher Results

The first noticeable difference is in the number of candidates returned. Our matchmaker strictly outputs the individuals in the ontology fulfilling the compulsory skills, whilst DocFetcher returns all candidates that have one of the requested words in the job description, thus leading to a large number of false positives. The figure below illustrates the matching strings for *Sam Heck*, who is displayed as a strong candidate (with a score of 11), despite not actually fulfilling any of the conditions in the request.

Sam Heck
Advanced in using web development technologies such as AJAX, CSS, HTML, JavaScript and PHP based on
work experience
Beginner in Web Development Standards
Advanced in using XML
SQL
Project Management

Figure 6.9: DocFetcher Incorrect Match

False negatives also occur in the text-based search, mostly due to the lack of background knowledge and semantics, affecting the ranking of candidates. For instance, Emily Brown's CV specifies knowledge of DLC (Data Link Control), a type of protocol which would fulfil the "experience with network environment, tools and protocols" skill request. DocFetcher does not pick this up though, not having this kind of knowledge; the ontology matchmaker, however, does see it as a match, because DLC is modelled as a type of Protocol under Networking skills. Should this have been spotted by DocFetcher, Emily would have ranked higher in the results, as it does in our application.

6.5.6 What About Structured Documents?

Our case study focuses on a text-based search of unstructured documents, highlighting issues occurring mostly due to this lack of structure. But, as illustrated in Chapter 1, people's CVs could be flexibly structured (for instance, with XML), with skills arranged in a clear hierarchy (e.g. with parent-child relationships, use of synonymy). This would eliminate issues such as the earlier false negative example, where DLC was not a match because it was not known as a type of Protocol.

Nonetheless, ontologies take structuring further, with the properties added to hierarchies (recall Section 4.1.2) providing the means and flexibility to express and retrieve exactly what

we want with regard to skills (or any other concept). A great example is the `hasPart` object property, which solves a problem simple hierarchical structures cannot. Parent-child relationships are not always appropriate (recall Microsoft Office as a skill vs. Excel or PowerPoint, also in Section 4.1.2), so the ability to model skills as being part of a more general skill is key to both expressing them and later retrieving them correctly. Similar properties can be created in addition to sub-class relations to model any type of situation and relationship, which is the core advantage ontologies have over other structured and semi-structured formats.

6.5.7 Case Study Summary

This case study and its further discussions emphasised the advantage of using ontologies in modelling and querying skills over unstructured and structured text formats. The comparison of results illustrated how the use of ontologies can help increase both precision (by ignoring irrelevant candidates) and recall (by retrieving all relevant candidates).

Structuring skills was paramount to achieving this, by helping:

- clearly distinguish between candidates (e.g. differentiate between Jack who has 10 years of experience using Java, but only limited experience of Agile Methodologies and Jane who has 10 years of experience with Agile Methodologies, but limited knowledge of Java);
- create superclass-subclass and has-part relationships (e.g Java as a match for Object Oriented Programming, Microsoft Office as a match for Excel for having-part Excel).

OWL proved to be highly suitable for structuring skills, yet other technologies (such as XML) may also be used to (partially) achieve similar hierarchies.

Nonetheless, OWL ontologies provide means for describing further expressive properties and relationships; however, they do come with a drawback concerning the time it would take to model the ontology itself (understanding the domain of interest, agreeing on classes and relationships, etc.) and represent the request in terms of the ontology, comparing to simple text representation and retrieval.

6.6 Summary

This chapter covered the various types of testing undertaken to evaluate the implementation of our OWL ontology-based matchmaking tool.

Manual testing with Protégé helped review the accuracy and consistency of the ontology throughout the project lifecycle, as well as the results returned by the Java application.

JUnit tests were key in ensuring that working code was not affected when more functionality was added in each iteration, and also helped depict implementation issues.

The performance and scalability tests showed the ability of our application to handle increased capacities of users and variations in number of queries. It also taught a valuable lesson concerning the impact different reasoners have on performance, which can be easily disregarded.

The case study illustrated the advantages of using OWL ontologies over other technologies for structuring and describing skills, achieving higher precision and recall.

Chapter 7

Summary and Outlook

7.1 Achievements

The project successfully achieved all the requirements set out in the initial objectives:

- An extensible ontology, defining a variety of IT skills, structured in a superclass-subclass hierarchy, and semantically linked via defined properties and annotations;
- A Java matchmaking tool, which allows the user to express desirable or compulsory skills preferences, types of proof, competency level and experience required. Using the OWL API, the tool retrieves all potential candidates for the request, ranked in a most-promising order;
- A dynamic User Interface, allowing the user to express preferences for multiple skills, with a real time link to the ontology, utilised in displaying skills, proof, level and experience suggestions;
- A substantial insight in the advantages and gains of using OWL to structure skills, in comparison to other unstructured and structured technologies.

