Customizing Host IDE for Non-programming Users of Pure Embedded DSLs: A Case Study

Milan Nosáľ[a,*], Jaroslav Porubän[b], Matúš Sulír[b]

[a] *Independent / Svagant*
Mlynská 28, 040 01 Košice, Slovakia
[b] *Department of Computers and Informatics*
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia

Abstract

Pure embedding as an implementation strategy of domain-specific languages (DSLs) benefits from low implementation costs. On the other hand, it introduces undesired syntactic noise that impedes involvement of non-programming domain experts. Due to this, pure embedded DSLs are generally not intended for, nor used by, non-programmers. In this work, we try to challenge this state by experimenting with inexpensive customizations of the host IDE (Integrated Development Environment) to reduce the negative impact of syntactic noise. We present several techniques and recommendations based on standard IDE features (e.g., file templates, code folding, etc.) that aim to reduce syntactic noise and generally improve the user experience with pure embedded DSLs. The techniques are presented using a NetBeans IDE case study. The goal of the proposed techniques is to improve the user experience with pure embedded DSLs with a focus on the involvement of non-programming domain experts (or non-programmers in general). The proposed techniques were evaluated using a controlled experiment. The experiment compared a group using Ruby and non-modified RubyMine IDE versus a group using Java and NetBeans IDE customized to use the proposed techniques. Experiment results indicate that even

*Corresponding author
Email addresses: milan.nosal@gmail.com (Milan Nosáľ), jaroslav.poruban@tuke.sk (Jaroslav Porubän), matus.sulir@tuke.sk (Matúš Sulír)

 This is the accepted version of: M. Nosáľ, J. Porubän, M. Sulír. Customizing host IDE for non-programming users of pure embedded DSLs: A case study. Computer Languages, Systems and Structures (COMLAN), Vol. 49, 2017, pp. 101–118. http://doi.org/10.1016/j.cl.2017.04.003

inexpensive host IDE customizations can significantly alleviate issues caused by the syntactic noise: Java with its inflexible syntax performed better than Ruby with its concise syntax.

Keywords: domain-specific language, pure embedding, syntactic noise, controlled experiment

1. Introduction

Domain-specific languages (DSLs) are languages tailored for a specific, rather narrow domain [1, 2]. The main advantages of DSLs are high abstraction level, high expressivity due to usage of domain concepts in a language, and involvement of domain experts [2, 3, 4, 5]. As a result, applications of the DSL approach can be found in many domains, such as secure logging [6], questionnaires [7], database design [8], feature modeling [9], expert systems [10], etc. Disadvantages are mainly centered around language creation costs and insufficient tool support [2, 11].

From the practical point of view, there are two common possibilities of DSL implementation[1]: embedded DSLs (EDSLs) and stand-alone (external) DSLs.

A stand-alone DSL has its own parser and thus does not pose any restrictions on the DSL syntax. However, the creation cost of such a language can be high. Not only is it quite time-consuming, but it also poses high mental and knowledge requirements on the developer, like proficiency in context-free grammars and specialized tools (e.g., parser generators, language workbenches, etc.) [13, 14, 15].

From the implementation viewpoint, there exist heterogeneous and homogeneous embedded DSLs [12]. Heterogeneous embedding utilizes language composition on the level of tools – in heterogeneous embedding, the embedded language and the host language are not processed by the same compiler. An example of

[1] For a more thorough discussion of the categorization of DSL approaches and the terminology, we refer the reader to [12].

heterogeneous embedding are island grammars [16]. "Islands" are grammars embedded in the host language grammar, which is called "water". "Islands" are parsed using their own parser, the "water" is parsed by the host language parser.

Homogeneous embedding, on the other hand, reuses a host language compiler to reduce implementation costs [12]. As an example of such embedding, we can mention extensible compilers – an extended compiler parses both the embedded and host language. A special case of homogeneous embedding is pure embedding (first coined by Hudak [17]) when the host compiler is fully reused: the pure EDSL is a valid sentence in the host language, without a need of preprocessing or compiler extension. In the case of pure embedding, the DSL uses the host language's syntax to embed a DSL into a host GPL (general-purpose language). Basically, a pure EDSL is an API (application programming interface) designed with a strong focus on fluency, readability and the use of domain-specific terms. For its low creation costs, pure EDSLs are an obvious choice for many real-life applications and for DSL prototyping.

Faster creation of pure EDSLs comes at a price: they contain "syntactic noise" like brackets or import statements. Therefore, pure embedding poses an important question to the developer: What host language to choose? Some dynamically typed languages with very flexible syntax, e.g., Ruby, offer a low level of syntactic noise [18, 19] – there is only a small portion of syntax elements unrelated to the problem domain. This is the reason they are a common choice for the EDSL approach. However, large codebases are written in languages with rigid syntax like Java, which is also used to implement pure EDSLs [14, 20], although less commonly. Furthermore, there is a much larger community of programmers available (i.e., a larger workforce).

The main drawback of syntactic noise is that it impedes non-programming domain experts' involvement. For this reason, one could come to a conclusion that an EDSL is just a language with a higher level of abstraction meant only for programmers. One of the characteristics of DSL programs is that they can often be written by non-programming domain experts [5]. However, pure

EDSLs, or EDSLs in general, are often designed only for programmers. We try to challenge this state by using small, inexpensive IDE customizations to support EDSL usage by non-programmers. Our work is based on the idea that when we evaluate the design of DSLs, we should consider *not only their textual expressiveness but also IDE support*. Therefore, the goal of this paper is twofold.

First, *we will show how an existing GPL IDE can be used to facilitate usage of pure EDSLs by non-programmers*. Although in our work we focus on non-programmers, programmers can benefit from these techniques as well. We will present a set of techniques achieved by small and inexpensive IDE customizations, requiring almost no effort to implement.

Second, *we aim to show how proper IDE support for a syntactically rigid language like Java can compete with, or even outweigh the advantages of a flexible language like Ruby* (Ruby, along with Groovy or Clojure, is one of the most preferable languages for pure embedding [21]). In the DSL community there is a lack of evaluation research which was outlined also in other studies [22, 23]. Therefore, we performed a controlled experiment with non-programmers, comparing a Java group with small IDE customizations vs. a Ruby group with a standard IDE.

Contributions of this work can be stated as follows:

- a description of techniques aiming to improve the experience of non-programmers (and also programmers) with writing pure EDSL programs,
- implementation of a NetBeans plugin for folding and guarding surrounding noise for the purposes of a case study,
- minor implementation of origin tracking for domain error messages that piggybacks on NetBeans IDE, and
- experimental evaluation of the aforementioned techniques on non-programmers.

The case study presented in this paper is performed on a combination of the Java language and the NetBeans IDE. We used Java, since due to its rigid

syntax, its EDSLs usually end up with quite a lot of syntactic noise. Thus we were able to achieve higher contrast between the original EDSL and the EDSL with a customized IDE. We used the NetBeans IDE for its native support of guarded sections, which is applied in one of the presented techniques. Its two competitors, IntelliJ IDEA and Eclipse, do not support guarding code sections out of the box – by implementing this functionality ourselves, we would increase the costs of the IDE customizations.

