Università degli Studi di Udine
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Corso di Laurea in Tecnologie Web e Multimediali

Bachelor Thesis

Design and implementation of a Language Server for the Jolie programming language

Candidate
Eros Fabrici

Supervisor
Prof. Marino Miculan

Academic Year 2018-2019

Al mio cane,

per avermi ascoltato mentre ripassavo le lezioni.

Acknowledgements

Sed vel lorem a arcu faucibus aliquet eu semper tortor. Aliquam dolor lacus, semper vitae ligula sed, blandit iaculis leo. Nam pharetra lobortis leo nec auctor. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Fusce ac risus pulvinar, congue eros non, interdum metus. Mauris tincidunt neque et aliquam imperdiet. Aenean ac tellus id nibh pellentesque pulvinar ut eu lacus. Proin tempor facilisis tortor, et hendrerit purus commodo laoreet. Quisque sed augue id ligula consectetur adipiscing. Vestibulum libero metus, lacinia ac vestibulum eu, varius non arcu. Nam et gravida velit.

Abstract

Nunc ac dignissim ipsum, quis pulvinar elit. Mauris congue nec leo ornare lobortis. Nulla hendrerit pretium diam nec lobortis. Nullam aliquam laoreet nisl, sit amet facilisis lectus accumsan ut. Duis et elit hendrerit metus venenatis condimentum. Integer id eros molestie, interdum leo sit amet, aliquet metus. Integer fermentum tristique magna, vel luctus neque rhoncus vel. Ut hendrerit et quam et semper. Mauris egestas, odio sed aliquet luctus, magna orci euismod odio, vitae lacinia tellus tellus non lectus. Aliquam urna neque, porta et mattis aliquam, congue sit amet lorem. In ultrices augue sit amet ante vehicula, vitae rhoncus turpis auctor. Donec porta scelerisque eros, at mollis enim imperdiet ut.

Contents

I Introduction **1**

1 Introduction **3**
1.1 Sezione 3
1.1.1 Sottosezione 3

2 Jolie and the Language Server Protocol **5**
2.1 Jolie: Java Orchestration Language Interpreter Engine 5
2.2 Language Server Protocol 5

II Project **7**

3 Analysis of the problem **9**
3.1 Language Server Protocol features 9
3.2 Functional Requirements 9
3.3 Non-functional requirements 10
3.3.1 Complying with the Language Server Protocol specification 10
3.3.2 Distributed 11
3.3.3 Modular 11

4 Design of the system **13**
4.1 Architecture 13
4.1.1 Modules, interfaces, communication 13
4.2 What needs to be implemented 13

5 Implementation **15**
5.1 Choosing a language: Jolie (why) 15
5.2 Implementation details 15
5.2.1 Complying with LSP's JSON-RPC protocol 15

6 Validation **17**
6.1 Met requirements (screenshots) 17
6.2 Unmet requirements 17

7 Conclusions: review, what needs to be done **19**

III Appendici **21**

A Altro capitolo **23**

I

Introduction

1 Introduction

1.1 Sezione

1.1.1 Sottosezione

2

Jolie and the Language Server Protocol

2.1 Jolie: Java Orchestration Language Interpreter Engine

2.2 Language Server Protocol

II

Project

3

Analysis of the problem

In this chapter I will identify the requirements, gathered an analysis of the Language Server Protocol and over a series of meetings with the supervisor.

3.1 Language Server Protocol features

LSP supports six key feature which are:

1. **Code completion**: feature that speeds up the process of coding applications by reducing typos and other common mistakes;

2. **Hover information**: feature that shows information, like the type signature, when the user moves the pointer over an element (such as a function definition);

3. **Jump to definition**: feature that shows the definition of a selected symbol;

4. **Workspace symbols**: displays all the symbols of the workspace, in order to help the user search for elements inside the workspace (such as classes, variables, methods, etc.);

5. **Find references**: given a symbol, this features lists all the project wide references;

6. **Diagnostics**: the tool (e.g. VSCode) flags syntax errors, warnings together with a description.

The protocol supports many other features that are all linked to the above listed. The requirements elicitation phase was based on these main ones.

3.2 Functional Requirements

Functional requirements are statements of services the system should provide, particularly how it should react to particular inputs. From the features analyzed in the previous section, we extracted the following functional requirements, listed in ascending order of importance.

