FREIE UNIVERSITÄT BOZEN
LIBERA UNIVERSITÀ DI BOLZANO
FREE UNIVERSITY OF BOZEN · BOLZANO

KRDB RESEARCH CENTRE
KNOWLEDGE REPRESENTATION
MEETS DATABASES

Faculty of Computer Science, Free University of Bozen-Bolzano,
Via della Mostra 4, 39100 Bolzano, Italy
Tel: +39 04710 16000, fax: +39 04710 16009, http://www.inf.unibz.it/krdb/

KRDB Research Centre Technical Report:

A Graphical User Interface for Querytool

M. Trevisan

Affiliation	KRDB Research Centre, FUB
Corresponding author	M. Trevisan evenjn@gmail.com
Keywords	ontology-based data access, menu-based natural language interface
Number	KRDB09-06
Date	16-10-09
URL	http://www.inf.unibz.it/krdb/

Abstract

This document reports about the development of a web-based graphical user interface for QueryTool. The user interface integrates two other key components of the QueryTool; the query logic (API) subsystem and the natural language generation subsystem. The document collects and refines the requirements for the GUI and describes the design choices faced during the realization of the QueryTool interaction model in the Google Web Toolkit framework.

Contents

Chapter 1

Introduction

QueryTool is an editor for queries over knowledge bases. The ideas behind QueryTool have been laid out first in [1] in the context of the SEWASIE (SEmantic Webs and AgentS in Integrated Economies) European IST project. Verbatim from [1]:

> "The query interface is meant to support a user in formulating a precise query – which best captures her/his information needs – even in the case of complete ignorance of the vocabulary of the underlying information system holding the data. The final purpose of the tool is to generate a conjunctive query (or a non nested Select-Project-Join SQL query) ready to be executed by some evaluation engine associated to the information system."

The tool has undergone many incarnations since its inception. This document reports on the development of a Web-based graphical user interface integrating the latest version of the QueryTool engine together with a natural language generator developed for this purpose.

1.1 Objectives

The objective of this project is to develop a web-based graphical user interface for QueryTool suitable for a demonstration. The user interface should not be a stand-alone interface, but it should be integrated with two other components of the QueryTool, namely the query logic subsystem and the natural language generation subsystem. These subsystems are currently being developed in different projects and they reached an almost final state. Since the project requires integrating the two subsystems, this activity will also help detecting architectural issues in the interaction between these two subsystems, such as wrong assumptions on the behaviour of other parts of the system, or the lack of functionality which was not planned but which is nevertheless necessary. This project should report any issue of this kind encountered.

1.2 Related Work

This work is related to a family of menu-driven natural language interfaces for databases, early examples of which are [5] and [3]. The purpose of these systems is to guide the user in building a query. These interfaces let the user compose a natural language text by inserting, and deleting fixed blocks of text, choosing from a set of blocks suggested by the system. The natural language description of the user's informative need composed in this way hides an underlying query formulated using a standard query language (e.g., SQL).

The latest development in this area comes from research on controlled natural languages (CNLs). For example, [2] describes a controlled natural language interface for a wiki authoring system[1] using the Attempto controlled natural language. Likewise, for Attempto and several other controlled natural languages ported to the Grammatical Framework [4] a web-based predictive editor has been developed[2]. While both these systems and QueryTool restrict the fragments of natural language available to compose the query, the main difference between these systems and QueryTool is that the former rely on constraints on the grammar of the controlled language, while QueryTool relies on constraints on the ontology being queried. This also impacts the interface interaction model. The CNL-based tools support the user by suggesting the words to append at the end of the text composed so far, and they don't support directly the replacement and the deletion of sections of the text. They could, however, allow the user to free type in any portion of the sentence, and accept the typed text as long as it fulfils the constraints of the CNL. Our system does not allow free typing (at least not yet) and offers instead a rich support for substitution of portions of the text.

1 E.g. http://attempto.ifi.uzh.ch/webapps/acewikigeo/

2 E.g. http://www.cs.chalmers.se/~aarne/GF/demos/index.html

Chapter 2

Requirements

In this section we present the requirements for the developed system. These requirements have been collected from informal discussions with the stakeholders in the context of a different project. We reviewed and corrected them, and we present them here in detail. For the sake of readability, in this section we will describe the requirements as if the system was already built. A sentence like [1] is to be read as [2].

[1] The system can produce a XML file containing a map from URIs to natural language templates.

[2] The system must be able to produce a XML file containing a map from URIs to natural language templates.

In general terms, the system must integrate two existing systems, namely the query logic system and the natural language generation (NLG) system, and it must have a user interface which allows to interact with the query logic system, benefiting from the services of the NLG system. The architecture must be client-server, and the client must be executed on a web browser. The system must provide a visual access to the query, and editing facilities for it, within a web page. The query must be represented as a continuous text, composed as a sequence of text spans. Each label of the query must be mapped to a span, the map being injective.

In the following, we describe in detail a series of requirements on the interactive features of the interface that build on this initial specification.

2.1 Hovering

In graphical user interfaces terminology, the user hovers on a graphical element whenever the mouse cursor moves from some point outside the element to some point inside the element. In normal conditions, as the user hovers on the query, the system gives visual hints about its structure:

- When the user hovers on a span associated with the label of a node N, the span becomes lightly highlighted, together with all the spans associated with a label of N, or with the label of an edge leaving N, or with the label of a descendant of N or with the label of an edge leaving a descendant of N (see figure 1, part 2).

- When the user hovers on a span that is associated with the label of an edge E, the span becomes lightly highlighted, together with all the spans associated with a label of the node E is entering (see figure 1, parts 3 and 4).

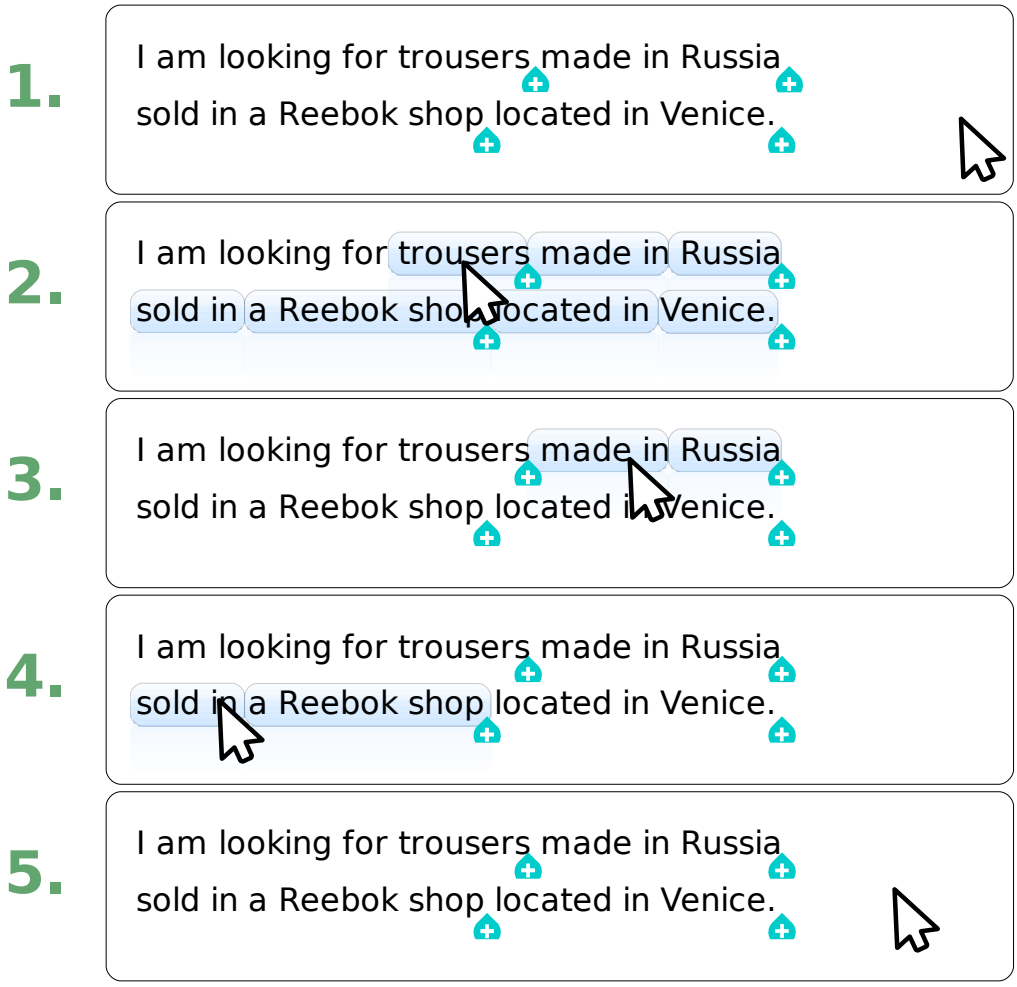

Illustration 1: Hovering triggers a light highlighting effect

In any case, the highlighting is such that distinct spans are visualized as distinct: this may be achieved by rounding corners of the highlighted area around each distinct span, as displayed in figure 1. This highlighting effect on hovering triggers in any condition except when a menu is displayed. The visual appearance of selected nodes does not change as the result of an hovering event.