The project lifecycle involved significant development at a personal level, the immersion in the semantic web technologies leading to a steep learning curve with regard to DL, OWL ontology design, OWL reasoners and tools and the OWL API. It also provided opportunities for improvement in programming with Java, particularly concerning the GUI implementation. The thorough testing undertaken was also challenging, requiring an in depth understanding and analysis of the tool's extensibility and limitations, as well as of the factors impacting these.

The design and implementation description, the results obtained and the case study insights give the reader an overview of the effort involved in applying an OWL ontology-based approach to matchmaking, as well as the advantages gained over keyword-based methods. The key lessons to share with the reader can be summarised to:

- Building an ontology from scratch is challenging, time-consuming and tedious at times, yet it results in a re-usable, extensible framework, that allows representing the relevant dimensions of the concepts involved and linking them using suitable properties;
- The OWL API provides powerful support methods and interfaces for OWL ontologies, allowing real-time reasoning and integration with the front-end software application;

- OWL Reasoners perform differently, some better than others, thus it is worth testing an application with a variety of them;
- Other approaches to matchmaking exist and different technologies may achieve similar hierarchies to OWL, but they lack extensive semantic abilities around defining properties/relationships.

7.2 Future Work

With a UK recruitment company having shown interest in our matchmaking implementation, we are confident that our approach will be considered and used in the future, at larger scales¹ and with further features and functionalities.

Any structured/semi-structured approach will require offers to be represented according to this structure's scheme, which increases costs, yet these may be justified by improved precision and/or recall and can also be supported by relevant tools (such as Protégé in our case).

In terms of design, our OWL ontology could be easily extended to include even more (IT) skills and skill categories, as well as more expressive relationships to achieve better description and retrieval. The Java GUI application was not a main focus, so development could be made to enhance the user experience, particularly around allowing the user to make changes to previously requested skills, an option which is currently not available.

Some other ideas we were considering for future implementations involve particular features such as:

- allowing for partial skills as an option or counteroffer;
- coupling of skills (e.g experience with Java in a Linux environment);
- allowing the user to add weights to various skills, stressing the importance of some skills over others.

¹As per the test results in Chapter 6, the application can handle larger individual or query capacities.

