Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol

Hendrik Bünder[1(✉)] and Herbert Kuchen[2(✉)]

[1] itemis AG, Bonn, Germany
buender@itemis.de
[2] ERCIS, University of Münster, Münster, Germany
kuchen@uni-muenster.de

Abstract. In model-driven software development (MDSD) projects, frequently domain experts and developers work together on the same model. However, they have quite different preferences concerning tools for working with a model. While developers require a powerful integrated development environment (IDE), domain experts are overwhelmed by the amount of functionality of an IDE and its confusing user interface. They prefer a simple editor, often provided as a web application, which does not require a local installation. Currently, both stakeholders typically agree on a common tool, which is frustrating for at least one of them. The Language Server Protocol (LSP) is a standard that aims to include language smarts into simple editors without turning them into IDEs. Originally, it has been designed for programming languages. In the present paper, we will give evidence based on a case study and a corresponding SWOT analysis that it is even more beneficial for a textual domain-specific language (DSL) as it is often used in MDSD. We will focus on the language workbench Xtext which supports the LSP. In particular, we will investigate how the LSP can be used to integrate a DSL into different development tools (editors and IDEs). Supplementing the SWOT analysis, we have also evaluated the practical relevance of the LSP.

Keywords: Textual domain-specific languages · Model-driven development language server protocol · Case study

1 Introduction

Domain-specific languages (DSLs) enable the efficient creation of complex software systems by allowing to describe the concepts of a particular domain in a typically declarative and semantically rich way. Due to their high level of abstraction, the documents written in a DSL are typically called *models* rather than programs. Such models are captured in an either textual or graphical notation. In model-driven software development, the models are automatically transformed into documents on a lower level of abstraction. Frequently, they are directly

S. Hammoudi et al. (Eds.): MODELSWARD 2019, CCIS 1161, pp. 225–245, 2020.
https://doi.org/10.1007/978-3-030-37873-8_10

transformed into the source code of the selected programming language. One key factor to success for a domain-specific language is a mature editor support that includes language smarts such as syntax highlighting, validations, and code completion.

Focusing on textual DSLs, there is a variety of potential editors. On one side of the scale, simple text editors such as VI [16] or Microsoft Notepad can be used. While these editors focus on simple and fast text changes, they do not support any language-specific smarts. On the other side of the scale, sophisticated integrated development environments (IDE) come with a feature rich, language specific editor. Additionally, they provide an ecosystem for analyzing, running, testing, and debugging source code.

For creating mature editor support for DSLs, language workbenches such as MPS [5], Spoofax [13], and MontiCore [15] offer many features. Based on the integration into IDEs such as Eclipse [33] or IntelliJ [12], language editors can be created quickly by semi-automated processes. Although this integration enables the fast creation of language specific editors, it also couples the editor to the host IDE.

Since DSLs are often edited by both, technical and non-technical users, conflicting requirements arise. The technical users prefer the DSL editor to be deeply integrated into the IDE that is best for their use case. For example, a developer implementing an application in JavaScript [23] might prefer Visual Studio Code [18]. Alternatively, a developer implementing an application in Java [1] or C++ [29] might prefer the Eclipse IDE. Since a DSL might describe different parts of an application or provide multiple generators, it should be easily integrated into different IDEs. On the other hand, DSLs have to provide an editor that is optimized for non-technical users. In contrast to developers, non-technical users are distracted by the overloaded user interfaces of IDEs and prefer simple editors. Yet, they should be supported by all the language smarts that are key factor to success for a DSL. In addition, non-technical users favor web-based editors where no additional installation is required.

The stated requirements cause a lot of effort for toolsmiths, since they have to provide editors for different IDEs, text editors, and browsers. Consequently all language smarts such as code completion or syntax highlighting have to be implemented in the programming language used by the IDE or tool in which the editor should be integrated. Since this is not an economically reasonable strategy, toolsmiths tend to support only a single IDE. Thereby, developer productivity decreases due to the fact that the supported IDE for the DSL might not be the ideal IDE for enhancing, testing, running, and debugging the source code generated from the DSL. Additionally, non-technical are forced to used IDE to collaborate instead of a more concise web browser based editor.

The Language Server Protocol (LSP) [22] introduced by Microsoft, RedHat and Codenvy decouples the language smarts from the actual editor integration. While intended to provide editor support for general purpose programming languages, it can equally well be used for DSLs. The LSP is supported by the Xtext language workbench. DSLs built upon the Xtext language workbench support

all LSP features including multi-IDE, multi-Editor and browser integration. In order to exemplify the steps required to integrate an arbitrary DSL into multiple editors, we have conducted a case study. In addition, we have performed an analysis of strengths, weaknesses, opportunities and threats (SWOT) of leveraging the Xtext LSP integration. The present paper is an extended version of a preceding conference paper [4]. In order to supplement the SWOT analysis, we have now also analyzed the utilization of the LSP. The analysis focuses on open source github projects that have been implemented utilizing Xtext and the LSP. In order to also get an idea of the usage of the Xtext and LSP integration in proprietary closed source projects, the Eclipse textual modeling framework (TMF) forum was analyzed in order to evaluate, how extensively the topic is being discussed.

The present paper is structured as follows. First, Sect. 2 gives an overview of the LSP. In Sect. 3, the Entity-DSL implemented in the context of the case study is described. Further, we outline the results of the conducted SWOT analysis identifying the potential of the LSP in combination with the Xtext language workbench. Section 4 presents our observations from analyzing open source projects utilizing the Xtext and LSP integration. It also presents our results from examining the Eclipse TMF (Xtext) forum. Section 5 summarizes and discusses our results, before we describe related work in Sect. 6. Finally, we conclude in Sect. 7.

2 Overview of the Language Server Protocol

This section describes the language server protocol [22]. Readers who are familar with it, can skip this section. A more detailed presentation can be found in [4].