At the right side of the last of every list of spans associated with labels of the same node there is a button. These buttons are displayed in figure 1 as blue onions with a white cross in it. We will refer to this button as the add-button of the node. These buttons are always visible except when one or more spans are selected. As the buttons appear and disappear as a consequence of changing selections, they should not interfere with the layout of the text. Each of

these buttons is associated with a single node of the query, and hovering on the button will highlight lightly all the spans associated with a node label of the associated node.

2.2 Sticky edges

Sticky nodes are not affected by deletion and substitution operations. The system allows to mark a node as sticky by clicking on the span associated with the label of the edge entering the node. For this reason in the GUI interaction model there are sticky edges but no sticky nodes. A sticky edge can be turned not sticky by clicking again on the span associated with its label. The visual representation of spans associated with labels of sticky edges differs from the one of spans associated with labels of non-sticky edges in that the text of the former is bold. Spans associated with node labels are not visually affected by sticky edges.

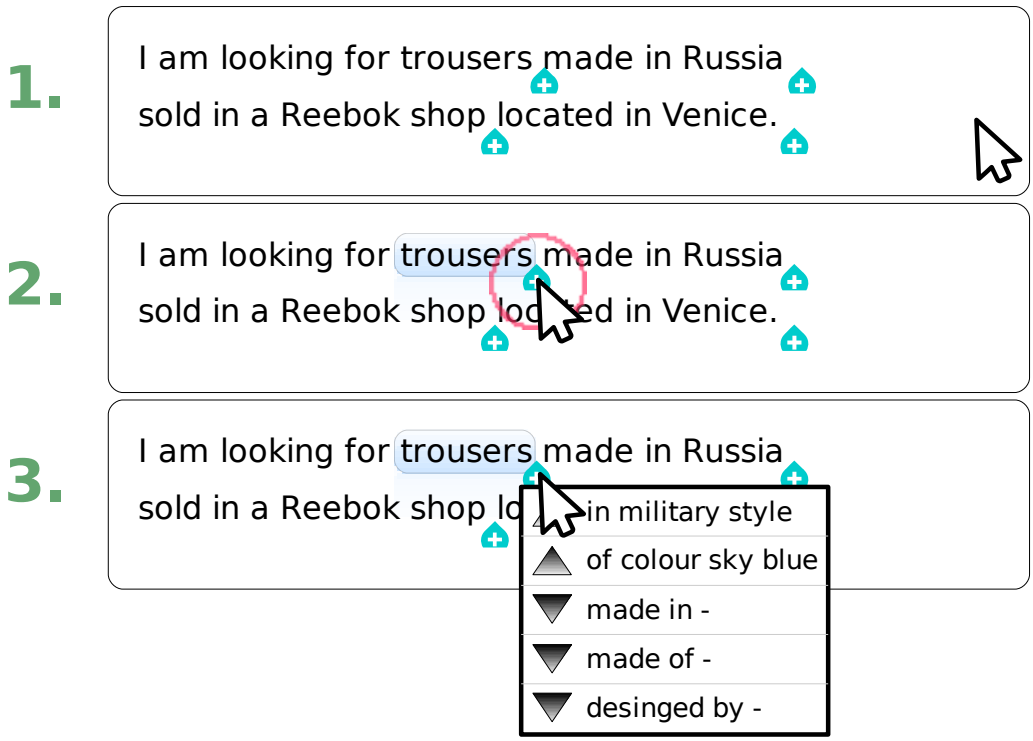

Illustration 2: Sequence of interaction for extending a query with a compatible concept. (Part 1)

2.3 Extension of the query

The query logic subsystem provides two instruments to extend an existing query with new constraints: the addition of a compatible atomic concept to an existing node (`addCompatible`), and the addition of a new node with a node label, connected to an existing node via a new labeled edge (`addProperty`). The GUI provides access to these functions through a pop-up menu, activated clicking on the add-button of a node. Illustration 3 displays a sample interaction which results in the invocation of `addCompatible`.

The menu contains a selection of available arguments that can be used to invoke either `addCompatible` or `addProperty`. The selection of arguments is provided on request by the logic subsystem, through the methods `getCompatibles` and `getProperties`. Both these methods take as argument a specific node of the query, and return a list of atomic concepts and couples, each couple consisting of a relation and an atomic concept. Each element of this list can be proposed to the user as possible arguments for the invocation of `addCompatible` or `addProperty` on the specific node. The natural language generation system is capable of producing a natural language description of an atomic concept or a couple.

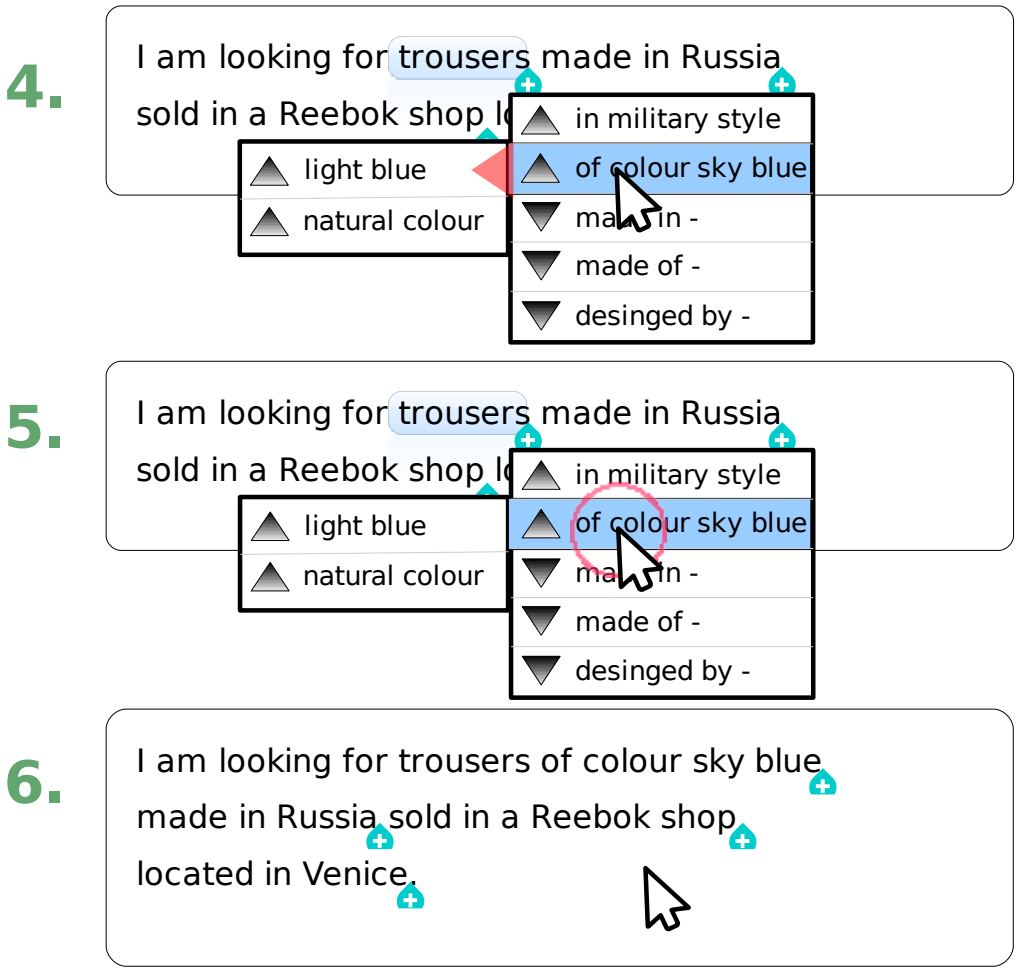

Illustration 3: Sequence of interaction for extending a query with a compatible concept. (Part 2)

Upon clicking on the add-button of a node, the GUI displays a menu containing a list of options, each option being associated with an element retrieved via `getCompatibles` or `getProperties`, each option labeled with the natural language description of the associated element. In this menu, the options associated with compatible atomic concepts are listed first, followed by options associated with couples. Each option comes with an icon on the left: a downward-pointed triangle for couples, and an upward-pointed triangle for compatible atomic concepts. Hovering on any of these options triggers a pop-up menu, on the side of the existing menu. We will refer to this menu as the sub-menu associated with the option.

The sub-menu associated with a compatible atomic concept option contains a selection of options associated with direct super-concepts of the compatible atomic concept, labeled in a fashion similar to the way the options are labeled in the other menu. Each of these options, in turn, comes with a upward-pointed triangle button on the left side. Each of these option triggers a similar pop-up menu upon hovering.