Table 3.1: Functional Requirements

Functional r. No.	Description
FR 1	The server must provide information of eventual programming errors and warnings every time a document is opened/modified.
FR 2	Every time the user starts typing an operation name, the server returns a list of possible completion items that consists of the full operation name and the output port.
FR 3	Every time the user starts typing a reserved word, the server provides a list of possible completion items.
FR 4	Every time a document is opened/modified/closed, the server saves/updates/deletes the information in his memory like the text, URI and a data structure containing all the information regarding the Jolie program (an abstract syntax tree of the *behavior* and data regarding the *deployment*).
FR 5	The server provides the type signature to the client every time the latter sends an hover request.
FR 6	The server shows the operation definition every time it receives a definition. request
FR 7	The server provides a list or hierarchy of symbols of a specific document requested by the client.
FR 8	Information regarding the workspace are sent to the client and updated every time the client requests them (workspace symbols for example).
FR 9	The server resolves project wide references of a given symbol.

3.3 Non-functional requirements

Non-functional requirements are requirements that are not directly concerned with the specific services delivered by the system to its users. They may relate to emergent system properties such as reliability, response time, and store occupancy. Alternatively, they may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems.
Regarding the aforementioned non-functional requirements, the response time is critical for the project, as the server should be able to respond to the client as soon as possible. Nonetheless, this kind of calculations (such as code completion) are not computationally heavy.
Other non-functional requirements, are listed above.

3.3.1 Complying with the Language Server Protocol specification

The Language Server Protocol defines a common language, between a tool and a language server. As a consequence, the latter must be able to recognize the different requests received from the client and send back proper responses that the client is able to interpret and compute.
Following an example: the request-response method `initialize`, that provides the server client's capa-

bilities and it expects a response with the server capabilities, has the following type:

Request type

```
interface InitializeParams {
        processId: number | null;
        rootPath?: string | null;
        rootUri: DocumentUri | null;
        initializationOptions?: any;
        capabilities: ClientCapabilities;
        trace?: 'off' | 'messages' | 'verbose';
        workspaceFolders?: WorkspaceFolder[] | null;
}
```

Response type

```
interface InitializeResult {
        /**
         * The capabilities the language server provides.
         */
        capabilities: ServerCapabilities;
}
```

3.3.2 Distributed

The server must be adapted to the LSP, in order to make it works with different tools that uses the protocol concerned. Therefore, it must be **editor-agnostic** and it should be designed and implemented in order to be deployed both locally and in a different machine with respect to the clients, so it can interact with different tools at the same time. Consequently the best communication method to use in this case is the socket.
Regarding the communication protocol and data format, LSP supports only JSON-RPC 2.0, as mentioned in Chapter 2, therefore also the server must support it.

3.3.3 Modular

Not every language server can support all features defined by the protocol. LSP therefore provides, the previous mentioned, **capabilities**. A capability groups a set of language features. A development tool and the language server announce their supported features using capabilities. For instance, the server announces that it can handle the `textDocument/hover` request, but it might not support the `textDocument/references` request. Similarly, a development tool announces its ability to provide `textDocument/didChange` notification when a document is modified, so that a server can compute textual edits to format the edited document.
On account of this, the server must be designed in order to easily add new capabilities or improve the existing ones. After implementing a new feature, the programmer just need to modify the server capabilities thus the client can start sending requests regarding the newly activated capability.

Table 3.2: Non-Functional Requirements

Non-functional r. No.	**Description**
NFR 1	The server must support the JSON-RPC protocol.
NFR 2	The server must be designed to be modular.
NFR 3	The server must be able to respond to a client request as soon as possible.
NFR 4	The server must work both when deployed in a different machine and when deployed in a local machine, with respect of the client location. As a consequence, it has to support the socket channel of communication.
NFR 5	The server if distributed, must be able to handle more clients sending multiple requests at the same time.

4 Design of the system

4.1 Architecture

4.1.1 Modules, interfaces, communication

4.2 What needs to be implemented

5 Implementation

5.1 Choosing a language: Jolie (why)

5.2 Implementation details

5.2.1 Complying with LSP's JSON-RPC protocol

The JSON-RPC 2.0 is a remote procedure call protocol that uses the JSON, a lightweight data-interchange format which is language independent. As stated in Chapter 2, an LSP message supports only two headers (*Content-Lenghth* and *Content-Type*), while the Jolie protocol runs over HTTP. This incompatibility led to a complex rework on the aforementioned Jolie protocol in order to meet the FR 1.
Further details can be read in Chapter 5.

6 Validation

6.1 Met requirements (screenshots)

6.2 Unmet requirements

7

Conclusions: review, what needs to be done

III

Appendici

A

Altro capitolo

Summary