In order to enable the comfortable editing of the overwhelming variety of existing programming languages, editor- and IDE-vendors face the challenge to support corresponding sophisticated *language smarts* for them. Well-known examples of language smarts are *goto definition*, *hovers* and *code completion*. At the same time, the number of editors and IDEs is growing steadily. Therefore, providing editor support for all languages in all available integrated development environments and editors creates an $O(n \times m)$ complexity.

Through separation of language smarts and editor integration, the LSP strives for mitigating this issue. By providing a server process in charge of language smarts and a client process integrating the results into the specific editor, both parts become independent. The implementation of language server and development-tool extension is supported by mature Software Development Toolkits (SDK) that encapsulate low-level communication leading to the efficient creation of client and server processes. By communicating through the standardized LSP, development tool and language server can be implemented separately even in different programming languages. The separation of concerns in combination with the feature rich protocol decreases the complexity to $O(n + m)$ [28].

Since the introduction of the LSP in 2016, more than 50 language servers have been implemented covering languages such as Python, Ruby, and Java.

Additionally, development-tool extensions are available for more than 10 IDEs and editors such as VSCode or Eclipse [28]. The advantages of the LSP are not only useful for programming languages but also for textual domain-specific languages, since they require the same language smarts. Currently, only the Xtext language workbench for developing DSLs supports the LSP. Thus, we use this workbench in the sequel.

The LSP architecture specifies a client and a server process that communicate through the standardized LSP. The protocol specifies capabilities that represent possible messages exchanged between client and server process [22]. In order to provide the right features for the right editor, development tool and language server exchange supported `Capabilities` during the initial communication. In order to enable the extension of the LSP for a specific language, the LSP offers `ExperimentalCapabilities`. Instantiations of these `Capabilities` are not part of the protocol. Thus, development tool and server have to be extended manually to support these custom features.

The language server will ignore missing `Capabilities` and expect these language smarts not to be supported by the development tool. For the requested `Capabilities`, the language server answers with the `Capabilities` it supports [20].

While the number of features supported by the LSP is increasing steadily, there are six key features [28], namely *code completion* (supported by 90% of the language-server implementations), the *hover* feature (see Fig. 1; 88% support), *goto definition* (83% support), *workspace symbol* (causing a search for the given symbol in the whole workspace; supported by just 60% of the servers), *find references* (63% support), and *diagnostic* (causing a notification back to the development tool; 78% support). The Xtext language workbench supports all these six key features.

Fig. 1. Theia editor integration: hover feature [4].

The LSP has its limitations. For instance, it solely focuses on integrating language smarts. Well-known IDE features such as building, testing, running and debugging source code remain IDE-specific. Other protocols care about such features [3,14]. Comparing the development-tool extension capabilities to features provided by native IDE integration reveals additional shortcomings. For example leveraging the native Eclipse integration of Xtext DSLs provides mature UI features such as outline view, project-creation wizards, or type hierarchies [32]. The trade-off for integrating language smarts into multiple editors is the loss of such advanced user-interface features.

Language servers and the corresponding development-tool extensions can either be used by cloning the respective github repository or by installing an extension through a market place of the respective IDE or editor.

3 Case Study: Language Server Protocol with Xtext DSLs

A case study was conducted to quantify the efforts required to provide a language server and two development-tool extensions for a simple example DSL. The language server is supposed to provide the six key features as explained in Sect. 2. The IDE integration is exemplified by providing a development-tool extension for Theia as well as for Eclipse.

Figures 2 and 3 show the expected integration into Theia and Eclipse, respectively. The grammar of the example DSL will be explained in the next subsection. As shown by the two figures, both editors provide content sensitive proposals determined by the language server. However, the keyword highlighting is not supported by the LSP. Thus, it had to be implemented for each IDE separately.

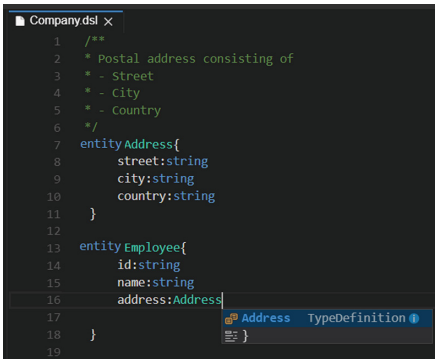

Fig. 2. Theia Editor Integration [4].

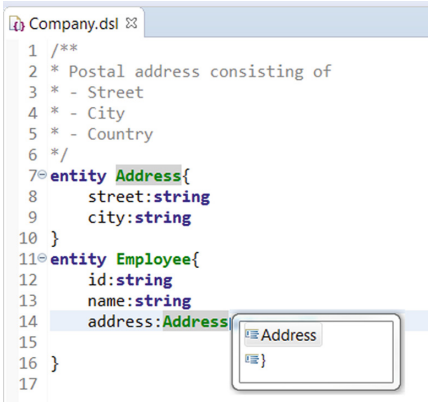

Fig. 3. Eclipse Editor Integration [4].

A detailed description on how the language server and the development tools have been implemented will be explained in the next subsections.

3.1 Language Server Implementation of an Entity-DSL

Our simple example DSL allows to describe entities with operations and properties. Its simplified grammar in EBNF format is shown in Listing 1.1. Readers, who are familar with Xtext, can find the original syntax description in Xtext format in the Appendix.

A `model` consists of a sequence of `entities`. An `entity` has an identifier as its name and it may inherit from another entity (keyword `extends`). Moreover, it may have an arbitrary amount of `features`, i.e. `properties` and `operations`. An `operation` has an identifier as its name, an optional list of `parameters` and a `return type`. A `property` has an identifier as its name and a `type`. A `type` can reference an `entity` by its name or it can be a `primitive` type, i.e. `string`, `number`, or `boolean`.

```
Model:
   Entity*;
Entity:
   'entity' ID ('extends' ID)? '{' Feature* '}';
Feature:
   Property | Operation;
Property:
   ID ':' Type;
Operation:
   'op' ID '(' (Parameter (',' Parameter)*)? ')'  (':' Type)?;
Parameter:
   ID Type;
Type :
   Primitive |  EntityReference;
EntityReference :
   ID;
Primitive :
   'number' | 'string' | 'boolean';
```

Listing 1.1. EBNF rules for the Entity-DSL. A * suffix indicates an arbitrary number of repetitions; a ? suffix indicates an optional element. Vertical bars separate alternatives. Parentheses () are used to limit the sope of a * or ? suffix. Symbols in quotes are terminal symbols.