Both to demonstrate the IDE customization and to perform a controlled experiment, we used our *Test-me!* language[2]. It is a DSL for defining tests for exams (inspired by the Moodle learning management system). In Listing 1, there is a Ruby version of a sample test. Listing 2 presents a Java version of the same test (as you can see, with significantly more syntactic noise – class definition, methods' definitions, parentheses).

2. IDE Reuse and Customization Techniques

In this section, we will briefly present several techniques which reuse existing IDE features and customize them to suit the EDSL needs. As case study objects, we will use the NetBeans IDE with the Java language. However, note that the approach is not limited to this combination. Other IDEs and languages have similar possibilities. Proposed techniques include the following:

1. A technique for *generating the boilerplate* EDSL sentence skeleton to substitute the need for a host language programmer presence during skeleton creation (section 2.1).

2. A technique for *hiding syntactic noise* that reduces the "amount" of syntactic noise presented to the EDSL user (section 2.2).

[2]The implementation of the language along with a usage example can be found in the following GitHub repositories: `http://github.com/MilanNosal/java-testDSL`, `https://github.com/MilanNosal/ruby-test-dsl`

Listing 1: An example of a biology test in *Test-me!* EDSL, Ruby version

```
require_relative "./TestDSL/testDSL"

create_test "Biology test", 20

multiple_choice_question "Which of the following are herbivores?", 10
incorrect_answer "Lion"
correct_answer "Sheep"
incorrect_answer "Bear"
correct_answer "Cow"

open_answer_question "What does a cat say?", 20, "Meow"

pairing_question "Combine males and females:", 10
pair "Lion", "Lioness"
pair "Bull", "Cow"
pair "Tiger", "Tigress"

run_test
```

Listing 2: An example of a biology test in *Test-me!* EDSL, Java version

```
package test;

import language.builder.ParsingException;
import language.builder.TestBuilder;

public class BiologyTest extends TestBuilder {

    @Override
    protected void define() {
        create_test("Biology test", 20);

        multiple_choice_question("Which of the following are herbivores?", 10);
        incorrect_answer("Lion");
        correct_answer("Sheep");
        incorrect_answer("Bear");
        correct_answer("Cow");

        open_answer_question("What does a cat say?", 20, "Meow");

        pairing_question("Combine males and females:", 10);
        pair("Lion", "Lioness");
        pair("Bull", "Cow");
        pair("Tiger", "Tigress");
    }

    public static void main(String[] args) throws ParsingException {
        new BiologyTest().compose();
    }
}
```

3. A technique *preventing inexpert editing* of the hidden syntactic noise (section 2.3).

4. A *code completion support* technique that should increase user experience writing sentences in EDSL (section 2.4).

5. Finally, an *error reporting* technique providing clickable navigational links to domain errors (section 2.5).

We would like to emphasize that we do not claim these techniques represent the full extent into which an IDE can be inexpensively customized for the use with a pure EDSL. This is a case study with one particular IDE – the NetBeans IDE – and the Java language. Throughout the following sections, we also discuss the applicability of the presented techniques in a broader scope. Furthermore, different combinations of IDEs and host languages might come with other features that can be used for pure EDSL support. The goal of this work is to assess the applicability of IDE customizations for EDSLs and motivate further research in this area.

2.1. Generating the Boilerplate

In Listing 2 with the test in Java, we see quite a large amount of boilerplate code: a package declaration, an import statement, a class declaration, and a method declaration along with an annotation. For a seasoned programmer, writing such code is tedious and time-consuming. For a non-programmer, such a task can be an impassable obstacle since it requires nontrivial knowledge of the implementation language.

The simplest solution to support boilerplate generation in IDEs is to use *file templates*. Every IDE has built-in file templates for supported programming languages. Consider a typical Java interface creation process: A wizard asks the user for a package and interface name. Then it instantiates a file template containing a Java interface skeleton, substituting specially marked placeholders in the template with the user-supplied values. The newly created file is then opened in the IDE, ready for use.

In the same vein, we recommend creating a file template for each pure EDSL. The main idea is that templates remove the need for host language programmer involvement in the sentence skeleton creation – the template can provide a skeleton in just a few mouse clicks. File templates are supported in all state-of-the-art IDEs, thus the technique can be used widely both in terms of IDEs (e.g., Microsoft Visual Studio and its Item Templates, file templates in JetBrains products), and host languages (.NET languages in Visual Studio, Python in PyCharm, Ruby in RubyMine, etc.).

For our case study, we have written a NetBeans IDE template for the *Test-me!* Java-based pure EDSL that creates the whole structure and an example of a test definition. The template is a part of the *EDSLAddon*[3] plugin for the NetBeans IDE. When the plugin is installed, a user can invoke a wizard to create a new *Test-me!* file. Figure 1 illustrates the *Test-me!* file template.

Of course, a file template is EDSL-specific, so a single template cannot be reused for more pure EDSLs. However, for a language author, writing a template for one pure EDSL is a simple task comparable to writing a single new pure EDSL sentence.

2.2. Hiding Syntactic Noise

Creating a ready-to-edit file by a wizard offers neat support for a non-programmer trying to use a pure EDSL. However, she might still be confused by unknown, domain-unrelated concepts present in the file – the "syntactic noise". Syntactic noise consists of syntactical elements of the GPL unrelated to the problem domain – e.g., braces or import statements. For the purpose of this work, we will distinguish two types of syntactic noise depending on the relative position of its language structures to the pure EDSL sentence:

- *Surrounding noise* – elements preceding or following the sentence of the pure EDSL language.

[3] `https://github.com/MilanNosal/edsl-addon`

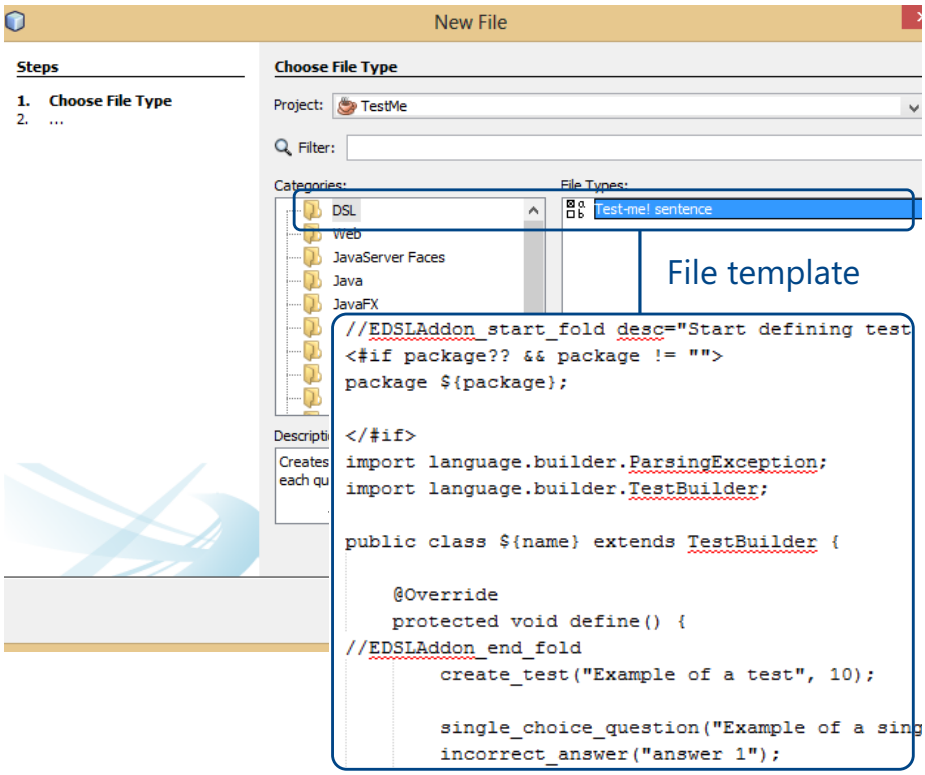

Figure 1: Java-based *Test-me!* EDSL sentence template

- *Interlacing noise* – elements that appear directly in the body of the pure EDSL sentence, but are redundant from the point of the DSL.