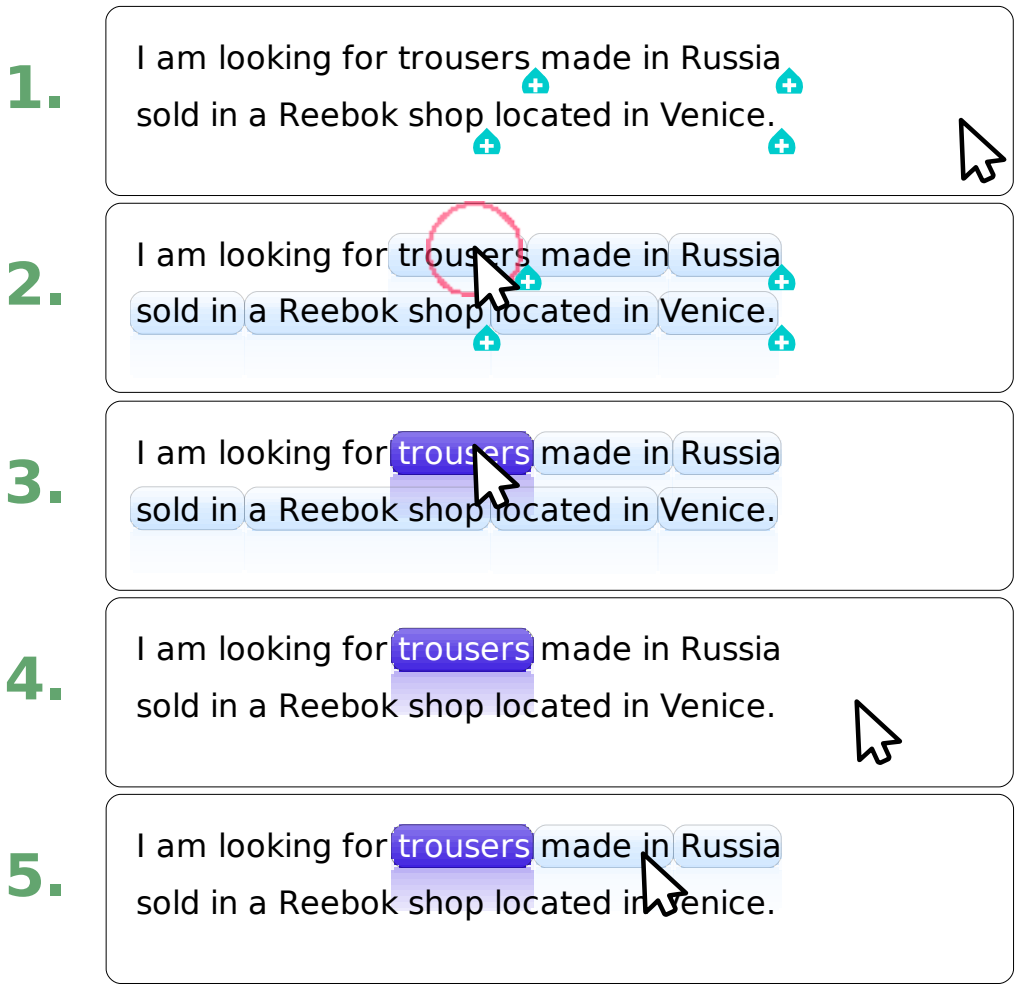

Illustration 4: Sequence of interaction for creating a label selection

The sub-menu associated with a couple option contains a selection of options associated with couples, such that the property is the same in both couples, but the atomic concept of the second couple is a direct sub-concept of the atomic concept in the first couple. Each option labeled in a fashion similar to the way the options are labeled in the other menu. Each of these options, in turn, comes with a downward-pointed triangle button on the left side, and triggers a similar pop-up menu upon hovering.

In any case, the ramifications and the contents of these sub-menus are provided on request by the logic subsystem together with the list of compatible atomic concepts returned by `getCompatibles`, or together with the list of couples returned by `getProperties`.

Clicking on any of these options triggers the invocation of either `addCompatible` or `addProperty` using as arguments the data contained in the element associated with the option clicked, together with the node associated with the add-button clicked to show the pop-up menu. Upon clicking, the menus disappear, and the GUI updates its representation to reflect the new state of the query after it was extended.

2.4 Selections

Deletion and substitution operations operate on a selection of labels. In the query logic subsystem, a selection is a set of node labels containing:

- no label (empty selection), or
- a single label (label selection), or
- all the labels of a node (node selection), or
- all the labels of a node together with all the labels of descendants of that node (sub-tree selection)

To create a label selection, the user must click on the span associated with the desired label. To create a node selection, the user must double-click on a span associated with a label of the desired node. To create a sub-tree selection, the user must triple-click on a span associated with a label of the root of the desired sub tree. To create an empty selection, the user must click on an area of the GUI where clicking does not have any other effect, for example on the white space between the lines of the text of the query, or on a span not associated with any label.

When a node has only one node label, a label selection containing that label happens to be also a node selection, containing all the labels of the node. Similarly, when a node has no children, a node selection, containing all the labels of the node, is also a sub-tree selection. Moreover, when a node has only one label and no children, a selection containing that label is a label selection, a node selection and a sub-tree selection. For the purposes of the GUI, a selection cannot fit more than one classifications, therefore selections which fit into two or more classes are to be considered as fitting only the higher priority class. The label selection class has the lowest priority, while the sub-tree selection has the highest. When a double click would trigger the same kind of selection a single click would (i.e. a node selection), it triggers a sub-tree selection instead.

Spans associated with selected labels are highlighted in a stronger way then they are when highlighted because of hovering. In the case of a sub-tree selection, labels associated with edges of the sub-tree are also highlighted. Unlike the highlighting effect triggered by hovering, it is not possible to distinguish the distinct spans covered by the selection. This may be achieved by highlighting all the spans without rounding the corners of the highlighted area around each distinct span. Illustrations 4 and 5 show the different highlighting triggered by hovering and by selection, and their interaction.

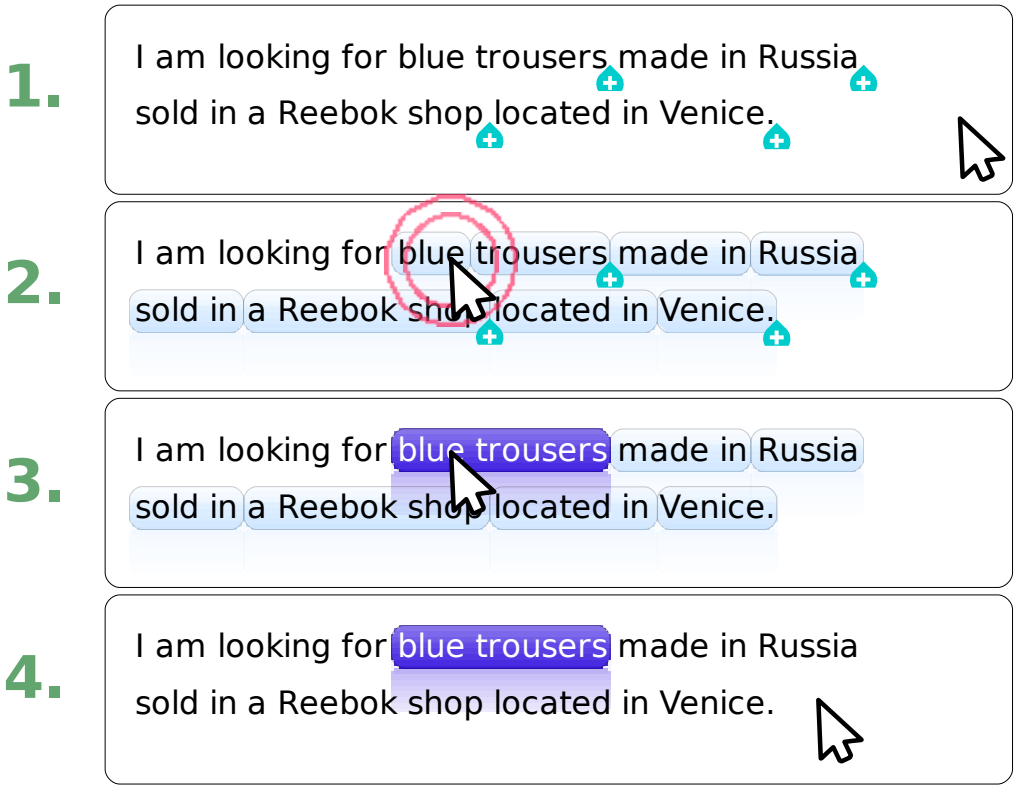

Illustration 5: Sequence of interaction for creating a node selection

2.5 Deletion

The system allows to delete a selected portion of the query by pressing the delete key on the keyboard. The deletion is possible only as long as the selection is a label selection, a sub-tree selection, or an empty selection. In case the delete key is pressed while the current selection is a node selection, the system will inform the user that the selection cannot be deleted. Upon deletion, the selection becomes empty, and the GUI updates its representation to reflect the new state of the query after the elements have been removed.