The original Xtext version of the mentioned DSL syntax serves as input for the generators provided by the Xtext language workbench that create the corresponding workbench for the DSL, including abstract syntax tree, parser, linker, formatting, etc. The LSP for Java (LSP4J) framework is utilized by the generated workbench to create requests, responses and notifications in accordance with the LSP. By encapsulating the low-level JSON communication through a Java API [24], the features of the LSP are available to the generated Java-based language-server implementation generated from the Xtext grammar. Due to the powerful generators included in the Xtext language workbench, all six key features are fully generated. There is no manual implementation required.

3.2 Building a Development-Tool Extension for the Theia IDE

As desktop and cloud IDE, Theia integrates different languages based on the LSP. In addition, Theia separates backend processes and the graphical user interface. Thereby, the frontend can either be a browser or a desktop application,

while at the same time the backend process is executed locally or remotely on a cloud-based infrastructure [30].

Language servers in Theia are provided through a backend extension, that is capable of configuring the language server. Theia uses dependency injection mechanisms [8] to provide extensions. Listing 1.2 shows the code required to inject the Entity-DSLs backend extension. Here and in the following listings, we do not expect the reader to understand every detail. The listings rather serve to demonstrate that the implementation requires very limited effort.

The `BaseLanguageServerContribution` class from the LSP-specific Theia framework is extended by the `EntityDslContribution`. As shown in Listing 1.2, there are two working modes for the language server. First it can run as local sub-process on the same machine. Alternatively, it can be executed remotely using web-socket communication [7]. The client-specific libraries encapsulate the details for establishing the connection.

```
@injectable()
class EntityDslContribution extends BaseLanguageServerContribution {

 start(clientConnection: IConnection): void {
  let socketPort = getPort();
  if (socketPort) {
   this.connectToRemoteServer(clientConnection,socketPort)
  } else {
   this.connectToLocalServer(clientConnection)
  }
 }
}
```

Listing 1.2. Registering a Language-Server Extension in Theia.

After the backend extension has been registered, language smarts can be provided through the Theia editor. However, the keyword highlighting as shown in Fig. 2 has to be specified by a frontend extension. Within this extension, the keywords are specified and it is determined, how they should be highlighted. Moreover, the frontend extension contains information about the file extensions handled by the given language server. Listing 1.3 exemplifies how the "globPatterns" method of the `EntityDslClientContribution` must be overwritten to bind the language server to the file extension.

```
@injectable()
export class EntityDslClientContribution extends
     BaseLanguageClientContribution
{
 protected get globPatterns() {
  return ['**/*.dsl'];
 }
}
```

Listing 1.3. Binding the ".dsl" file extension to the language server.

By providing a backend and frontend extension as shown above the Theia IDE can provide language smarts for the Entity-DSL as shown in Fig. 2.

3.3 Building a Development-Tool Extension for the Eclipse IDE

Xtext as language workbench and also the language tooling created by the workbench are heavily integrated into Eclipse through its plugin mechanism. Thereby, Xtext DSL editors within Eclipse have a lot of default features included, such as outline views or type hierarchies. Yet, for the purpose of the case study, the Entity-DSL should provide an editor for the Eclipse IDE based on the LSP. Fortunately, the language server for the Eclipse (LSP4E) framework [6] encapsulates low-level communication through a Java-API.

The basic mechanism for providing extensions to the Eclipse ecosystem is by utilizing its plugin architecture [33]. The LSP-based editor support for the Eclipse IDE will also be provided by a specific plugin that extends the LSP4E plugins extension point [10]. The DSL-agnostic extension point "org.eclipse.lsp4e.languageServer" provided by the LSP4E plugin is extended as shown in Listing 1.4. It delegates the implementation to the `EntityDslLanguageServerClass`. This class handles all DSL-specific features for the Entity-DSL.

```
<extension point="org.eclipse.lsp4e.languageServer">
   <server id="org.eclipse.lsp4e.languages.dsl"
      class="org.eclipse.lsp4e.languages.dsl.EntityDslLanguageServer"
      label="Entity-DSL Language Server">
   </server>
</extension>
```

Listing 1.4. Registering a Language Server Extension in Eclipse.

The implementation of the `EntityDslLanguageServer` that extends the LSP4E framework class `ProcessStreamConnectionProvider` is shown in Listing 1.5. The language server is started as soon as a file with the file extension ".dsl" is opened. The `createLauncherCommand` method returns the command required to start a sub-process running the language server. Next, the `ProcessStreamConnectionProvider` computes the working directory for the language server.

```
public class EntityDslLanguageServer extends ProcessStreamConnectionProvider
    {
   public MyDslLanguageServer() {
      setCommands(createLauncherCommand());
      setWorkingDirectory(workingDirectory());
   }
}
```

Listing 1.5. Entity-DSL Extension.

If the language server is running on a remote host, the `ProcessOverSocketStreamConnectionProvider` class from the LSP4E plugin can be extended. Thereby, the communication to an already running language server on a remote host can be realized via a web-socket infrastructure.

```
public class EntityPresentationReconciler extends PresentationReconciler {
  private Set<String> keywords = new HashSet<>(Arrays.asList(new String[] {
     "entity", "extends", "op", "string", "number", "boolean"}));
}
```

Listing 1.6. Keyword Definition for Syntax Highlighting.

As for the Theia extension, the correct highlighting within the editor needs to be specified by providing an additional class. For the Eclipse IDE the `PresentationReconciler` is extended by the `EntityPresentationReconciler` that defines syntax highlighting for comments and keyword in the Entity-DSL as shown in Listing 1.6.