In the Java-based EDSL sentence introduced in Listing 2, the surrounding noise is a prefix made of the package declaration, imports, the `Biology` class declaration, its `define()` method declaration; and a postfix consisting of closing curly brackets for the class and method definitions, and the whole `main()` method. The `define()` method body is the actual pure EDSL sentence. In this case, the interlacing noise consists mainly of the parentheses and commas. Both syntactic noise types are necessary in the sentence since the code would become uncompilable by the standard Java compiler without them. Just to compare it to Ruby, in the Ruby-based EDSL sentence introduced in Listing 1 the surrounding noise consists of the single include statement (an equivalent of

the Java import section).

Our next step is to hide the syntactic noise from the user's view of the code. Ideally, we would strive to remove the syntactic noise completely. However, since we restricted ourselves to inexpensive IDE customizations, we consider removing surrounding noise good enough.

In IDEs, there is already a feature that allows hiding uninteresting code (uninteresting at a given time) – *code folding*. Folding some parts of the code to provide its projections is a feature found in probably all IDEs, from Xcode for Swift and Objective-C, through IntelliJ IDEA for Java, to Visual Studio for .NET languages and C++. Code folding allows collapsing (hiding) and expanding certain parts of the document via clickable icons appearing left to the source code (see Figure 2 showing folding in the NetBeans IDE). In some of the IDEs (e.g., NetBeans or IntelliJ IDEA), implementing a code fold can be as simple as adding special comments to the code. The template creator (i.e., the DSL author) can specify it using the snippet presented in Listing 3.

Listing 3: Example of custom fold definition

```
// <editor-fold desc="Start defining test here" defaultstate="collapsed">
package test;
...
// </editor-fold>
```

In the above fragment, we defined that the fold is collapsed by default, thus hiding the syntactic noise from the perspective of the DSL user unless she explicitly expands it. Furthermore, a textual description is displayed when the fold is collapsed, giving the user hints about DSL usage (see Figure 2).

In NetBeans IDE, specifically, we encountered a small implementation problem. We wanted to hide a method declaration, but not its body. This would cause overlapping with a built-in method body fold, which is not allowed. As a workaround, we implemented a simple NetBeans IDE plugin that re-registered the custom fold manager (a class taking care of folds) with higher priority to

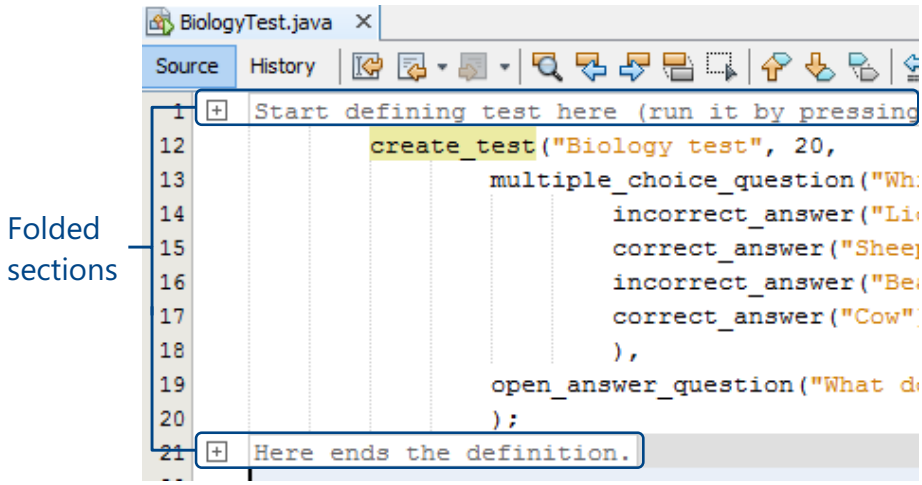

Figure 2: Code folding used to hide surrounding noise

override the standard Java fold manager. From the implementation viewpoint, this only required a simple configuration file.

In the *Test-me!* Java-based pure EDSL we were able to hide 10 significant lines of code (LOC) of the surrounding syntactic noise, and instead show 2 LOC that inform the user where the DSL sentence has to be written. If the plugin is installed, the custom tags can be used to hide practically any piece of code, for any pure EDSL (though folding of interlacing noise would be impractical and complicated due to the character of folding and the fact that interlacing noise is frequently edited).

2.3. Preventing Inexpert Editing

Although hidden, the folded parts of the code are still sensitive to inexpert manipulation: they still can be expanded and then edited. If the declarations will get damaged, domain expert will most likely fall short of expertise to fix them, and a Java programmer will have to get involved. The solution to this issue would be an IDE feature that is able to protect code snippets of interest against any inexpert manipulation.

The NetBeans IDE provides a feature called *guarded sections*[4]. A guarded section is a read-only section of a document that is protected by the editor. This way we can ensure that the hidden surrounding noise will be protected against inexpert manipulation. Guarded sections ensure the EDSL user will never invalidate sentence skeleton generated by the file template.

We added this guarding to our folds so that the section which is marked to be folded is also protected. To add guarding to folded parts, we had to implement a custom fold manager. Instead of the `<editor-fold>` XML-like tag, it uses a pair of `EDSLAddon_start_fold` and `EDSLAddon_end_fold` comments. The implementation has less than 200 LOC[5] and is reusable for any pure EDSL in Java without the need of any modification.

In Figure 3 there is a screenshot showing code guarding in action. The NetBeans IDE highlights guarded sections with the gray background, to indicate they are disabled for editing.

When using this approach, the template creator must design the template in a way that the guarded sections will never need to be manually modified (even in case of an error). For example, in our language, all necessary classes are already imported in the imports section.

2.4. Code Completion Support

So far, we were interested only in syntactic noise – the noise present in the textual representation of the DSL sentence itself. However, noise can be present also in additional IDE views (thus we will call it the *view noise*) presented on top of the source code. One such view is a code completion window. When activated, it shows the developer a list of language elements from which she can choose (see Figure 4). Often the number of suggested elements is overwhelming

[4]Although we claim that the techniques are not restricted to NetBeans with combination with Java, guarded sections are not featured by all the IDEs. E.g., up to the date of writing this article, two major competitors of NetBeans IDE – IntelliJ IDEA and Eclipse – do not provide this feature.