2.6 Substitution

The system allows to replace a selected portion of the query with a node with a single label, or in case the selection was a label selection, to replace the selected label with another label. The logic subsystem provides a function, `substitute`, which takes as input a selection and an atomic concept. The GUI provides a visual access to this function: left clicking or long-clicking on a span of text strongly highlighted because of a selection triggers a pop-up menu. This menu contains several options, each option corresponding to an atomic concept which can be used together with the selection to invoke the substitute function.

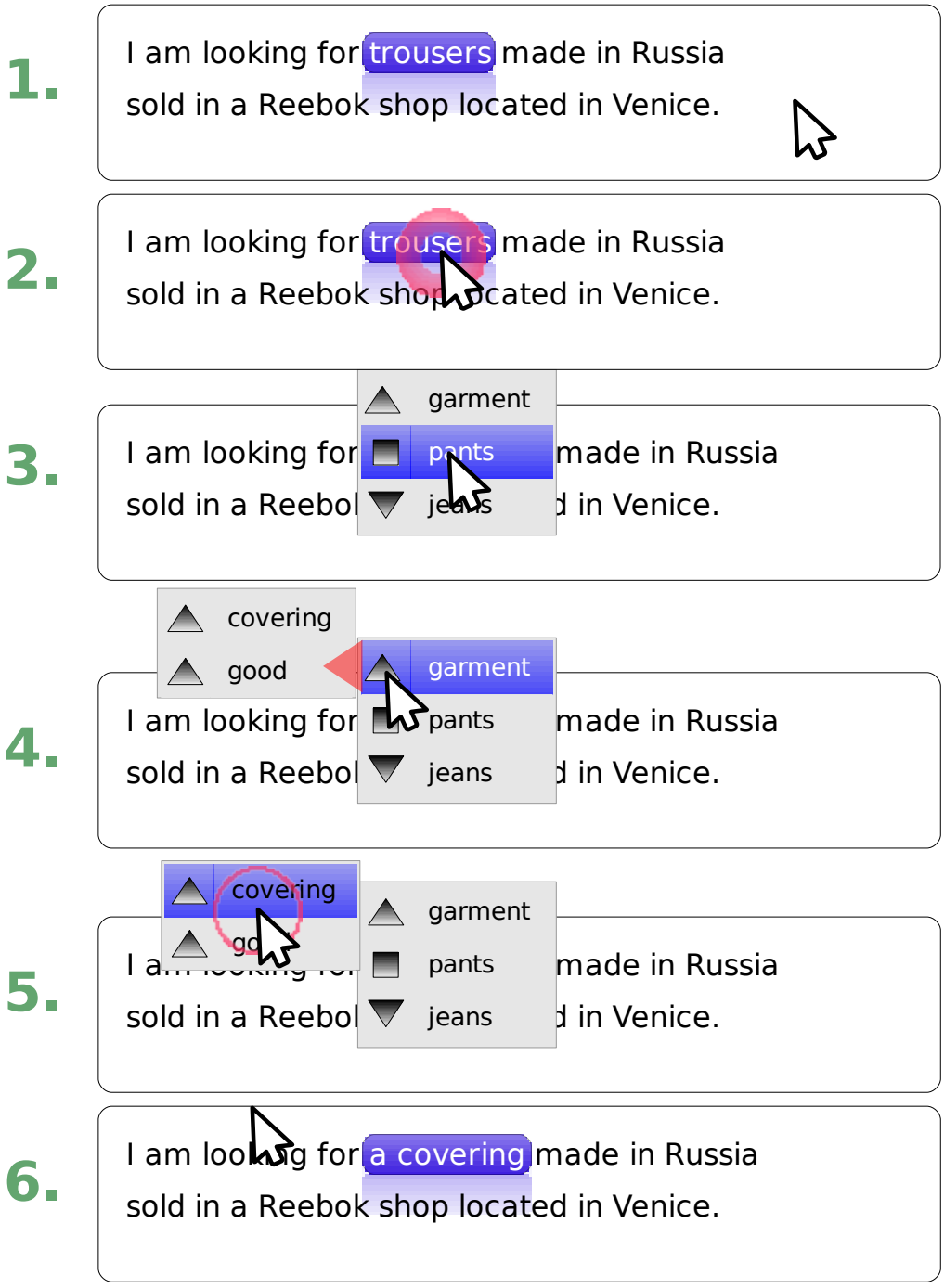

Illustration 6: Sequence of interaction for a substitution

These options can be retrieved on request from the logic subsystem using the function `getReplacements`. This function returns three sets of concepts and a map from concepts to

sets of concepts. The first set of concept returned is the set of direct super-concepts of the concept represented by the selection. The second set of concepts is the set of concepts which are equivalent to the concept represented by the selection. The third set of concepts is the set of concepts which are direct sub-concepts of the concept represented by the selection, compatible with the rest of the query.

The map associates each concept X in the first set to the set of X's direct super-concepts, and each concept Y, direct or indirect super-concept of X, to the set of Y's direct super-concepts. In a specular way, the map associates each concept X in the third set to the set of X's direct sub-concepts, and each concept Y, direct or indirect sub-concept of X, to the set of Y's direct sub-concepts.

Each option in the menu is associated with an element of one of these three sets. Options associated with elements from the first set appear first in the list, with an upward-pointed triangle icon on the left of each option. Options associated with elements from the second set appear following the ones from the first set, with a square icon on the left of each option. Options associated with elements from the third set appear following all the other ones, with a downward-pointed triangle icon on the left of each option. The natural language generation subsystem can generate text describing an arbitrary atomic concept. Each option is labeled with the generated description of the associated atomic concept.

Hovering on an option triggers a pop-up menu which appears on the side of the existing menu. This menu is populated with options, each associated with an element in the set associated by the map to the atomic concept the option triggering the pop-up menu was associated with. For example, hovering on an option associated with an element X of the first set will trigger a pop-up menu containing as many options as there are direct super-concepts of X. In any case, the structure and the content of these sub-menus reflects the information provided by the map retrieved from the query logic subsystem via the `getReplacements` function. The options in these sub-menus are visually similar to the options in the existing menu. Moreover, hovering on one of these labels also triggers a similar sub-menu.

Clicking on any such option triggers an invocation of `substitute`, with the current selection and with the concept associated with the option as actual arguments of the invocation. Upon completion of the invocation, the selection becomes empty, and the GUI updates its representation to reflect the new state of the query after the selected elements have been replaced with new ones.

Chapter 3

Design

In order to develop a Web-based application, of the system, we chose to rely on Google Web Toolkit (GWT). GWT is a software framework for Web-based applications. It has a client-server architecture, where the server side software is run on a Web server, and the client software is run on the user's browser. GWT is a framework in Java, so applications instantiating it are also written in Java, including the code for the client side. However, GWT comes with an application which translates the client-side code in Javascript code, which can be run immediately on a browser. The generated Javascript code is highly optimized, and it includes workarounds for the idiosyncrasies of different Web browsers. This system allows to design and develop a client-server Web system almost entirely in Java. Nevertheless, certain graphical properties of the client interface need to be specified using HTML and CSS.

The architecture of the whole system is depicted in figure 7. The server side system is composed of three subsystems: the query logic system, the natural language generation system, and the Web service. The Web service communicates with the other two components, which are otherwise not connected, to provide a single, simple interface to the client. GWT does not restrict the Java code on the server side, but it does restrict the code on the client side, which need to be translatable into Javascript. GWT imposes further restrictions of the data structures which can be transmitted between client and server (Interchange Data), in addition of binding them to be translatable into Javascript.

While relying on a client-server architecture, the system built so far is not taking advantage of this. Rather, it behaves like a monolithic application. In particular, the server creates a new query at boot time, and provides it to all instances of the client requesting it. It is not possible to have different clients work on different queries at the same time.