While integrating LSP-based editors into Eclipse and Theia is comparable on a conceptual level, both require tool-specific implementations. Defining syntax highlighting and the start-up of the language server are the main concerns of both development-tool specific extensions. Nevertheless, regarding programming language, architecture, and API both development tools are significantly different.

3.4 Experimental Results of Implementing the Entity-DSL

To quantify the effort of integrating a DSL editor into two different IDEs leveraging the LSP, the time required to implement a language server and the respective development-tool extensions was measured. The first part of the case study was the language-server implementation based on the EBNF-like grammar of the Xtext language workbench. Due to the fact that parser, linker and all required classes for the six key features of the language server as specified in Sect. 2 have been generated completely, the language server implementation could be finished in about two hours (Table 1).

Table 1. Effort for implementing the Entity-DSL [4].

Task	Time (in minutes)
Language Server Implementation	127
Theia Editor Integration	414
Eclipse Editor Integration	317

The implementation of the Theia integration required more manual effort, since a backend and frontend extension had to be provided. Since both required manual implementation, configuration and testing the overall effort was about 7 h. Due to some conceptual similarities between the Theia and Eclipse extension, implementing the Eclipse integration took only a little bit more than 5 h. In general, the fast implementation of all three parts was mainly enabled by the sophisticated SDKs that encapsulate the low-level communication.

In addition to the case study, the efforts spent on providing three extensions in context of the Yakindu Solidity Tools project were analyzed. The project provides an integrated development environment for Ethereum/Solidity based smart contracts [37].

As shown by Table 2, the Yakindu Solidity Tools project provides three language server based editor integrations, namely for Theia, Visual Studio Code,

Table 2. Effort for implementing Multi Editor Support for the Solidity IDE.

Task	Time (in minutes)
Language Server Implementation	350
Theia Editor Integration	835
VSCode Editor Integration	725
Atom Editor Integration	680

and Atom. The creation of the language-server implementation took around 350 min in total. All three editor extensions required a little bit more than a day of work with 835, 725 and 680 min of effort, respectively. Since the Yakindu Solidity IDE supports more keywords and DSL-specific functionality the implementation took more effort for all four compared to the Entity DSL.

It is clear that the observations from two projects cannot achieve statistical significance. In order to achieve that we would need to investigate hundreds of projects, which is not realistic taken into account the duration and costs of such projects. However, observations from the considered projects nevertheless give a rough idea of the effort required to provide multi-editor support for a DSL. They indicate that providing support for multiple editors can be provided with moderate effort using the Language Server Protocol and that this approach is substantially easier and faster compared to a native integration.

3.5 Analysis of the Potential of the Language Server Protocol

Based on the case study, a SWOT analysis [11] has been conducted in order to get a comprehensive view on the potential of the LSP for multi-editor support of DSLs. By analyzing strengths and weaknesses as well as opportunities and threats, the potential of the LSP is examined internally and with respect to the external environment. By investigating the strengths of the LSP, advantages increasing the market penetration will be identified. In addition, weaknesses and threats will be determined that must be eliminated or mitigated to ensure the success of the LSP.

The analysis starts with evaluating the internal strengths of the LSP. By specifying a standardized protocol for development tool and language server to communicate, language smarts can be implemented once and integrated multiple times. The LSP consequently establishes a solid foundation for multi-editor integration not only for general-purpose programming languages but also for domain-specific languages. Reducing the overall integration complexity to $m+n$ rather than $m \times n$, multiple editors can be integrated efficiently.

The lightweight JavaScript Object Notation (JSON) format used for exchanging messages in combination with the JSON-RPC protocol ensures compact payloads. By reducing the amount of data sent back and forth between language server and development tool, efficient communication over a network is possible.

Although both processes are currently running on a single machine, the JSON-RPC protocol enables a good user experience when editing text files.

Being able to support multiple editors is especially advantageous for domain-specific languages. Domain models might be specified collaboratively by technical and non-technical team members. While technical users might prefer a certain IDE, such as Eclipse or VSCode, in which the editor needs to be integrated seamlessly, non-technical users typically prefer a browser-based interface to a simple editor such that no additional tool needs to be installed. This reduces the distribution and maintenance costs. Moreover, browser-based editors can be designed to be more concise and clearly structured compared to IDEs that are confusingly overloaded with buttons and menus from a non-technical user's perspective.

The LSP provides a flexible approach to extend the protocol using custom capabilities. Thereby, toolsmiths can provide language- or editor-specific language smarts. In contrast to the LSP capabilities, the editor integration has to be built manually. Consequently, these smarts might not be supported by all editors. In the long run, the LSP might adopt custom extensions that provide value for the majority of programming languages.

In contrast to the strengths of the LSP, the following weaknesses can be identified. Since the LSP is founded on the assumption that one language server serves one development tool, the number of parallel language servers quickly rises in multi-language scenarios. Editing three or more languages in parallel is a valid scenario. In such a situation, the programming language to implement the language server has to be chosen with respect to its memory footprint. Three language servers implemented in Java will all run in their own Java Virtual Machine (JVM) on a single machine allocating approximately 3 GB of memory. Taking into account the size and complexity of the different languages, a poor user experience might result. Yet, there are more memory efficient SDKs, e.g. utilizing JavaScript, to implement the language server.

Since every development-tool extension requires its own implementation in the programming language suitable for the development tool, the toolsmiths have to decide which editors to support. Therefore, DSL-editor providers require knowledge about providing extensions for Theia, VSCode, or Eclipse. In addition, every extension has to be tested separately to ensure matching user experience in terms of keyword highlighting or validations.

By separating language-server implementations on a conceptual level, they become independent and scalable. Yet, there is no communication between different language servers, leading to continuous re-implementations of basic features. In contrast to native IDE integration, language servers now have to implement functionality, e.g. to resolve classes from a classpath, that otherwise could have been reused.