[5]See `https://github.com/MilanNosal/edsl-addon`

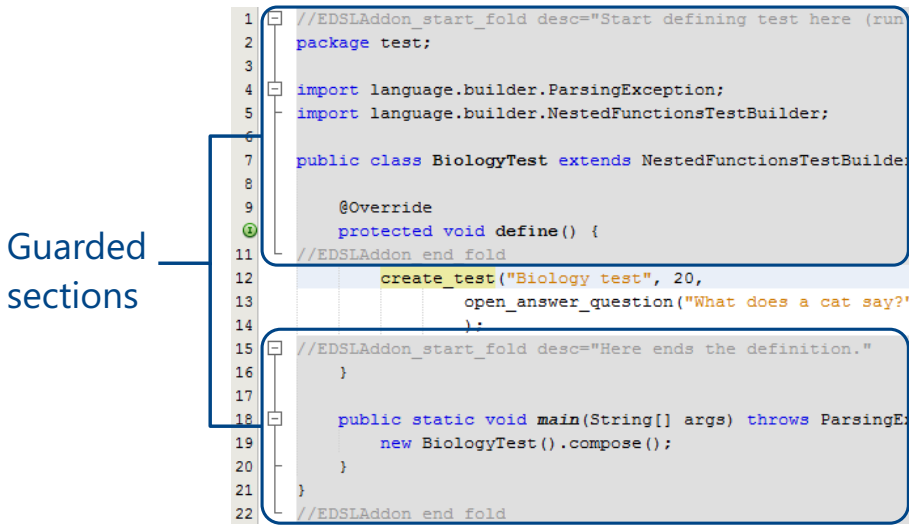

Figure 3: Guarding hidden sections

and the non-programming DSL user can get lost.

To cope with this problem, the NetBeans IDE[6] displays the most relevant suggestions above a thin gray line (see Figure 4). As a criterion, it uses type information. For example, if a method has a parameter of type `String`, the IDE displays available string variables above the line.

When designing a DSL, the author can use this behavior with advantage. If an appropriate pure EDSL design pattern is followed, the IDE displays just relevant domain-specific concepts above the line.

In the first implementation iteration of our EDSL, we used the Nested Functions pattern[14], which composes functions by nesting function calls as arguments to other function calls. Nesting of methods requires that the parameters of outer method calls type-match the return types of the inner calls. This way the IDE can infer the possible method calls that can be nested in a given context.

In Figure 4 there is an example of a completion box that offers the most relevant language concepts above the thin gray line, and because of the Nested

[6]Other IDEs provide advanced code completion engines as well.

Functions and Object Scoping patterns, those are the domain-specific methods. At that place where the box is activated, the call to the `create_test` method expects a question parameter. All proposed methods return objects of `Question` subclasses. In this process we did not have to modify the IDE, we only had to choose a suitable pure EDSL design pattern to leverage the properties of smart code completion. The Nested Functions and Object Scoping patterns can be applied for many EDSLs.

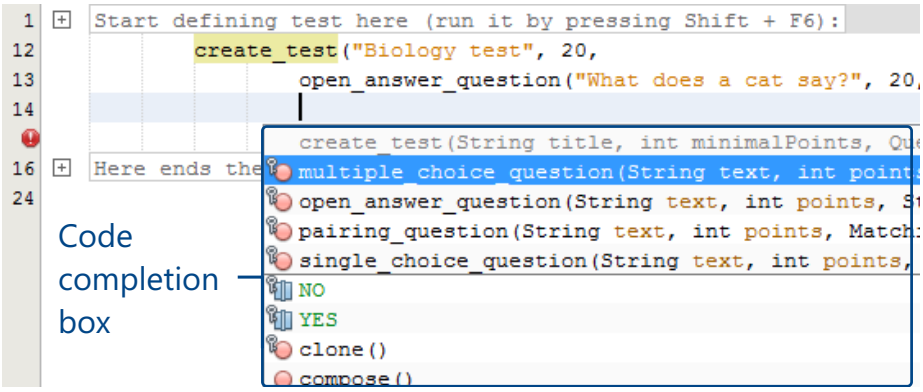

Figure 4: Smart code completion suggesting domain-specific concepts

There is one additional property of code completion: it can greatly alleviate the effects of interlacing syntactic noise. Code completion generates skeletons of method calls, including quotes, comma separators, semicolons, parentheses, and other Java lexical symbols. This is another argument why IDE support is important to consider when designing a DSL.

However, after pilot testing (see sections 3.2.5 and 3.5.2 for a detailed discussion) we found out that even with NetBeans' code completion feature the non-programming users had serious problems working with the language. We observed that they were overwhelmed by the necessity of proper parentheses nesting – a phenomenon that we too often see even with our first-year bachelors that are learning to program. For that reason, we decided to use the Function

Sequence pattern[7] that was used in the Ruby version too. Listing 2 presented in section 1 already shows the final form with the Function Sequence pattern.

2.5. Error Reporting

One of the challenges of DSLs is domain-specific error reporting. Suppose a user of the *Test-me!* DSL mistakenly inserts the same answer into one question two times (see the top part of Figure 5).

A typical approach to reporting such a mistake in EDSL would be printing a simple textual message, like "`The answer 'Sheep' is a duplicate`". An error reported this way can be very difficult to locate, especially if the DSL sentence is long, contains many occurrences of the word "Sheep", or if it is spread across files. Therefore it can be very useful to include also the location of the error - source file and line number. We implemented a simple utility class[8] that allows registering model objects (e.g., of type `Answer`) and then reporting errors with links to method calls that created these objects (e.g., a specific `incorrect_answer` call) for any pure EDSL. In our example, the error message is modified to look similar to the one in Listing 4.

Listing 4: Error message warning about a duplicate answer with a location report

```
The answer 'Sheep' is a duplicate,
in file BiologyTest.java, line 16.
```

While this certainly looks better and a programmer would find the location, manually opening and scrolling the file is cumbersome. Furthermore, a non-programmer could eventually get lost when searching for the correct file in IDE windows. Therefore, we recommend providing the user with *clickable naviga-*

[7] The Function Sequence pattern [14] combines function calls as a sequence of statements that builds the language model. Instead of nesting function calls, they have to be only called in a specific sequence (first `create_test()`, then `multiple_choice_question()`, etc.). The Ruby-based version shown in Listing 1 also uses the Function Sequence pattern.

[8] `https://git.io/voF9k`

tional links to the line in the source code that caused the domain error. To achieve this, we recommend using IDEs' output windows. IDEs usually present errors in the source code as clickable links to the error location. During printing a text to the output window, IDEs are able to recognize a pattern that defines a location in a source file, and they create a clickable link to provide easier navigation. This is common in all modern IDEs.

In Figure 5 there is an example of an error report produced by our utility class and then presented in the NetBeans Output window. Clicking the link to the error opens the appropriate source file in the editor and puts the cursor on the given line.