References

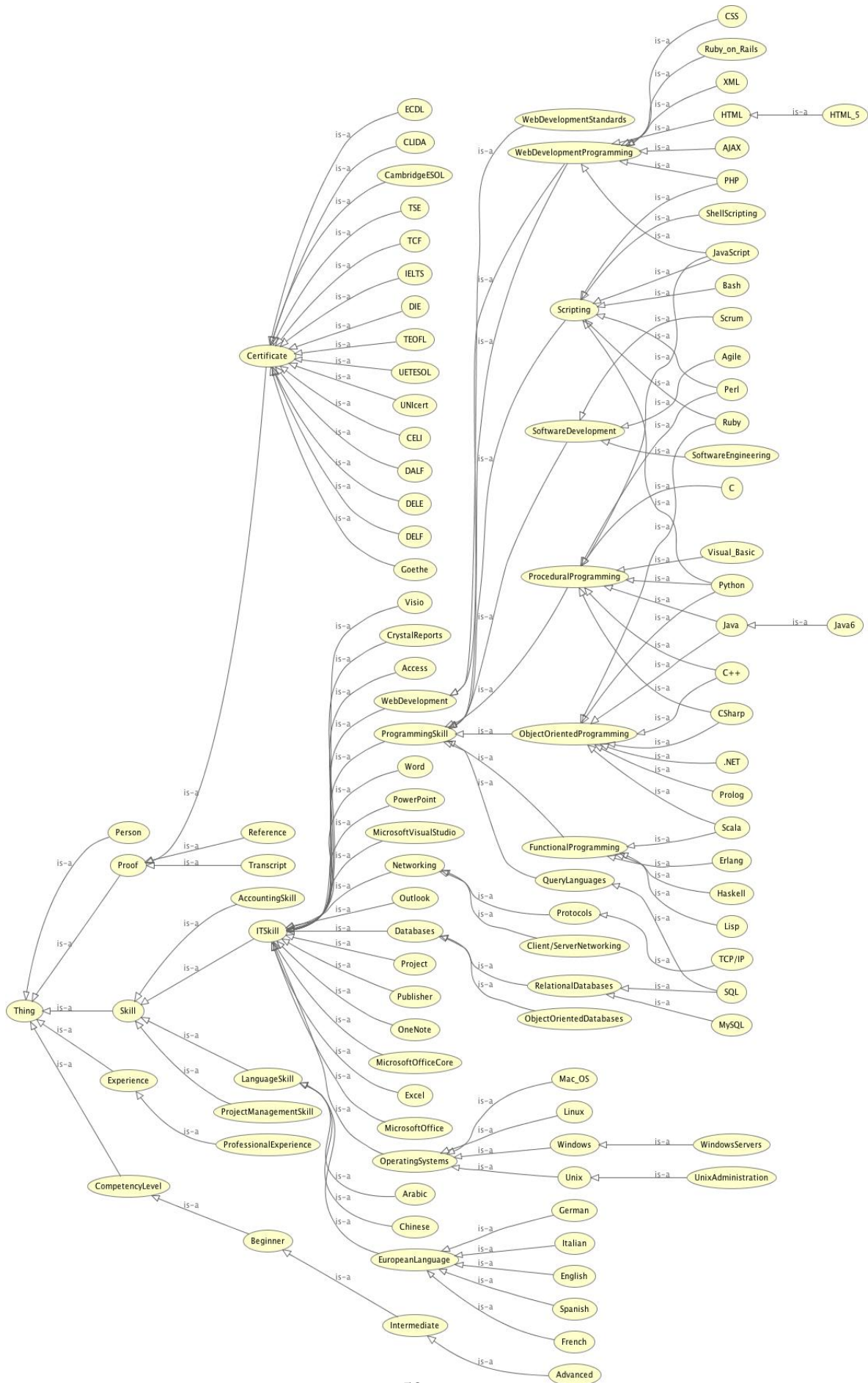
- [1] What is protégé-owl? <http://protege.stanford.edu/overview/protege-owl.html>, 2013. Last accessed: April 15, 2013.
- [2] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics. In *Handbook of Knowledge Representation*, chapter 3, pages 135–180. Elsevier, 2008.
- [3] Jurgen Bock, Peter Haase, Qiu Ji, and Raphael Volz. Benchmarking OWL Reasoners. In *Proceedings of the Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*, June 2008.
- [4] Simona Colucci, Tommaso Di Noia, Eugenio Di Sciascio, Francesco M. Donini, Marina Mongiello, and Marco Mottola. A formal approach to ontology-based semantic match of skills descriptions. *J. of Universal Computer Science, Special issue on Skills Management*, 9:1437–1454, 2003.
- [5] Ofer Strichman Daniel Krning. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.
- [6] Frank van Harmelen Deborah L. McGuinness. Owl web ontology language. <http://www.w3.org/TR/owl-features/>, 2004. Last accessed: April 14, 2013.
- [7] Kathrin Dentler, Ronald Cornet, Annette ten Teije, and Nicolette de Keizer. Comparison of reasoners for large ontologies in the owl 2 el profile. *Semant. web*, 2(2):71–87, April 2011.
- [8] Tommaso Di Noia, Eugenio Di Sciascio, Francesco M. Donini, and Marina Mongiello. Abductive matchmaking using description logics. In *Proceedings of the 18th international joint conference on Artificial intelligence, IJCAI’03*, pages 337–342, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.
- [9] Tommaso Di Noia, Eugenio Di Sciascio, Francesco M. Donini, and Marina Mongiello. A system for principled matchmaking in an electronic marketplace. In *Proceedings of the 12th international conference on World Wide Web, WWW ’03*, pages 321–330, New York, NY, USA, 2003. ACM.
- [10] Jianfeng Du, Shuai Wang, Guilin Qi, Jeff Z. Pan, and Che Qiu. An abductive cqa based matchmaking system for finding renting houses. In *Proceedings of the 2011 joint international conference on The Semantic Web, JIST’11*, pages 394–401, Berlin, Heidelberg, 2012. Springer-Verlag.
- [11] Alberto Fernández, Axel Polleres, and Sascha Ossowski. Towards fine-grained service matchmaking by using concept similarity. In *SMRR*, 2007.