The language server was built in order to provide sophisticated editor support in multiple editors for textual programming languages. Since the Xtext language workbench supports the LSP, it can also used for DSLs. The combination of Xtext and LSP works seamlessly, because Xtext is also based on

text files. However, DSLs can also be implemented using projectional editors such as IntelliJ's Meta Programming System (MPS) [9]. Further, there is a variety of graphical DSLs to specify domain models. Since the language server does not support any of the two approaches, powerful alternatives or extensions to textual domain-specific languages are ignored.

In addition to internal strengths and weaknesses, the SWOT analysis also analyzes external factors in form of opportunities and threats. The first opportunity lies in the spread of domain-specific languages which are by now widely adopted in research and practice. While language workbenches, such as Xtext or MPS, support the fast implementation of sophisticated DSLs, the editor funtionality is often bound to a single IDE. Yet, developers demand different IDEs for different languages or projects. Moreover, non-technical users refuse to interact with IDEs after all. A holistic DSL that enables collaboration of technical and non-technical users needs multi-editor support in order to enable the efficient creation and maintenance of domain models. In an industry where the number of programming and domain-specific languages as well editors and IDEs is rising steadily, the LSP provides a solution to efficiently provide multi-editor support.

One important requirement for multi-editor support is to enable non-technical users to engage in the creation of domain models. Since DSLs are describing abstract technology agnostics constructs, such as insurance contracts or banking products [34], non-technical users need to be able to engage in the creation. While DSLs integrated into heavyweight IDEs are confusing due to the amount of menus, buttons and toolbars, web-based editors are perceived as more concise and clearly structured. Moreover, web-based approaches eliminate additional tool installations. In addition to an IDE integration, the LSP enables the parallel use of web-based editors such as Monaco [21], that can be easily integrated into web-browsers. Providing a browser-based editor lowers the barrier for non-technical users and fosters collaboration on domain models.

Our SWOT analysis closes with examining threats regarding the LSP. Since the LSP is the first standardized approach to provide multi-editor support, there are no technological alternatives. On the one hand, this makes it unlikely that the LSP will be replaced by a competing approach. On the other hand, the investments in development tools and language servers depend on the LSP. These investments would be lost, if the LSP was discontinued.

Providing high-level DSLs quickly generates the demand for multi-notation support including tables, formulas, or charts. Projectional language workbenches such as MPS offer such functionality, but are currently limited to supporting only a single editor. In order to gain more market share, the LSP needs to be enhanced to handle DSLs based on projectional editors.

Browser-based multi-notation language workbenches provide web editors for DSLs typically on a high-level of abstraction. WebGME [17] is a famous example from this category that provides a browser-based editor to specify browser-based DSL editors. In general, multi-notational editors provide an alternative to parser-based languages utilizing the LSP. Yet, approaches like WebGME lack support for native IDE or multi-editor integration.

4 Utilization of the Language Server Protocol

As mentioned above, a relevant threat when relying on the LSP is that the LSP could be abandoned, when it is not used widely enough. Thus, as a supplement to our SWOT analysis, we will now investigate the utilization of the LSP in practice. First, open-source projects built upon the LSP and Xtext were analyzed. Second, the Eclipse TMF (Xtext) forum and its LSP-related threads were examined. While the first indicates the actual market penetration of LSP-based Xtext editors in the open-source community, the second demonstrates how intensive the topic is discussed in the Xtext community, including closed-source projects.

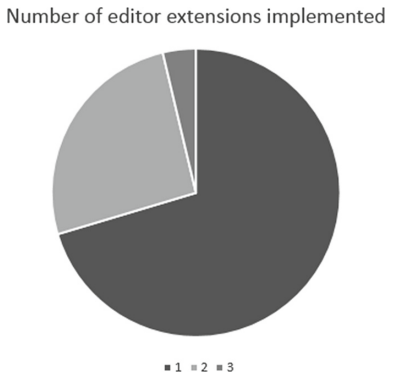

Fig. 4. Number of editor extensions implemented.

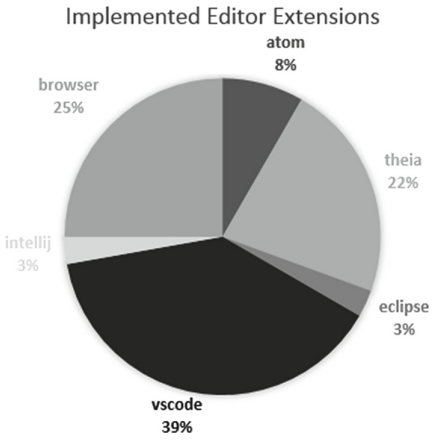

Fig. 5. Kind of editor integration.

4.1 LSP-Based Xtext Solutions in Practice

The first part of the analysis started with crawling all the 98 found open-source github repositories utilizing the LSP for academic or commercial purposes. Projects using Xtext and LSP integration solely for demonstration or testing purposes have been excluded. A manual inspection revealed that out of the 98 search results only 51 had an actual implementation of the language server and only 27 of those had at least one development-tool extension implemented.

As shown in Fig. 4, about 70% of all projects only provide one editor extension. Out of the 27 projects, only 7 implement two development-tool extensions which is equal to 25%. Only the Solidity-IDE project [37] provides three editor extensions namely for Atom, Theia and Visual Studio Code.

Figure 5 shows the different editor extensions implemented using the LSP for the 27 projects. Since the editor integration for Eclipse is more effective using the native integration, only 3% of the projects provide an Eclipse extension using the LSP. Further, only 3% of the analyzed projects provide an extension for IntelliJ. In contrast to IntelliJ and Eclipse, Atom is rather an editor than an IDE. It is only supported by 8% of the projects. The majority of 39% provides an extension for Visual Studio Code. Additionally, 29% of the examined projects provide a browser extension based on the LSP. The Theia IDE that was also used in the case study mentioned above is enhanced by 22% of the projects.