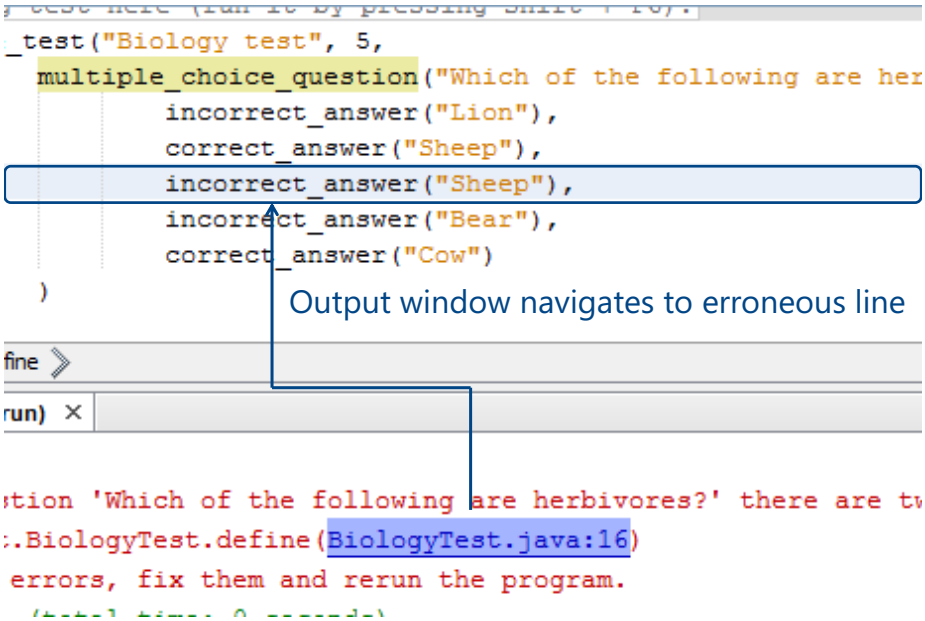

Figure 5: Output windows reports a duplicated answer in a multiple choice question and navigates to it

Using our utility class, any Java-based pure EDSL can report domain-specific errors with clickable links in the NetBeans IDE. Generally, the approach is usable also for other languages with proper IDE support (e.g., Java with IntelliJ IDEA, or Ruby with RubyMine).

Of course, this technique does not solve the problem of pure EDSL error reporting in general. If the EDSL sentence is not a valid host sentence, then syntactic errors are still reported in terms of the host language domain. Nevertheless, we expect that providing navigational links to errors' sources can help the users to work with pure EDSLs.

3. Experimental Evaluation

To evaluate the usefulness of the suggested IDE techniques, we conducted a controlled experiment. It consisted of a single task – the participants tried to write a DSL sentence in the given language/IDE. We investigated whether the IDE features used with a syntactically rigid and verbose host language can outweigh the syntactical flexibility of another language used with only a standard IDE. We compared a combination of Java-based pure EDSL and a customized NetBeans IDE (see sections 2.1–2.5) vs. a Ruby-based pure EDSL with a standard RubyMine IDE. The choice of the competitor was driven by the fact that Ruby is considered one of the best host languages for pure embedding [21]. As an IDE we used JetBrains RubyMine since it is one of the most used Ruby IDEs [24].

3.1. Hypotheses and Variables

We were interested in two indicators – effectiveness (correctness) and efficiency (correctness divided by time). Therefore, we formulate our null and alternative hypotheses as follows:

$H1_{null}$: The proportion of participants who correctly write the test DSL sentence using Java and customized NetBeans $\leq$ the proportion of participants correctly writing it using Ruby and RubyMine.

$H1_{alt}$: The proportion of participants who correctly write the test DSL sentence using Java and customized NetBeans $>$ the proportion of participants correctly writing it using Ruby and RubyMine.

$H2_{null}$: The efficiency of a test DSL sentence creation with Java/NetBeans $\leq$ the efficiency with Ruby/RubyMine.

$H2_{alt}$: The efficiency of a test DSL sentence creation with Java/NetBeans > the efficiency with Ruby/RubyMine.

We will statistically test the hypotheses with a 5-percent significance level ($\alpha = 5\%$)[9].

There is one *independent variable* – a combination of the used host language and IDE. There are two possible values: *Java* means the subjects used a Java-based pure EDSL and the NetBeans IDE customized with the presented techniques; *Ruby* means they used a Ruby-based pure EDSL with a common Ruby IDE (no customization).

There are two *dependent variables*. The first one is correctness of the task solution. The variable has two possible values: *yes* if the user was able to correctly solve the task in the given time span, or *no* if she was not.

The second dependent variable is efficiency, numerically expressed as correctness (0 or 1, since there is only one task) divided by time in minutes (with under 30 seconds precision). For a task which was not correctly completed in the allotted time frame, the efficiency was zero.

3.2. Method

Now we will describe the design and execution of the experiment.

3.2.1. Participants

Since our aim was to challenge quite a well-settled opinion that EDSLs are not for non-programmers, we decided to use non-programming participants. To conveniently get a larger sample of non-programming participants, we contacted a nearby high school[10] which specialized on French foreign language (under the assumption that their students would not be programmers). Moreover, since as students they were often tested, they were familiar with the *Test-me!* domain.

[9] This is an allowed probability of a Type I error, i.e., accepting H_{null} when it should have been rejected instead. According to Dybå et al. [25], 0.05 is a commonly accepted norm in software engineering research.

[10] Gymnázium Milana Rastislava Štefánika v Košiciach, `http://www.gmrske.sk/`

To test whether the potential participants fitted our non-programmer criteria, we asked each one of them to describe their previous experience with programming. Except for one[11] of 62 potential participants, they all claimed that they never programmed before. We queried also their Informatics teachers about the curriculum their students were going through, to ensure no threat to validity. In their curriculum, they focus on practical working with computers (using Photoshop, Microsoft Office tools, etc.), and the programming topic is presented only briefly and theoretically to provide the students with an idea what programming is. Thus, from the viewpoint of the school curriculum, they did not program.

In the end, using convenience sampling [26], we obtained 62 participants. Two of them were excluded: As we already noted, one had previous programming experience. The other one was excluded because of a technical problem during the experiment execution (the experimenter made an error and provided the student with the documentation for the other version of the language, not the one the subject was supposed to work with). This provided us with a total of 60 participants. The age of the subjects spanned from 14 to 18 years.

The experiment was done during substituted lessons in the Informatics laboratory of the high school. The participation was presented as a voluntary activity – students were given an option to reject participation without any sanction, as recommended by Carver et al. [27]. Although the participation was not awarded in any sense, the instructor tried to motivate students by presenting the materials in an interesting way and asking for their feedback during the training. At the same time, we did not fully reveal our hypotheses before the experiment [28]. By limiting the actual experiment time to 20 minutes, we kept students focused.

[11] In his free time, he was writing web pages in HTML and JavaScript.

3.2.2. Design

Each subject received only one treatment, i.e., the experiment design was unpaired. The *Java* group contained 32 participants and the *Ruby* group 28. The *Java* group consisted of 16 students in the age of around 18 and 16 students in the age of around 14. The *Ruby* group consisted of 15 students in the age of around 17 and 13 students in the age of around 15.

We assigned the participants into groups according to their attendance to the lessons which we could access. During two lessons, the students were assigned to the *Java* group, students attending the other two lessons to the *Ruby* group. Each lesson was attended by students of a different classroom.

3.2.3. Materials

Participants were given one of the two versions of the *Test-me!* language.

The *Java* group used a Java-based pure EDSL version[12] – an example of the language sentence is presented in Listing 2. They worked with the language using NetBeans IDE with the *EDSLAddon* plugin that provided an EDSL file template, folding and guarding for the *Test-me!* language.

The *Ruby* group used a Ruby-based version of the language[13] – an example of the language sentence is listed in Listing 1. They worked with the language using the RubyMine IDE by JetBrains without any modifications.

Both implementations contained customized documentation of the language, explaining how to create a sentence in the language, documentation to each domain-specific method call, and finally instructions how to run the test. The documentation also contained a short tutorial explaining the benefits of using the IDE (code completion, error reporting). The documentation is included in the GitHub repositories of the languages (folder "dokumentacia").