3.1 Client-Server interface

This section describes the interface exposed by the Web service, and the data structures it relies on to transfer the data to the client. In our system, the server and the client communicate via the GWT remote procedure call (RPC) system. The GWT RPC system imposes severe restrictions on the data structures which can be sent from one side of the system to the other. In practice, legal data structures must be either primitive types, arrays of legal structures, Java-serializable classes containing only legal data structures in their fields. For example, the class `java.net.URI` does not fulfil these requirements. For this reason, instead of propagating these

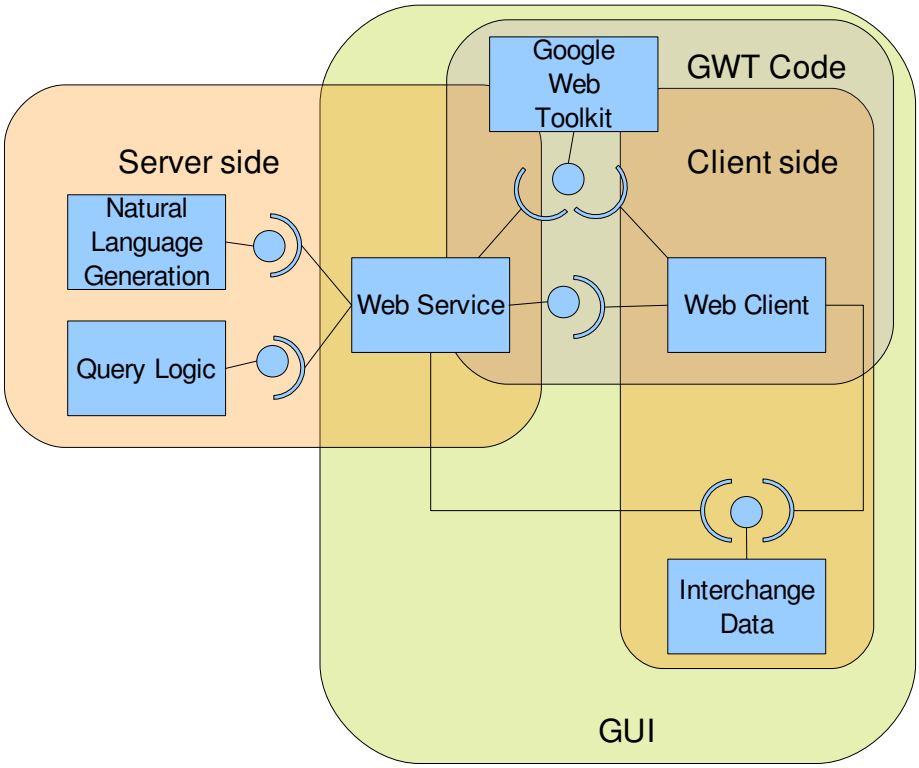

Illustration 7: Main components of the Querytool graphical user interface

constraints into the code of the query logic subsystem and in the code of the natural language generation subsystem, we chose to design a family of data structures specifically for the purpose of transporting the data from the Web server to the client and vice-versa. These data structures are: GWTQuery, GWTNode, GWTLabel, GWTToken, GWTSelection, GWTAtomicConcept, GWTRoleRange, GWTExtensionsList.

A GWTQuery consists of the root node of a tree of GWTNodes, plus a list of GWTTokens. A GWTNode contains a list of node labels in the form of GWTLabels, an edge label in the form of a GWTLabel, and a list of child nodes in the form of GWTNodes, plus an integer ID. It may optionally refer to a list of extensions, a GWTExtensionsList. A GWTQuery is always built as a reproduction of a query of the query logic subsystem. While the latter doesn't have explicit data structures to represent labels, GWTNode has GWTLabels. A GWTLabel consists of the string encoding of the URI of a SchemaClass or SchemaProperty, plus a pointer to the GWTNode it is attached to. GWTLabels are necessary for GWTTokens to establish a link between the generated text and the elements of the query. Each GWTToken consists of a fragment of text plus an optional pointer to a GWTLabel. The concatenation of the fragments of text contained in the GWTTokens stored in a GWTQuery results in the natural language representation of the query. Those GWTTokens which point to GWTLabels allow to use their text fragments as proxies for the elements of the query.

A GWTSelection is a data structure representing a selection created by the user. It is built on the Web browser and transmitted to the Web server whenever an operation takes a selection as argument. A GWTSelection consists of a pointer to a GWTNode, a pointer to a GWTLabel, and a boolean value to distinguish between node selections and sub-tree selections. These

elements allow to model any selection among those described in the requirements. In addition, a GWTSelection also contains the list of sticky nodes.

A GWTAtomicConcept contains the string identifier of a SchemaClass, a list of related GWTAtomicConcepts, and a label in natural language describing the SchemaClass. A GWTRoleRange contains the string identifier of a SchemaClass, the string identifier of a SchemaProperty, a list of related GWTRoleRanges, and and a label in natural language describing the SchemaProperty and the SchemaClass together. The server uses GWTAtomicConcept and GWTRoleRange as basic building blocks of the forests of possible arguments for addition and substitution operations. Each of these elements contains all the information needed to generate an option in a menu: a natural language label, the identifier of the value(s) to be provided as argument, and a list of elements to be displayed in a sub-menu.

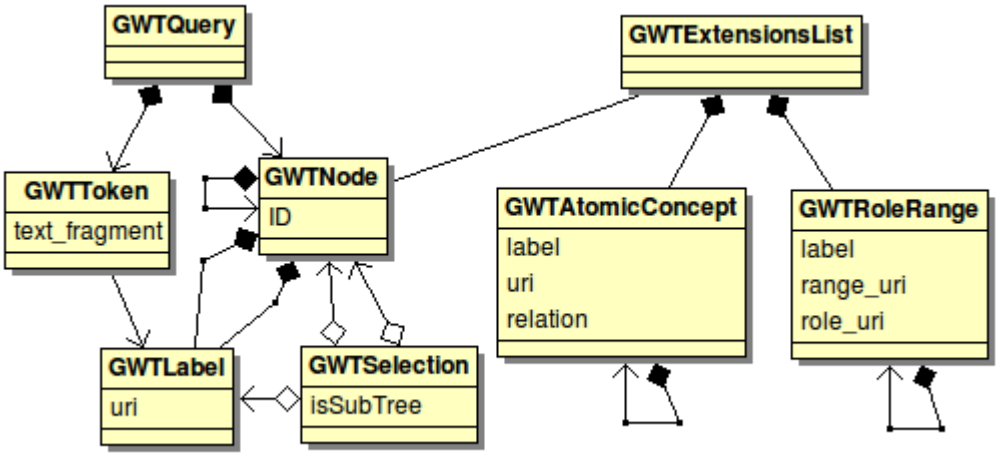

Illustration 8: UML Class Diagram of the used in the client-server interface

A GWTExtensionsList consists simply of a list of GWTAtomicConcepts and list of GWTRoleRanges. Each GWTAtomicConcepts also optionally contains a tag identifying it as a super-concept, sub-concept or equivalent-concept of some other concept which can be identified given the context.

Altogether, these classes contain all the information needed by the client to display the query and the possible operations on it. This data is sent to and sent from the Web service on the Web server, which exposes the following methods:

- getQuery – returns a GWTQuery mirroring the query built so far, if any, or creates a new query.
- getExtensions – takes as argument a list of GWTNode, and returns a list of GWTExtensionsList, each containing all the possible arguments which can be used together with the corresponding GWTNode in the input list to invoke either addCompatible or addProperty.
- getReplacements – takes as argument a GWTSelection, and returns a GWTAtomicConceptsList containing all the possible arguments which can be used together with the input GWTSelection to to invoke substitute.

- `addCompatible` – takes as arguments a `GWTNode` and a `GWTAtomicConcept`, and returns a GWTQuery mirroring the query built so far after the `addCompatible` operation of the query logic subsystem was executed.

- `addProperty` – takes as arguments a `GWTNode` and a `GWTRoleRange`, and returns a `GWTQuery` mirroring the query built so far after the `addProperty` operation of the query logic subsystem was executed.

- `substitute` – takes as arguments a `GWTSelection` and a `GWTAtomicConcept`, and returns a `GWTQuery` mirroring the query built so far after the `substitute` operation of the query logic subsystem was executed.

- `delete` – takes as argument a `GWTSelection` and returns a `GWTQuery` mirroring the query built so far after the `delete` operation of the query logic subsystem was executed.

All these services can be invoked by client through the Remote Procedure Call system provided by GWT. As such, they are asynchronous, meaning that requests are non-blocking.

3.2 Server system

The web service consists of a Java class implementing the methods declared in the exposed interface and a class which translates query data structures from and to the three different query representation languages used by the query manager, by the natural language generator, and by the GWT client.

As anticipated, the present design of the server does not allow multiple clients to interact concurrently. Instead, at creation time the server loads a predefined ontology, predefined linguistic resources, and creates an empty query which is readily available to clients.

The server comes with a utility to generate the lexical resources needed by the NLG subsystem. The generation takes as input an OWL file and produces a couple of XML files containing a lexicon and a mapping from ontology elements to language templates; these files should be reviewed and corrected by the ontology engineer. This utility is presently accessible from command line only.

3.2.1 Multiple query representations

From the architectural point of view, the main task of the server is to integrate the three systems: the GUI, the query logic subsystem, and the NLG subsystem. These three systems operate on three different data structures: the GUI uses its own because of restrictions imposed by GWT. The query logic subsystem and the natural language generation subsystem use each their own representation because they were developed independently over the course of the time. The two representations are similar, but at the same time they are not coincident. The query representation in the NLG system allows to mark an edge as being labeled with the inverse of a relation, while the query logic does not. Moreover, an explicit representation of labels is employed by the NLG system to track what elements of the query the text refers to, but labels are not represented within the query logic subsystem. Refactoring the two systems to use a single representation would increase the coupling between the two systems with no significant increase in performance nor in maintainability.