-
- [12] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
- [13] Pascal Hitzler. Whats happening in semantic web. In Petko Valtchev and Robert Jschke, editors, *Formal Concept Analysis*, volume 6628 of *Lecture Notes in Computer Science*, pages 18–23. Springer Berlin Heidelberg, 2011.
- [14] Matthew Horridge, Holger Knublauch, Alan Rector, Robert Stevens, and Chris Wroe. A practical guide to building owl ontologies using the protégé-owl plugin and co-ode tools edition 1.0. *The University Of Manchester*, 2004.
- [15] Ian Horrocks. Ontologies and the semantic web. *Communications of the ACM*, 51(12):58–67, December 2008.
- [16] Yong-Bin Kang, Yuan-Fang Li, and Shonali Krishnaswamy. Predicting reasoning performance using ontology metrics. In Philippe Cudr-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web ISWC 2012*, volume 7649 of *Lecture Notes in Computer Science*, pages 198–214. Springer Berlin Heidelberg, 2012.
- [17] C. Larman. *Applying Uml and Patterns: An Introduction to Object-Oriented Analysis and Design, and the Unified Process*. Safari electronic books. Prentice Hall PTR, 2002.
- [18] Tommaso Noia, Eugenio Sciascio, and Francesco M. Donini. Extending semantic-based matchmaking via concept abduction and contraction. In Enrico Motta, Nigel R Shadbolt, Arthur Stutt, and Nick Gibbins, editors, *Engineering Knowledge in the Age of the Semantic Web*, volume 3257 of *Lecture Notes in Computer Science*, pages 307–320. Springer Berlin Heidelberg, 2004.
- [19] Natalya F. Noy and Deborah L. McGuinness. Ontology development 101: A guide to creating your first ontology. <http://www.ksl.stanford.edu/people/dlm/papers/ontology101/ontology101-noy-mcguinness.html>, 2001. Last accessed: April 8, 2013.
- [20] Uli Sattler and Thomas Schneider. Description logics: a nice family of logics. http://www.informatik.uni-bremen.de/~ts/teaching/2012_dl/slides/day1_intro_a.pdf, 2012. Last accessed: April 15, 2013.
- [21] D. C. Smith, C. Irby, R. Kimball, W. L. Verplank, and E. Harslem. Designing the star user interface. *Byte*, 7(4):242–282, 1982.
- [22] A. Stellman and J. Greene. *Applied Software Project Management*. O’Reilly Media, Incorporated, 2006.
- [23] Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez, and Boris Villazón-Terrazas. How to write and use the ontology requirements specification document. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part II*, OTM ’09, pages 966–982, Berlin, Heidelberg, 2009. Springer-Verlag.

- [24] Gruber Tom. **Ontology.** <http://tomgruber.org/writing/ontology-in-encyclopedia-of-dbs.pdf>, 2008. Last accessed: April 14, 2013.
- [25] W3C. **Vocabularies.** <http://www.w3.org/standards/semanticweb/ontology>, 2013. Last accessed: April 14, 2013.

Appendix A

Full Ontology Hierarchy



Appendix B

OWL API Main Methods Used

OWL API Method	Purpose
<code>getInstance()</code>	Retrieve candidates that fulfil the <code>dlQuery</code> conditions
<code>getSubClasses()</code>	Retrieve all the sub-classes of a class: e.g. <code>Proof</code>
<code>getAnnotations()</code>	Retrieve class annotation - used for the types of <code>Proof</code>
<code>getClassesInSignature()</code>	Retrieve all classes in the ontology
<code>getRootOntology()</code>	Retrieve the OWL ontology
<code>createOWLOntologyManager()</code>	Create an ontology manager used for all methods
<code>loadOntologyFromOntologyDocument()</code>	Load an ontology from its IRI
<code>parseClassExpression()</code>	Parse the <code>dlQuery</code> string to retrieve its corresponding class expression
<code>precomputeInferences()</code>	Get the reasoner to compute inferences in the ontology

Appendix C

Case Study - Experiment Queries

Fig 6.2: `hasSkill some Access and hasProof some Proof, where Access is marked as Desirable`

Fig 6.3 and Fig 6.4:

- single query: `hasSkill some Access and hasProof some Proof, where Access is marked as Desirable`

- two queries: `hasSkill some Access and hasProof some Proof; hasSkill some Word and hasProof some Proof, where Access and Word are marked as Desirable.`

- four queries: `hasSkill some Access and hasProof some Proof; hasSkill some Word and hasProof some Proof; hasSkill some Scala and hasLevel some Beginner; hasSkill some PowerPoint and hasProof some Proof, where Access, Word, Scala and PowerPoint are marked as Desirable.`

Fig 6.5: `hasSkill some German; hasSkill some English, where German is marked as Compulsory and English is marked as Desirable`

Appendix D

Case Study - Match Maker Results Breakdown

Person	Programming Skill (+Experience)	C++ (+Exp)	Java (+Exp)	Linux (+Exp)	Software Development (+Exp)	Software Development	Software Engineering	Networking (+Exp)	TCP/IP (+Exp)	Agile	Scrum	Total
Dan Hunt	✓	0	1 (+1)	0	1	✓	✓	1 (+1)	1 (+1)	1	1	9
Jack Smith	✓	1 (+1)	0	1 (+1)	1 (+1)	✓	✓	0	0	1	0	7
Emily Brown	✓	0	1	0	1 (+1)	✓	✓	1	0	1	1	6
Jessiva Brooks	✓	0	0	0	1 (+1)	✓	✓	1	0	1	0	4
Michelle Rivero	✓	0	1 (+1)	0	1	✓	✓	0	0	0	0	3

Note: The tick suggests the fulfilment of a compulsory skill.