As depicted in Fig. 6, there are only 6 projects that provide their editor extensions through at least one of the available marketplaces. The y-axis shows the cumulative number of downloads over all available stores or marketplaces for the given projects. The *JHipster* IDE is the clear outlier with more than 20,000 downloads; most of them through the VSCode marketplace. The projects *yang*, *plexus-interop*, and *ex.xtext.lsp* have been downloaded between 5,000 and 1,700 times. While for the first two, the downloads stem from the VSCode marketplace only, the latter has been downloaded more than 3,000 times from the IntelliJ marketplace. *ck2xtext* and *scila* have been downloaded 707 and 98 times, respectively, from the VSCode marketplace.

We can conclude that the LSP is significantly used in practical open-source projects, although its full potential has not yet been leveraged.

4.2 Eclipse TMF (Xtext) Forum Analysis

In order to analyze to which extent the LSP feature is discussed by the Xtext community, the Eclipse TMF (Xtext) forum was examined. The forum is the main point of discussions about features and best practices regarding Xtext. Since the LSP support is part of Xtext since version 2.11.0 that was released February 1, 2017, the analysis starts from the first quarter of 2017. The first quarter of 2019 marks the end of the analysis period that is divided quarterly. For the given time period a total number of 981 threads were discussed in the Xtext forum. The analysis was done based on the subject and creation date of each thread. In order to count all topics regarding the LSP, all subjects mentioning the words *Language Server Protocol*, *language server*, or *LSP* were counted. In

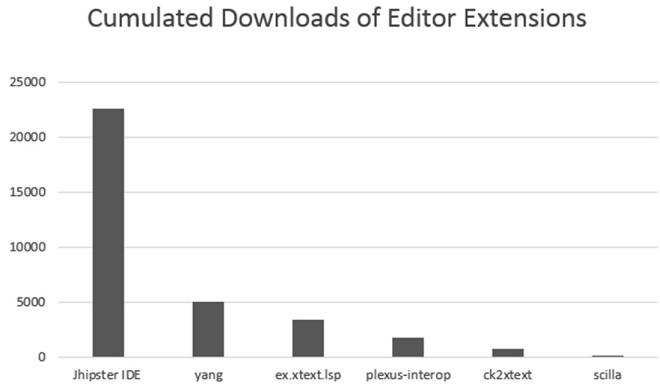

Fig. 6. Cumulative Downloads of Editor Extensions.

addition, subjects using the words *vscode*, *Visual Studio Code*, or *Theia* were counted as LSP related. The analysis focuses on the subjects of the threads rather than on the entire discussion. Thus, we only take those questions into account that specifically focus on the topic.

In order to compare the search results for LSP-related topics to other important topics in the forum, the subject headlines were analyzed to find the two-word phrases, which occurred most frequently. Table 3 shows the six two-word phrases most frequently occurring in the subjects of the threads in descending order of occurrences. While the first three show no direct relation to a specific Xtext topic, the last three are all related to specific features of the Xtext workbench.

Based on the results of the text analysis, the subjects of the threads were analyzed for the questions regarding content assist, cross reference and xtext grammar. Since the topic content assist is synonymously named proposal provider or code assist, these terms were included in the search query. For the other two topics, no additional terms were required. Yet, for questions regarding Xtext grammar subjects only mentioning the word grammar without Xtext were also counted. All four analyses were done in a case-insensitive way.

Figure 7 shows the results for the four topics. The first questions regarding the LSP Xtext integration were asked in the second quarter and third quarter of 2017. While there are two questions on average per quarter regarding LSP, there are also quarters such as the fourth quarter of 2017, where no questions were asked. Although being only the third most often occurring topic based on Table 3, "Xtext grammar" is questioned most often with an average of more than 7 questions per quarter. "Content assist" is questioned regularly in the Xtext forum and with an average of four question per quarter more often than the LSP topic. The fourth topic related to cross-references is discussed in less than two question per quarter on average.

Although, the absolute number of LSP-related topics in the considered form is not impressive, these topics are nevertheless among the most active topics

Table 3. Two-word phrases in thread subjects.

Two-word Phrase	Occurrences
how to	76
xtext 2	40
in xtext	35
content assist	26
cross reference	18
xtext grammar	14

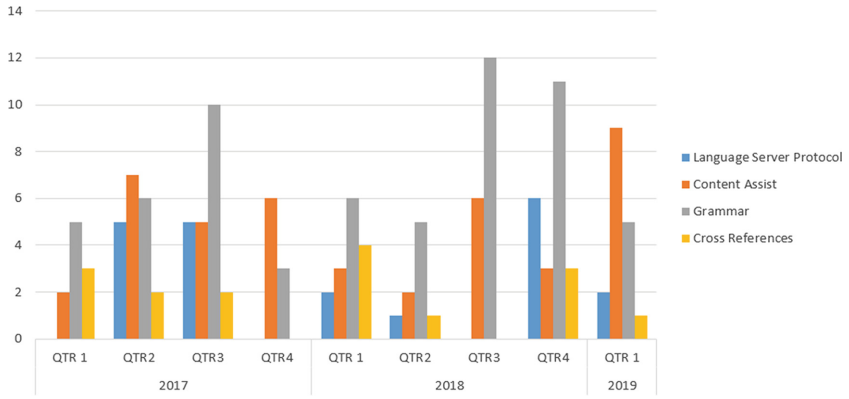

Fig. 7. Questions asked per topic and quarter in the Xtext TMF (Xtext) forum.

in that forum. Thus, the topic is relevant, although its importance has still a potential to rise.

5 Discussion

The LSP offers a standardized mechanism that enables an easy integration of language smarts into different editors and IDEs. As the case study has shown, the Xtext language workbench integrates this feature to provide arbitrary DSLs for different development tools. While the integration is slightly specific for each tool, the language server parts for the key features are generated completely. Thereby, LSP-based DSL editors can be created quickly as Subsect. 3.4 has shown. Although the results from the case study are supported by the findings from the real world open source project Yakindu Solidity Tools, additional projects should be analyzed to further support our claims. The SWOT analysis conducted as second part of the case study shows that the LSP addresses the right requirements, since the number of DSLs and development tools is steadily increasing. Yet, the Xtext and LSP integration is limited to textual DSLs, hence ignoring sophisticated alternatives such as MPS-based projectional DSLs.