In both cases, the language implementation generates an HTML+JavaScript

[12] `https://github.com/MilanNosal/java-testDSL` – the repository includes its EDSL documentation (folder "dokumentacia") and experiment task specification (experimentTask.pdf)

[13] `https://github.com/MilanNosal/ruby-test-dsl` – the repository includes its EDSL documentation (folder "dokumentacia") and experiment task specification (experimentTask.pdf)

document that shows the test and after filling it, the JavaScript code evaluates it. Generated HTML pages look exactly the same, regardless of the used language implementation.

All materials (and also the DSL) were localized into participants' mother tongue – Slovak[14].

3.2.4. Task

The experiment consisted of a single task only. The participants had to implement a runnable examination test according to the specification they were given. The task specification included a screenshot of the desired output of the test definition (the generated HTML+JavaScript page for the correct solution) and a short natural language description of each question that was presented in the screenshot. The task specification is available online[15].

3.2.5. Pilot Testing

Before executing the experiment, we did pilot testing. Pilot testing was done with subjects from the same high school – but without any overlapping with the participants of the actual experiment, to prevent a serious threat to validity. After the pilot testing, we have adjusted two properties of the experiment design.

First, as we mentioned in section 2.4, we finally decided to use the Function Sequence version of the Java language that does not use the code completion technique mentioned in section 2.4. During pilot testing, the subjects used code completion very scarcely. However, we observed that most issues with task completion originated in difficulties with proper pairing of the parentheses in function calls. Since we used the Nested Functions pattern, proper nesting of parentheses was crucial. For this reason, we decided to remove our technique of code completion (in section 2.4) from the experiment. Instead, we used the Function Sequence pattern that is also used in the Ruby version of the language

[14]The most recent versions of the projects on GitHub are localized to English to make it accessible to the readers of the article.

[15]`https://github.com/MilanNosal/java-testDSL/blob/master/experimentTask.pdf`

(see listings 2 and 1). Since after this change both versions of the EDSL used the same implementation pattern, this change did not introduce a threat to validity.

Second, pilot testing determined the timing of the experiment. With each group, we had only a single 45-minute lesson which we could use for a single run of the experiment. We had to design the experiment to fit into this time. We needed around 15 minutes for introductory training and around 10 minutes for organizational tasks (preparing the environments, running the scripts to collect results, etc.). That left us with around 20 minutes for the experiment itself. Similarly, due to timing restrictions, we did not do any post-experiment satisfaction surveys with the participants.

3.2.6. Procedure

We executed the experiment during four substituted lessons. The first run was performed with 16 students using the Java-based language version, the second with 15 students using the Ruby-based version, third with 16 students using Java, and fourth with 13 students using Ruby again. For each experiment run, we had 45 minutes – one teaching lesson.

Before the participants were given the task, they were given a short training to get familiar with the DSL. During the training, they were shown how to create a new source file with the necessary skeleton for writing a DSL sentence, how to define a test and a question, how to use the documentation that was provided, and finally how to run the test. During the training they were only passive observers, therefore when they were solving the task, it was the first time they were creating a sentence in the given DSL. Special attention was given to show them the benefits of using the IDE features such as code completion, inline documentation (Ruby-doc and JavaDoc), and in the *Java* group also the file templates. The training took approximately 15 minutes.

We bounded the time for the task itself to 20 minutes. During the task session, the subjects could discuss task definition (if there was something unclear) with the experimenter, however, no additional discussion about the *Test-me!*

EDSL was allowed. In the case of questions about the EDSL, the experimenter just referred the subjects to use the language documentation.

After 20 minutes (with under 30 seconds tolerance) since the start of the task phase, the experiment was stopped. Using a script, we created snapshots of the current state of each participant's project, and videos capturing their desktops were saved.

The remaining 10 minutes of the teaching lesson was reserved for organizational tasks (preparing the environments, etc.).

3.2.7. Data Extraction

The decision whether a task was fulfilled was semi-automatically performed by researchers. Automation was used to check whether the participant's solution was compilable and runnable (i.e., it did not contain any syntactic or domain-specific errors). We manually examined the tests to decide whether they exactly corresponded to the specification, or were comparably complex to the task specification[16]. Finally, videos were skimmed to determine completion times and to observe subjects' habits during working with the DSL.

3.3. Results

The summary results for correctness are in the upper part of Table 1. We can clearly see that the *Java* group performed better: 75% of the subjects implemented the test correctly. In the *Ruby* group, this proportion was lower (54%).

To confirm whether the results are statistically significant, we used Barnard's CSM test with confidence interval computation by Berger et al. [29], which is suitable for analysis of 2x2 tables. The computed p-value is 0.0406, which is

[16]On their own request, we allowed several participants to create their own tests instead of the simple Biology test that was the goal of the task. However, to ensure unbiased results, we required them to use all the question types that were required by the task specification, to use the same number of questions with at least approximately the same number of answers. This requirement ensured they had no advantage over the rest of the participants.

	Group	
	Java	Ruby
Correct	**24** (**75%**)	**15** (**54%**)
Incorrect	**8** (**25%**)	**13** (**46%**)
Mean efficiency [tasks/min]	0.044	0.032

Table 1: The summary results. Statistically significant results are in bold.

less than α (5%). Therefore, we reject $H1_{null}$ and accept $H1_{alt}$. The result is statistically significant. We showed that the correctness was better for the *Java* group.

The mean efficiency was also better for the Java group: 0.044 vs. 0.032 successfully completed tasks/min (see the lower part of Table 1). However, the Mann–Whitney U test on the given data produces a p-value of 0.1000 – the result is not statistically significant.

In Table 2, we provide detailed experiment results for individual participants.

3.4. Threats to Validity

This section discusses threats to validity of the controlled experiment according to Neto and Conte [30] and Wohlin et al. [26].

3.4.1. Construct Validity

The chosen design of *Test-me!* might be a threat to validity. As we mentioned in section 3.2.3, after pilot testing we changed the design of the Java version of *Test-me!*, which lead to significantly better results with it. This indicates that a slight change in the EDSL language syntax can significantly improve its success rate. A different design of the Ruby version could possibly end in better results with Ruby. We used the design as illustrated in Listing 1, because it is very simple, and it resembles natural language. Now although there is no doubt a language implementation pattern can significantly affect its success rate, this aspect should not affect the experiment because in the experiment

Correct	Time [min]	Efficiency [task/min]
1	20.00	0.050
0	-	0.000
1	20.00	0.050
0	-	0.000
1	14.50	0.069
1	14.50	0.069
1	14.50	0.069
1	20.00	0.050
0	-	0.000
1	15.50	0.065
1	19.00	0.053
1	19.00	0.053
1	20.00	0.050
1	19.50	0.051
1	16.50	0.061
1	16.17	0.062
1	16.75	0.060
1	18.00	0.056
1	18.50	0.054
1	13.50	0.074
0	-	0.000
0	-	0.000
0	-	0.000
1	18.25	0.055
1	16.33	0.061
0	-	0.000
1	16.33	0.061
1	13.75	0.073
0	-	0.000
1	20.00	0.050
1	20.00	0.050
1	14.67	0.068

(a) The Java group

Correct	Time [min]	Efficiency [task/min]
0	-	0.000
0	-	0.000
1	17.27	0.058
0	-	0.000
1	20.00	0.050
0	-	0.000
0	-	0.000
1	13.00	0.077
1	19.67	0.051
1	19.25	0.052
0	-	0.000
1	15.00	0.067
0	-	0.000
1	17.00	0.059
1	12.50	0.080
0	-	0.000
0	-	0.000
0	-	0.000
0	-	0.000
0	-	0.000
1	18.50	0.054
1	13.50	0.074
1	17.50	0.057
1	18.58	0.054
0	-	0.000
1	18.25	0.055
1	20.00	0.050
1	15.50	0.065

(b) The Ruby group

Table 2: Experiment results for individual participants

both Java version and Ruby version used the same implementation pattern – the Function Sequence pattern [14]. Same implementation pattern ensures that both the control group with Java and the experimental group with Ruby had the same conditions from this aspect.