Instead of adopting a single data structure for all the tasks, we chose to design a class representing the translation of a query in different data structures. The class, called QueryTranslation, provides the following services:

- built using Query, it gives access to a NLGQuery and a GWTQuery mirroring the Query
- given a GWTNode, it retrieves the corresponding Node
- given a NLGLabel, it retrieves the corresponding GWTLabel
- given a NLGNode, it retrieves the corresponding GWTNode

The translation is implemented mainly using hash maps in order not to introduce dependencies among the data structures of the three subsystems.

3.2.2 Server operations

The general flow of execution of an interaction with the query, invoked by the client through RPC, is the following:

- If any argument GWTNode or GWTLabel is provided, they are translated into a Node or into a SchemaClass, or into a SchemaProperty.
- The corresponding function in the query logic system in invoked with the translated arguments.
- The resulting query is translated into the NLG version and into the GWT version via the QueryTranslation.
- The main generation method of the NLG system is invoked using the NLG version of the query.
- The list of NLG tokens resulting from the generation is transformed into a sequence of GWTTokens in such a way that the GWTLabels the GWTTokens refer to correspond to the NLG labels the NLG tokens referred to.
- The GWTTokens are attached to the GWT version of the query and sent back to the client.

Most of the functions realized by the server simply follow these simple steps. The less obvious step being probably the transformation of NLG tokens into GWT tokens. NLG tokens produced by the NLG system contain references to query elements that are of no interest to the GUI. The GUI requires the association between spans and labels to be injective, so multiple adjacent tokens referring to the same entity need to be aggregated in one token, and further references to the same element within the text need to be removed. In addition, multiple tokens with no references to any element are aggregated into a single token.

The functions which follow these steps are relatively easy to implement. However, the server currently performs also two non-trivial tasks: getReplacements and getExtensions. The execution of these two methods is not trivial because the methods are currently not supported by the query logic subsystem. Therefore, the server relies on a custom implementation, using the basic reasoning services exposed by the query logic subsystem, to calculate the required

information. The present solution for these two methods is temporary and incomplete; it needs to be verified and possibly re-located within the query logic subsystem.

3.3 Client system

The web client consists in a composite widget embedded in the middle of a html page.

3.3.1 Text flow and highlighting effects

The GUI receives the natural language text describing the query in form of a sequence of tokens, possibly associated with edge and node labels of the query. The text should appear to the user as a single stream, but hovering and clicking should highlight only certain portions of it. We chose to represent the whole text as an html `<p>` paragraph, and use html `<span>` elements to apply different visual effects and associate different behaviours to different spans of text.

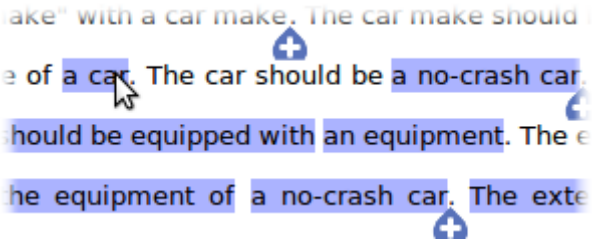

Illustration 9: Hovering effect using CSS styles: background colour

GWT provides a class, `InlineLabel`, which produces a simple `<span>` element, sensible to clicking and hovering, but not to focus[†]. This simple solution allows to control the text flow and the margins of the text using CSS. Encapsulating single spans into distinct html `<div>` elements would allow to move the focus on the single spans, which is necessary for using the GUI from the keyboard, but then CSS would not handle the paragraph together any more. Single spans can be applied different styles to highlight them with different shades, and they can be even decorated with a background image, to be tiled throughout the span area.

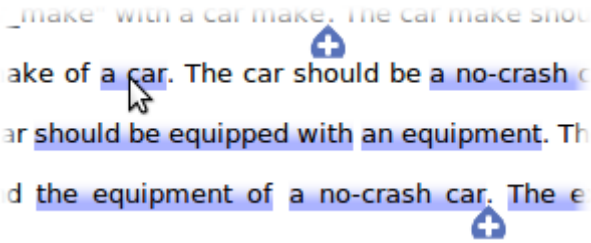

Illustration 10: Hovering effect using CSS styles: background image

An alternative design we considered was to encapsulate each text span in a separate button. This solution would allow to focus the single spans as well as to assign more complex background images, such as the ones displayed in the requirements section. However, this solution has some drawbacks: buttons don't get split automatically into two lines, so a long span of text would result in a large button, disrupting the layout of the text. Moreover, buttons cannot be arranged like justified text. In conclusion, the

† An element has the focus when it is responsive to keyboard-generated events. No two elements may have the focus at the same time.

overall effect would not look like a text at all. Therefore, we eventually adopted `InlineLabels`, sacrificing a straightforward keyboard access.
Single text spans are represented by a dedicated class, `Chunk`, and they are handled collectively by a separate class, `ChunkManager`. The light highlighting effects, depicted in figure 3, are triggered by hovering `Chunks`. The hovering events and their effect are handled and coordinated by a dedicated class, `HoveringManager`. The `HoveringManager` allows to disable hovering effects, for example when pop-up menus are active.

3.3.2 Add buttons

One of the most challenging features to realize was the add button. The challenge spurs from the combination of a few implicit and explicit requirements:

- The position of the button is determined by the position, on the screen, of certain spans of text.
- The button should be visible in certain scenarios but not in others
- When the button appears or disappears, the text layout should not change.
- The text layout should be as natural as possible.

We prototyped three possible design solutions for the add button.

The first solution is to wrap each text span into a container divided into two slots. The text span goes on the upper slot, and the button goes on the bottom slot, aligned on the right. As mentioned before, this solution is not feasible in that the HTML code generated by this solution places each text span in a separate container, and therefore the resulting text does not lay out as a simple paragraph. Therefore, it is not possible to control the layout of the generated text as a whole: it is not possible to justify it, long spans of text cannot be split on two lines, and the spacing between different spans is independent of the spacing between words, resulting in text which is not spaced uniformly.

The second solution is to realize the add buttons as pop-up elements, placed using the absolute coordinates of the relevant text spans. This solution has the advantage that the resulting text can flow freely as a single stream. The problem with this approach is that the position of the pop-up elements is not bound to the position of the text spans, and therefore any change in the layout of the text is not automatically reflected in the layout of the buttons, and must be handled programmatically case by case.

The third solution is to realize buttons as elements within the text flow. This solution has the advantage that the position of the buttons is automatically updated as the layout of the text changes, plus it has the advantage that the text is still a single stream whose layout can be easily controlled. The problem with this solution is that there is no way to prevent the buttons to flow to the next line instead of staying close to the labels they are associated with. Moreover, in order to hide the buttons without affecting the text layout, the button should be replaced with an empty space of the same size, resulting in a visible gap in the text flow.

We chose to pursue the second solution because it allows to achieve a better layout of both text and buttons, notwithstanding the increase of code complexity to keep the buttons aligned with the text. The issue with this approach is the text layout could change because of many kind of events, (e.g. the browser hiding the vertical scroll bar, some text becoming bold, resizing the browser window) and the system must be in control of all of them.