However, our analysis of the utilization of LSP-based DSLs in real world projects has revealed that the full potential of the LSP has not yet been leveraged. There are currently 27 open source projects utilizing Xtext and LSP. Only six of them are listed on an official marketplace. Projects such as JHipster that are utilizing Xtext and LSP and the corresponding market places to distribute the development-tool extension can be seen as early adopters. In addition, we know of several large DSLs that are implemented as company-specific closed source solutions. In order to get an idea, of the overall importance of the LSP, including closed-source projects, we have analyzed the amount of questions asked about the language server and Xtext integration. As shown in Fig. 7 there are between four and six questions asked per quarter about the LSP. Comparing it to the other topics that are available in the Xtext forum, the number of LSP-related questions is not impressive but steady.

According to our experience, especially large companies using Xtext have a deeply integrated Eclipse-based tool chain. However up to now, their main focus remains on Eclipse as primary IDE. Therefore, they prefer the powerful native Eclipse integration, which has turned out to be frustrating for non-technical users such as domain experts.

On the road to multi-editor support for arbitrary DSLs, the Xtext language workbench integrating the LSP marks an important milestone. Separating language smarts and editor functionality enables an efficient distribution of editor integrations for a broad variety of editors and IDEs. In addition, the standardized protocol establishes the minimal features for sophisticated editor support as listed by the community-driven LSP-site [28]. Moreover, the Xtext-LSP integration can increase the efficiency of model-driven software development with DSLs by integrating the language smarts into the IDE or editor that is most appropriate for running, testing, and debugging the generated code.

Since the LSP-based DSLs also support web-editors, they can be a valid alternative to other approaches. By integrating LSP-based Xtext DSLs into web-browsers, also non-technical users can interact with the domain model. The results from Sect. 4 substantiate this. They show that 25% of the LSP-based DSLs provide a browser integration. By providing a web-based editor, technical and non-technical team members can contribute to the domain models using their favorite editor.

6 Related Work

Textual Domain-Specific Languages can be divided into two categories, namely internal and external. While internal DSLs are embedded into a host language [34], external DSLs have their own tool support [31]. The creation of domain-specific editors is supported by language workbenches that provide frameworks to specify abstract syntax, parser, editor, and generators [9]. While MontiCore and MPS provide generators to create DSL editors that could be integrated into Eclipse or IntelliJ, Xtext and Spoofax focus on Eclipse-based extensions. In addition, Spoofax provides experimental support for an IntelliJ integration [13].

Likewise, Xtext has a strong focus on Eclipse-based extensions and lacks contributions for constantly supporting an IntelliJ integration [36].

In addition to approaches integrating language editors into classical IDEs, there is a variety of web-based language workbenches. These web-based language workbenches are divided into client-server and cloud-based Modeling as a Service (MaaS) approaches by Popoola et al. [25]. Modeling platforms from the latter category such as WebGME, CLOOCA, GenMyModel, MORSE, and MDEForge provide their services without requiring on-premise hardware. In contrast, client-server based approaches such as ModelBus, AToMPM, and DSLForge must be installed on a local server. In addition to relying on proprietary protocols [26], none of the language workbenches provides integration into editors or IDEs, such as Atom [2] or Eclipse, respectively.

While there exists a lot of research around language workbenches for textual external DSLs and web-based language workbenches, little attention has been paid to providing a multi-editor support for combining editor, IDE and browser-based integration. By introducing the LSP, Microsoft, RedHat, and Codenvy laid the foundation for providing language smarts in different development tools through a standardized protocol [22]. With Xtext being the first language workbench fully implementing the LSP [19] an additional milestone on the way towards multi-editor support was reached in 2017 [35]. While Rodriguez-Echeverria et al. [27] have already proposed an LSP infrastructure for graphical modeling, little attention has been paid to multi-editor integration of textual domain-specific languages.

7 Conclusions

We have reported the results of a case study conducted to quantify the effort required to integrate an Xtext DSL into different editors or IDEs utilizing the LSP. While the language server part can be generated completely based on an Xtext grammar, each development-tool extension has to be implemented specifically. However, the total integration costs are significantly below providing individual IDE-specific extensions. In addition, the SWOT analysis has revealed great potential of the LSP for standardizing and distributing language smarts for different editors and IDEs. However, exclusively targeting textual editors ignores alternative approaches such as projectional or graphical DSLs.

The LSP represents a major step towards multi-editor support for domain-specific languages. By separating language smarts and development-tool extensions by a standardized protocol, the integration costs for supporting multiple editors significantly decrease. However, the analysis of real world projects shows that there are only a few projects leveraging the Xtext-LSP integration and the ecosystem around it, e.g. marketplaces for development-tool extensions, to its full extent. Further research is required to analyze, whether these projects are early adopters or whether the Xtext-LSP integration is targeting only a niche market.

Large companies that build their tool-chains on top of the Eclipse ecosystem appreciate the seamless integration of different plugins that enables the efficient

creation of workbenches. The LSP is built on the assumption that every language server is independent and does not interact with other languages servers. While this approach increases the scalability, it also introduces disadvantages, since every language server has to re-implement certain functionality. The assumption that this separation of language servers and the consequent reimplementation of basic features hinders large companies to adopt the LSP, has to be substantiated by further research.

The contributions of this paper are the following. First, we have conducted a case study to quantify the efforts for implementing a DSL and integrating its editor into two different IDEs. Our case study has shown that this can be done with moderate effort. In addition, we have reported on the findings from a SWOT analysis performed to identify the potential impact of the LSP for textual domain-specific languages. Moreover, we have supplemented the findings of the SWOT analysis by investigating open-source github projects and the Eclipse TMF (Xtext) forum in order to examine the practical relevance of the LSP. The combination of the case study and the analysis of real world projects shows that the Xtext-LSP integration offers great potential that currently is not leveraged to its full extent.