The domain selected for the experiment test should not pose a threat, since the particular domain (Biology in this case) affects only string literals of the *Test-me!* EDSL, which are handled in the same way in both RubyMine and NetBeans. A test domain could only affect the motivation of the participants to finish the task – a more interesting domain could have increased the success rate. However, since we used the same domain for both control and experimental groups, this could not affect the experiment's results.

Although the experiment consisted only of a single task to be performed by the participants, it was reasonably complex to exercise the majority of features provided by the *Test-me!* DSL.

The participants were required to create an EDSL sentence from the scratch. We used this design intentionally to test also file templates, which would otherwise be useless. However, the task designed this way limits generalization of our results to creating EDSL sentences from the scratch. To be able to generalize the conclusions for program comprehension, extension and modification, we would need to perform the experiment with a task including comprehending, extending, and modifying programs.

To measure correctness, we used just boolean values – either correct or incorrect. It would be difficult to assess the implementations in percentages.

While the assessment of correctness was subjective (by a researcher), disputes were resolved by mutual agreement of two researchers. Since we had source code snapshots and videos available, they were resolved without time stress.

3.4.2. Internal Validity

Assignment to experimental groups was performed by convenience, i.e., by a classroom a student attended. Therefore, it was a quasi-experiment [31].

Age	14	15	17	18
Group	Java	Ruby	Ruby	Java
Correct	11	7	8	13
Incorrect	5	6	7	3
Total	16	13	15	16

Table 3: The results for individual age groups

However, there were no obvious confounding factors common to each classroom except age. The mean age of participants was approximately equal for both treatment groups, so this should not be a problem either. Also, the success rate of the different age groups was distributed evenly (see Table 3), which indicates that the age did not play an important role.

Another validity threat was the quality of the training before the task. Although the same author led the training in all runs, he might have provided a worse explanation in the case of the *Ruby* group. To alleviate this threat, we alternated runs, so that experience he gained would evenly affect both groups.

To mitigate instrumentation threats, the documentation mentioned in section 3.2.3 was provided to ensure they all have the same amount of information. The two versions of the documentation differed only when it was necessary – when lexical symbols were different, and when we were talking about IDE related content. During the experiment, we observed that a common error in Ruby group was a result of copy/pasting the import section from the documentation. Higher quality of the documentation might have improved the success rate of the EDSL language. Considering that the Java version of the documentation was of equivalent quality, we can conclude that the file template customization can even alleviate shortcomings of poor documentation.

Since the experiment lasted only under 45 minutes without any prior treatment, maturation should not pose any threats to validity.

3.4.3. External Validity

Only students from one high school participated in our experiment, so they might not be representative of a whole population of non-programmers. However, they were familiar with the domain (from the end-user point of view) and they did not have programming experience, as it is described in section 3.2.1.

The representativeness of our EDSL, compared to real-world DSLs, is another validity threat. Although *Test-me!* is not a trivial language, it has rather straightforward and simple structure (see listings 2 and 1). An EDSL can be much more complex from the syntactic viewpoint – it can include variables, functions, structures, etc. We could expect that the user's benefit from our techniques (e.g., file templates) might be rendered insignificant by the complexity of the language syntax. Therefore we cannot generalize our results for all possible EDSLs. Replication of the experiment with a more complex EDSL is needed.

In this experiment, we compared just two specific combinations of a host language and IDE. The fact that we used two different IDEs designed and implemented by two different companies (RubyMine by JetBrains and NetBeans by Oracle) might be another threat to validity. Although we expect that the IDE of a dynamically typed language will be inferior to the IDE of a statically typed language, there might be other threats to validity that stem from the fact that the tools are designed by different companies. However, we need to mention that RubyMine is considered one of the best IDEs for Ruby [24], thus usage of a different IDE should not significantly affect the results.

Since our approach uses host IDEs, a learning curve of the IDE plays a role in the EDSL usage. In our experiment design, we gave the participants only a short training before the experiment. The training was without the hands-on experience. Thus the learning curve was a part of the experiment. We made this decision because the need of the whole IDE training on top of the EDSL training would be comparable to the training in the host language programming – if we required non-programmer users to become proficient IDE users to use our

EDSL, it would be the same as if we required them to learn the host language in order to use the EDSL. Therefore we have decided to include the IDE learning curve into the experiment as a part of the task.

We have to note that we did not want to teach them in depth how all the features of IDE work – that would be comparable to teach them how the host language works and would in effect result in clear advantage of external languages in the end. If the whole complex IDE has to be explained and understood, that puts us back at the beginning – an external language would be a clear choice over the EDSL. However, if the IDE can be used with positive effect although the user does not understand it besides several features closely related to the EDSL (in our case our customizations), then we can consider the EDSL + IDE customizations a potential challenge to external DSLs.

3.5. Conclusion

In this experiment, we tried to evaluate if proper IDE support (customized NetBeans) for a pure EDSL written in an inflexible, verbose language (Java) can outweigh a succinct language with a standard IDE (Ruby and RubyMine). The experiment was performed with non-programmers (high school students), thus demonstrating that IDE support is important when non-programmers try to use a pure EDSL. Regarding correctness, the Java group performed significantly better than the Ruby group, which indicates that the customizations can outweigh the syntactic shortcomings of the host language.

Although the experiment shows that IDE customizations can increase the non-programmers success rate with EDSLs, we cannot use its results alone to motivate the use of EDSLs over external languages in cases when non-programmers are involved. As the next step in this research, we need to compare IDE customizations with external language approaches (e.g., language workbenches), both in the aspect of user success rate and of implementation costs.

3.5.1. Observations

From the observation, we can conclude that the most significant reuse techniques are file templates, guarding, and folding. These techniques ensured that the non-programmer had to focus solely on the test definition. Several users of the Ruby-based EDSL failed to fulfill the task because they were not able to create an EDSL skeleton properly. In Ruby, the skeleton contained only an import statement (`require_relative` from Listing 1). However, they were copy/pasting the import from the documentation, copying also a dot that ended the sentence in the documentation, and they failed to find this error and fix it. The use of a file template would prevent this issue (since no copy-pasting nor understanding of skeleton code is needed) – thus we conclude that customization can even compensate for documentation of poor quality.