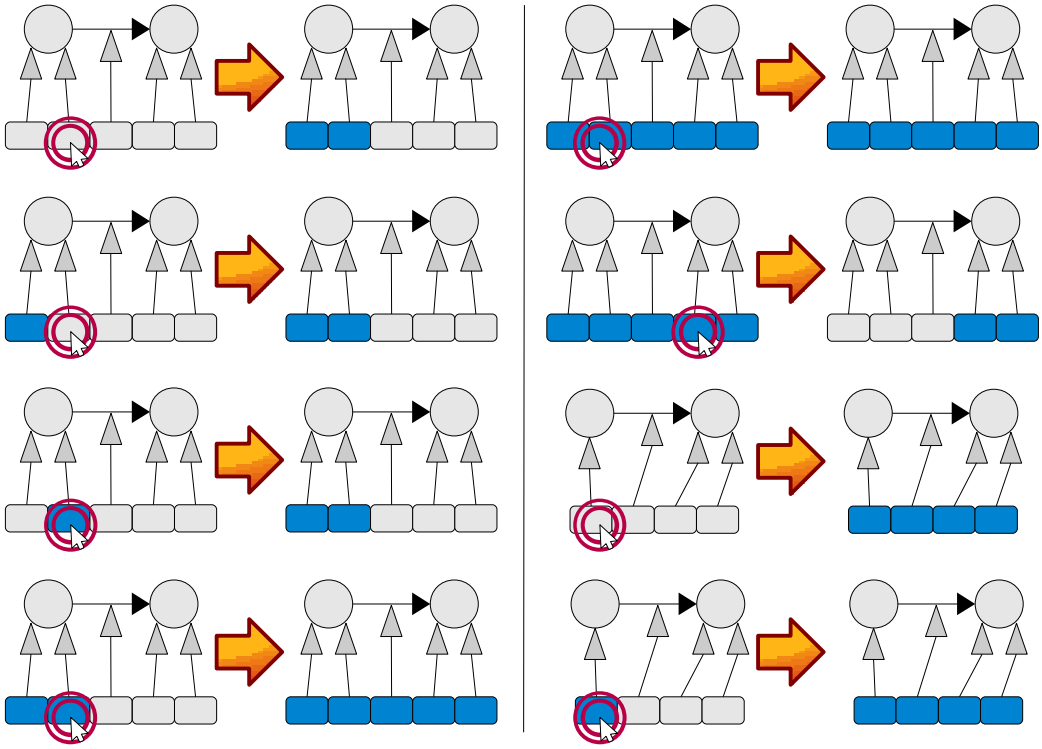

Illustration 11: Effect of double click in different contexts

3.3.3 Selections

The mechanism for creating selections could not be realized as it was specified in the requirements because of limitations of GWT. GWT can handle the simple left-mouse-button click, and, with a little extra work, the double click. However, there is no built-in support for right-mouse-button clicks nor for long clicks or triple clicks. In order to provide access to the three degrees of selection possible, we explored a different solution.

We decided to realize single clicks according to the requirements, and double clicks partially following the intended behaviour. We employ consecutive double clicks in place of triple clicks.

- A single click always clears the current selection and selects a single label.
- A double click on an label which was not part of a node selection, or which was itself a simple label selection triggers the selection of the entire node or, if the node has only a single node label, the entire sub-tree.
- A double click on a label which is part of a node selection triggers the selection of the entire sub-tree.

In this way, the single click maintains its traditional role, that is, to replace the current selection with the simplest selection containing the clicked element. The double click triggers instead an extended selection, where the degree of extension depends on the context. We believe that the user will never be surprised about the context-dependent effect of the double click, even when the second double click comes after a considerable amount of time after the first

one. Figure 11 shows a series of possible scenarios and the effect of double click in those scenarios.

Designing this functionality in GWT was intricate because double click events always come after two consecutive click events. As click events always reset the selection, it is not possible to rely on the current selection to calculate the effect of the double click. To handle them correctly, we had to count the number of click events since the last double click event. A number of click events equal to two means that the double click is coming right after another double click, and the selection needs to be expanded further. This system would need to be tweaked if other means of selecting elements were introduced (e.g. selection using keyboard).

The mechanism for creating the empty selection also needs some careful consideration. GWT distinguishes between two kinds of empty space: the empty space outside the root panel, and the empty space between click-sensitive widgets. An empty space outside the root panel exists when the GWT application is embedded in a component of an HTML page, instead of taking the whole page for itself. GWT does not allow to detect clicking events happening outside the root panel. It does, however detect events of focus being lost, also known as blur events, which could be a possible way to detect clicks outside the main panel: clicking outside the root panel shifts the focus from an element of the GWT application to an element outside it, therefore triggering a blur event but no detectable focus event. Unfortunately, this condition by itself cannot be exploited to trigger the creation of an empty selection, because the same condition is verified any time the focus shifts from an element of the application to another, a moment before the other element triggers a focus event, and there is no way to gain knowledge about the next scheduled events during the processing of a triggered event. Therefore, we set were forced to deviate from the requirements, as clicking on any element outside the widget itself won't clear the selection. We believe that this deviation is reasonable, as:

- If the interface is to be embedded in a web page, it allows the user to perform some activity on other elements of the page (e.g. clicking on hyper-links and other buttons) without losing the existent selection.

- If the interface is not to be embedded in a web page, the root panel can be extended to cover the whole page, obtaining the behaviour specified in the requirement.

For a different reason, it is not straightforward to trigger the creation of an empty selection when the user clicks on an empty space between the rows. It is possible to detect clicks in the empty space of an area, because it is possible to detect whether the user clicked in a panel or in a widget contained in the panel. The event triggered however, carries no information regarding which widget, if any, triggered the event, so it is not immediate to distinguish between clicks on widgets and clicks on the empty space. However, clicking on a click-sensitive element triggers a separate event, which is scheduled to be handled before the previously-mentioned event. It is therefore possible to have all click events generated by widgets, other than the panel, set a lock which prevents to create the empty selection. If this lock is set, the click handler for the panel gets to clear the lock; otherwise, it creates the empty selection. In this way it was possible to fulfil the requirement.

3.3.4 Substitution

The requirements specify that substitution is to be triggered by a right-click or by a long click on the selected text. As mentioned in the previous part, GWT provides no built-in support for right clicks nor for long clicks or triple clicks. Therefore, the available options to trigger substitution are either capturing a keyboard event, for example asking the user to press space to open

the substitution menu, or introducing a new graphical element in the interface. As add-buttons disappear once a selection is created, we chose to have a similar button, the substitute-button, appear when a selection is created, right after the add-buttons disappear. Clicking on this button triggers the substitution menu. The appearance and layout of this button is displayed in figure 6.

3.3.5 Menus

The requirements specify in detail how the forest of options is to be visually displayed, using a pop-up walking menu with a forest of sub-menus, as displayed in figure 12. This simple design has advantages and disadvantages.

The main advantage is that, in principle, its behaviour is well-known to the users, which makes it easier for users to learn how the tool works. However, this is not completely true because users don't expect that a menu voice giving access to a sub-menu can be also clicked to execute a command – as it would be the case in our system.

There are many problems with walking menus. Unlike many applications that employ them, in our application the content of these menus is not predetermined. Our menus may have an arbitrarily large number of options at any level (i.e. in the first menu as well as in any sub-menu) and an arbitrary depth. Using walking menus the user interface may get quickly cluttered. As sub-menus appear and disappear on hovering, as the user is forced to move the cursor along a narrow path, and as sub-menus overlap, a simple careless motion may cause the pointer to hover on the wrong menu, causing the disruption of the whole process. Even if we were able to lay out the menus properly, the user would have a hard time using them.

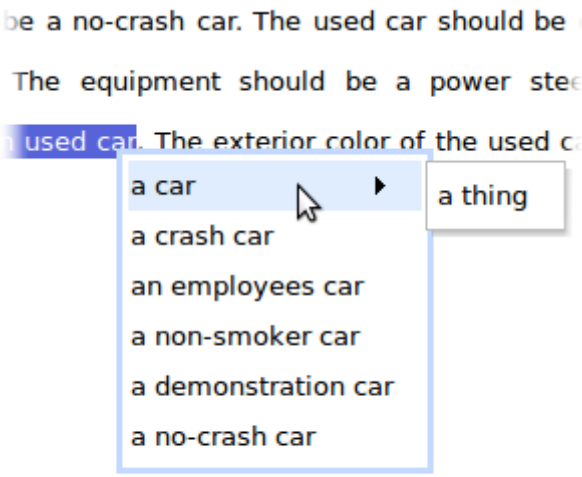

Illustration 12: A walking menu

However, GWT prevents us from doing so because of a technical limitation: built-in sub-menu widgets always position on the right of the parent menu, even if this happens to be outside the view-port of the browser. Moreover, as our menu options are labeled with fairly long fragments of natural language text, the menus tend to fill the horizontal space rather quickly. The result is, the sub-menu is often not visible, and any attempt to scroll the browser view-port to the right causes the whole chain of menus to disappear. As if it was not enough, GWT does not provide a way to display a custom image within a menu item.

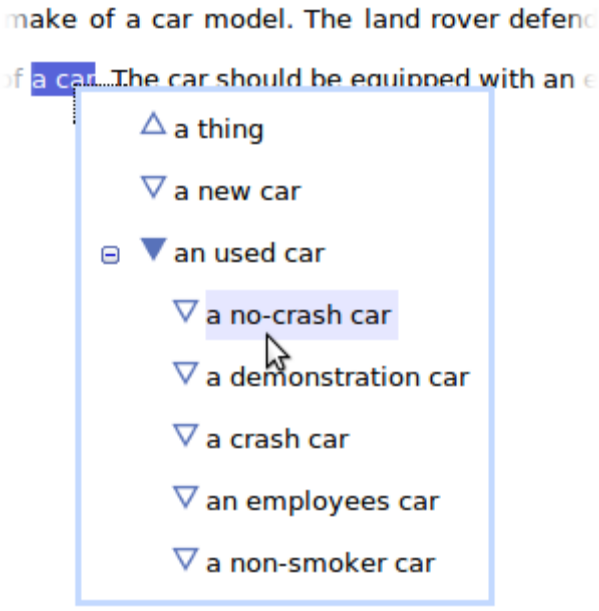

Illustration 13: A tree menu

Given this long list of problems, we propose to use, in place of walking menus, a single menu containing a tree widget, as shown in figure 13. A tree widget does not suffer of any of the problems listed above: the screen does not get cluttered, the pointer does not need to stay in track, the horizontal expansion is reduced, and the layout of sub-menus doesn't have to be managed, as there are no sub-menus. This solution has the advantage that the user gets a visual representation of the hierarchy in the form of a well-laid-out tree. The main disadvantage is that the user needs to click on buttons to navigate the hierarchy, while walking menus allow to navigate without clicking. However, it is also possible, in principle, to have the tree items expand on hovering, even though we didn't experiment with this solution yet.