Appendix: Syntax of Entity DSL in Xtext Format

```
Model:
   entities+=Entity*;
Entity:
   'entity' name=ID ('extends' superType=Entity)?
   '{'  features+=Feature* '}';
Feature:
   Property | Operation;
Property:
   name=ID ':' type=Type;
Operation:
   'op' name=ID '('(params+=parameter(','params+=parameter)*)?')
   '(':'returntype=Type)?;
Parameter:
   name=ID type=[Type];
Type :
   {Primitive} name=Primitive |  EntityReference;
EntityReference :
   {EntityReference} entityDefinition=[Entity];
Primitive :
   'number' | 'string' | 'boolean';
```

Listing 1.7. Grammar Rules for the Entity-DSL in Xtext format.

References

1. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. Addison Wesley Professional, Boston (2005)
2. Atom: A hackable text editor for the 21st century (2019). https://atom.io/
3. BSP: Build server protocol (2019). https://github.com/scalacenter/bsp

4. Bünder, H.: Decoupling language and editor – the impact of the language server protocol on textual domain-specific languages. In: Hammoudi, S., Ferreira, P.L., Selić, B. (eds.) Proceedings of the 7^{th} International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019), pp. 131–142. Prag (2019). https://doi.org/10.5220/0007556301310142, publication status: Published
5. Campagne, F.: The MPS Language Workbench: Meta Programming System, 3rd edn. Campagnelab, New York (2016). version 1.5.1 edn
6. Eclipse: Eclipse LSP4E (2018). https://projects.eclipse.org/projects/technology.lsp4e
7. Fette, I., Melnikov, A.: The websocket protocol. Technical report (2011)
8. Fowler, M.: Inversion of control containers and the dependency injection pattern (2004)
9. Fowler, M.: Language workbenches: The killer-app for domain specific languages (2005). https://www.martinfowler.com/articles/languageWorkbench.html
10. Gamma, E., Beck, K.: Contributing to Eclipse: Principles, Patterns, and Plug-ins. Addison-Wesley Professional, Boston (2004)
11. Helms, M.M., Nixon, J.: Exploring SWOT analysis - where are we now?: a review of academic research from the last decade. J. Strategy Manag. **3**(3), 215–251 (2010). https://doi.org/10.1108/17554251011064837
12. JetBrains: IntelliJ IDEA (2018). https://www.jetbrains.com/idea/
13. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and IDEs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2010, pp. 444–463. ACM, New York (2010). https://doi.org/10.1145/1869459.1869497
14. Kichwas Coders: Debugging protocol vs. language server protocol (2018). https://kichwacoders.com/2017/11/08/debug-protocol-vs-language-server-protocol/
15. Krahn, H., Rumpe, B., Völkel, S.: Efficient editor generation for compositional DSLs in eclipse. arXiv preprint arXiv:1409.6625 (2014)
16. Lamb, L., Robbins, A.: Learning the vi Editor, 6th edn. O'Reilly & Associates, Sebastopol (1998)
17. Maróti, M., et al.: Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure. In: MPM@ MoDELS, vol. 1237, pp. 41–60 (2014)
18. Microsoft: Code editing. Redefined (2018). https://code.visualstudio.com/
19. Microsoft: Implementations - language servers (2018). https://microsoft.github.io/language-server-protocol/implementors/servers/
20. Microsoft: Language server protocol specification - initialize (2018). https://microsoft.github.io/language-server-protocol/specification#initialize
21. Microsoft: Monaco editor - about (2018). https://microsoft.github.io/monaco-editor/
22. Microsoft: Overview - what is the language server protocol (2018). https://microsoft.github.io/language-server-protocol/overview
23. Mikkonen, T., Taivalsaari, A.: Using JavaScript as a real programming language (2007)
24. Spönemann, M.: The language server protocol in Java (2018). https://typefox.io/the-language-server-protocol-in-java
25. Popoola, S., Carver, J., Gray, J.: Modeling as a service: a survey of existing tools. In: MODELS (Satellite Events), pp. 360–367 (2017)

26. Rodriguez-Echeverria, R., Izquierdo, J.L.C., Wimmer, M., Cabot, J.: An LSP infrastructure to build EMF language servers for web-deployable model editors. In: Proceedings of the Second Workshop on Model-Driven Engineering Tools (MDETools 2018), pp. 1–10. CEUR (2018)
27. Rodriguez-Echeverria, R., Izquierdo, J.L.C., Wimmer, M., Cabot, J.: Towards a language server protocol infrastructure for graphical modeling. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 370–380. MODELS 2018. ACM, New York (2018). https://doi.org/10.1145/3239372.3239383
28. Sourcegraph: Languageserver.org (2018). https://langserver.org/
29. Stroustrup, B.: The C++ Programming Language. Pearson Education India, Bengaluru (2000)
30. Theia.org: Theia - cloud and desktop IDE (2018). https://www.theia-ide.org
31. Tomassetti, F.: The complete guide to (external) domain-specific languages (2017). https://tomassetti.me/domain-specific-languages
32. Vogel, L., Milinkovich, M.: Eclipse Rich Client Platform. Vogella series, Lars Vogel (2015). https://books.google.de/books?id=AC4_CQAAQBAJ
33. Vogel, L., Beaton, W.: Eclipse IDE: Java Programming, Debugging, Unit Testing, Task Management and Git Version Conrol with Eclipse, 3rd edn. Vogella Series, Vogella, Lexington (2013)
34. Völter, M.: DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. CreateSpace Independent Publishing Platform, Lexington (2013)
35. Xtext: Xtext 2.11.0 release notes (2017). https://www.eclipse.org/Xtext/releasenotes.html#/releasenotes/2017/02/01/version-2-11-0
36. Xtext: Idea support (2018). https://www.eclipse.org/Xtext/releasenotes.html
37. Yakindu: Yakindu solidity tools (2019). https://yakindu.github.io/solidity-ide/