3.3.6 Deletion

The requirements specify that it must be possible to delete the selected portion of the query by pressing the Delete key. Providing a consistent way to interact with the GUI using the keyboard is whole dimension of the design of the system that we skipped. Given this premise, we addressed this requirement with mixed results. We designed a mechanism triggering the deletion whenever the Delete key is pressed right after a non-empty selection has been created. When the user performs other actions, the focus is lost, and the Delete key has no effect, even if the current selection is not empty.

In addition to the keyboard-based delete, we provide a graphical tool to delete selected elements. Once a non-empty selection is created, a small button depicting a cross appears at the rightmost corner of the highlighted area, at the right of the substitute button. The query logic subsystem does not allow to delete all the labels of a node which has at least one child, and therefore some selections can't be deleted. This button is red when the selection can be deleted, and blue when the selection can't be deleted, as depicted in figures 14 and 15.

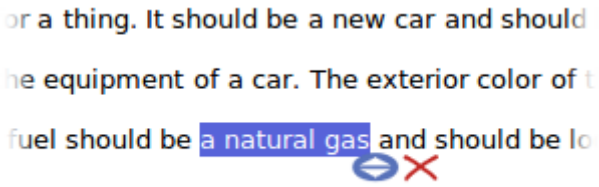

Illustration 14: Active deletion button, next to the substitute button.

3.3.7 Communication with the web service

All the communication with the server, triggered by events on various parts of the GUI, goes through a module called `CommunicationsManager`. This module executes remote procedure calls on the server, handling the responses. For two of these remote procedure calls, we introduced a delay before the request is actually sent to the server, because apparently the execution of a procedure call makes the GUI not responsive for a brief period of time. The two procedure calls are `getExtensions` and `getReplacements`: these two procedures are to be called respectively after a modification of the query (addition, substitution, deletion) and after a selection. If the calls were executed immediately, the interface would not get back to the user until the request is sent to the server. Delaying these calls instead results in an immediate visualization of the result of the previous operation and in a better responsiveness of the interface. After a while, the client sends the request, and after another while, the server's response reaches the client.

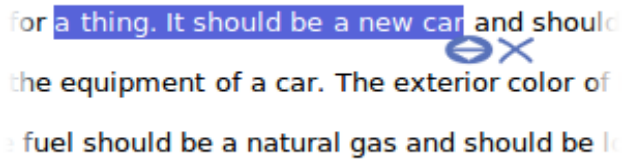

Illustration 15: Inactive deletion button, next to the substitute button.

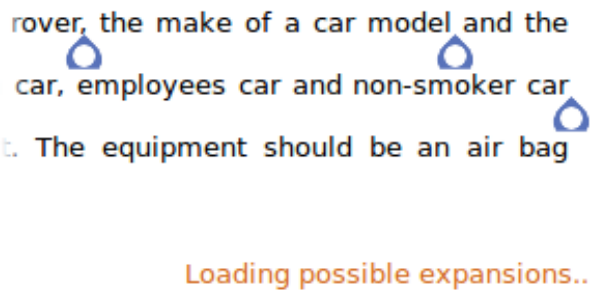

Illustration 16: The GUI signals to the user that the content of the add-menus is being loaded.

The client will use the information contained in these responses to populate the addition and substitution menus. Until the server's response is received, the GUI displays a white circle on the addition and substitution buttons, as depicted in figure 16, which signals that the menu is currently empty. After the server's response is processed, the buttons change their appearance to signal that the content of the menus has been loaded, unless the server's response says that the menu is empty – for example because there are no compatibles and no properties to add. This visual signal is also useful in case the server takes a longer time to compute the requested information.

3.4 Implementation

The implementation of the system resulted in three Java packages containing a total of 35 Java classes. The code of the GWT interface follows the GWT conventions for project organization, resulting in a package for the server code and a package for the client code. We placed the code of the data structures exchanged between the two in a third package. The code is complete with Javadoc documentation.

Chapter 4

Quality & Evaluation

Our plan for quality assessment was very simple. The result of our effort is a system which integrates two existing systems, the query logic subsystem and the natural language generation system, and introduces a user interface system. The focus of our quality assessment is on the following matters:

- The data produced by the query logic subsystem is coherent with the expectations of the natural language generation subsystem
- The data produced by both subsystems is coherent with the expectations of the GWT GUI
- Failures in any of the subsystems do not propagate to the whole application but can be captured and tracked.
- The GUI represents consistently the data provided by the two subsystems
- The GUI provides a smooth interaction, coherent with the behavioural requirements.

We did not focus on quality issues originating within the two integrated subsystem.

To evaluate the quality of the realized system we used the tools provided by GWT, which allow not only to run the entire system both simulating a web server and a browser, but actually to deploy a temporary instance of the application and run locally a web server, and therefore making it possible to access the application from web browsers such as Firefox or Internet Explorer.

In order to assess that the data flowing among the three subsystems is handled consistently, we developed a random query generator. Using the generator, we were able to test the system under different, possibly unpredictable conditions. The purpose of the generator was to verify that different queries are represented correctly, not to verify that different operations on the query are handled correctly. Instead, we tested the application on the GWT browser and on Firefox, performing a range of operations on random generated queries, along with different operations on the GUI which do not involve manipulation of the query itself.

The outcome of these informal tests is that the limitations of the system do not prevent it to qualify for its purpose, the purpose being a technology demo. The issues we detected so far are the following:

- Pop-up menus may increase the length of the host web page when opened. This may cause the browser to show the vertical scroll bar. If the scroll bar was not visible before the pop-up, the layout of the text will change but the change is not reflected by the add buttons, which will not be aligned any more.

- Double double clicks (i.e. a single burst of four clicks) do not trigger a sub-tree selection.

- Clicking in a non-sensitive area of tree widgets within the pop-up menus will move the browser's viewport at the beginning of the menu, and subsequent clicks on the same menu will also have this effect.

These limitations add to the limited keyboard access to the GUI, as described in the design section.

4.1.1 Projected features

The features listed in the requirements are not an exhaustive list of the features we would like to offer to the users of QueryTool. While our system, as it was realized so far, provides all the essential tools for editing queries, there are many issues that need to be addressed before the software can qualify for deployment. The first, essential feature is the management of multiple clients, which we already mentioned. The second issue is support of undo/redo, and the related issue of back/forward and refresh buttons of the browser. A third very desirable functionality is the upload of user ontologies, to be loaded on the fly by the server. All these features were not addressed in this document because of the limited scope of this project, and we foresee that further development of the tool could follow these lines.

Chapter 5

Conclusion

In this document, we introduced QueryTool, a software editor for queries over knowledge bases, and we documented the development of a Web-based graphical user interface integrating the latest version of the QueryTool engine. We presented the requirements that were laid out as a guideline for the realization of the GUI. We discussed the design choices needed to overcome the technical limitations of the development platform, as well as the proposed improvements over the original requirements. The outcome of this project is a software which provides the basic functionality of a graphical user interface, bridging the gap between the user, the query logic subsystem and the natural language generation system. While the software itself is not yet a stand-alone, full-featured query editing facility, it realizes a solid basis for the development of such a system.

Bibliography

[1] Dongilli, P., Franconi, E., & Tessaris, S. (2004). *Semantics Driven Support for Query Formulation*. Paper presented at the 2004 International Workshop on Description Logics (DL 2004), Whistler, BC, Canada.

[2] Kuhn, T. (2009). *How Controlled English can Improve Semantic Wikis*. Proceedings of the Fourth Workshop on Semantic Wikis, European Semantic Web Conference 2009, CEUR Workshop Proceedings, 2009.

[3] Mueckstein, E.M. (1985). *Controlled natural language interfaces: The best of three worlds.* In Proceedings of CSC '85: ACM Computer Science Conference (pp. 176-178). New York: Association for Computing Machinery.

[4] Ranta, A. (2004). *Grammatical Framework: A Type-theoretical Grammar Formalism*. The Journal of Functional Programming 14(2), 145–189.

[5] Tennant, H. R., Ross, K. M., Saenz, R. M., Thompson, C. W., Miller, J. R. (1983). *Menu-based natural language understanding*. Proceedings of the 21st annual meeting on Association for Computational Linguistics, June 15-17, 1983, Cambridge, Massachusetts.