From the observation of the participants, we could conclude that the users were able to use code completion even without in-depth training of the IDE features. 10 of the total 24 successful Java users and 7 of the total 15 successful Ruby users used also code completion during solving the task. In several of those cases we noticed how they used code completion to deal with errors – once a participant made an error and was not able to solve it, she deleted the whole erroneous line and started from the beginning using code completion. Even though they were non-programmers, they could understand the advantages of using an IDE – providing evidence that an IDE can be really beneficial even to non-programming end users.

All the failing cases except one were due to host language syntactic errors – confirming that syntactic noise is really the main problem of EDSLs when used by non-programmers. Only a single participant failed due to a domain error – he used two correct answers in a single option question that did not support multiple correct answers, and he was not able to fix it in time.

We did not observe enough domain-specific errors to claim error reporting with navigational links useful. To confirm its usefulness, we could perform a new experiment focused on domain-specific errors.

Since we were capturing the screens while the participants worked, we were able to determine the approximate time of a successful solution (with under half-minute precision). In the Java group, the mean (average) time of a successful task solution was 17.3 minutes with a standard deviation of 2.3 minutes. In the Ruby group, the mean time was 17.0 minutes with a standard deviation of 2.6 minutes. This means that although the Java group performed significantly better in terms of success rate, the time was not improved when compared to the Ruby group. We believe this is a result of less interlacing syntactic noise in the case of Ruby (compare listing 1 with listing 2) – Ruby does not require the use of parentheses on method calls, and it also does not require a semicolon at the end of each statement. In the Java group, on the other hand, the customizations were able to help with a better success rate in solving the tasks. For example, in the Ruby group, the most often impassable problem was with the copy/pasted erroneous import statement with the dot at the ending. In the Java group, this situation could not happen – the surrounding noise was generated using file template and prevented from editing by guarded sections, so there was no way to introduce an error there.

3.5.2. Conclusions from Pilot Testing

Pilot testing showed that there are pure EDSL design patterns that are very inconvenient for non-programmers. The Nested Functions pattern provides a nice fluent interface that can benefit from NetBeans smart code completion (see section 2.4). However, in the case of non-programmers, the complexity of nested calls seems to by far outweigh any of their possible benefits. Observation of the pilot testing participants showed that all of them had troubles with proper parentheses nesting. Keeping the track of the open and closed parentheses is a challenging task even for our first-year students that learn to program – non-programmers became overwhelmed by the task. In the experiment, we used the Function Sequence implementation pattern that did not nest any function calls, thus each opened parenthesis was closed before another one was opened again.

After we changed the used design pattern for our Java EDSL to the Function

Sequence pattern (that was used in Ruby as well), the success rate became radically better. This indicates that the "amount" of syntactic noise is not so significant as the complexity of its structure (abstract syntax) – the number of brackets stayed the same, commas were substituted by the comparable number of semicolons. The problem was the higher complexity of their structure – brackets had to be properly nested. This observation renders the technique from section 2.4 unsuitable for non-programmers.

4. Discussion

Based on the experiment, we can see there are two different approaches that can make the IDE more useful by reducing or removing syntactic/view noise:

- *IDE customization* – the language author can modify IDE's standard behavior by means of a plugin (Sections 2.1, 2.2 and 2.3).
- *Language manipulation* – the language author can manipulate the pure EDSL implementation to reuse the host language IDE (Sections 2.4 and 2.5).

The language manipulation is non-invasive to the reused IDE infrastructure and therefore it is the less expensive alternative. The IDE modification is an invasive approach that requires more skill, but is also much more powerful. We can use the IDE API to move the language view closer to the domain while keeping its textual notation untouched.

As we could see, the user experience with pure embedding can be enhanced in multiple different aspects by IDE reuse and customization. All the techniques presented in our experiments can be easily applied to any Java-based pure EDSL without regard to its size. The language manipulation techniques required slight modifications to the language implementation, e.g., registering created objects to the utility class and then using the class to report errors. The IDE customization techniques required some effort to prepare the plugin, but the plugin itself can be reused by multiple pure EDSLs. Furthermore, the language implementation itself does not have to change. It is up to the language

author to consider the options of the given IDE and to choose only techniques which are relatively cheap to use for the given pure EDSL.

We can also see there are still many issues that are inherent to pure embedding and these simple techniques cannot overcome them – such as syntactic error reporting in DSL terms. We can also conclude from the experiment that some techniques might help programmers but are not viable for non-programming domain experts – namely smart code completion utilization presented in section 2.4.

5. Related Work

While there are many works contributing to the DSL field (e.g., optimization of their parsing [32]), we focus specifically on the editing environments for DSLs. The need for proper IDE support is manifested by language workbenches that are able to provide some tool support in addition to parsers for external DSLs (for example JetBrains MPS, GME, MontiCore [33], Neverlang [34], Rosette [35]). There are several good overviews on language workbenches that can be used to learn more about current trends [36, 37, 38, 39]. Adam et al. [40] recognized spreadsheet-based DSLs – DSLs consisting of data written in spreadsheets. They recognized the need of tools for these languages as well. IDE and tools support for embedding is not a novel need either, take for example the Varis tool [41] aiming to provide proper IDE support for embedded client code in PHP web applications. In our work, we focus on embedded DSLs as well; however, our primary aim is not to provide coding support for programmers using the pure EDSL, but for non-programming domain experts (while keeping additional implementation effort to a minimum).

As we have already mentioned, our work is based on the idea that when we evaluate the design of DSLs, we should consider not only their textual expressiveness but also IDE support. In the GPL context, this idea is not new. Chiba et al. [42] use IDE to introduce language syntax extension – e.g., they implemented the *Kide* editor that creates virtual source files by showing together

several related pieces of code from multiple source files. Virtual source files can be used to put together code implementing crosscutting concerns, but instead of doing so on the static textual level (AspectJ), *Kide* does it on the presentation level in the IDE. We used the same idea in [43] to aid program comprehension.

Renggli et al. [44] had a similar goal as our work – they wanted to reuse the tools of the host language during the embedding. However, they do not focus on homogeneous embedded languages only, but their tool *Helvetia* supports also heterogeneous embedding. Moreover, their approach requires the language author to be seasoned in context-free grammars and transformations, as *Helvetia* uses an EBNF-like DSL to extend the host grammar and a DSL for transformations to modify the semantics of the homogeneous embedded DSL where needed.

Dinkelaker et al. [45] use island grammars to specify concrete syntax of an embedded DSL. They use annotations on program elements to define concrete syntax of a previously homogeneous embedded DSL and, in turn, they use the IDE plugin to generate a corresponding preprocessor and language support such as code completion. This changes the language to heterogeneous with the implementation costs lower than in other approaches (like for example Metaborg [46]). Kurš et al. [16] use island grammars for creating composable and reusable islands called *bounded seas* that compute scope for water parsing. We aim for the same result as these approaches, but we keep the language homogeneously embedded.

Scherr et al. [47] present a prototype for staging shallow embedded DSLs to get benefits of deep embedding. Both shallow and deep embedding concerns homogeneous embedded DSLs. In our work we used shallow embedding – the calls were directly interpreted and the HTML test was generated. Deep embedding creates an intermediate representation (IR) of the language sentence that allows further processing – optimization, etc. Staging is the process of creating an IR for shallow embedded DSLs. While the prototype created by Scherr et al. [47] is based on Java annotations, they recommend solidifying embedding as a first-class feature of a host GPL (e.g., by using keywords for staging instead

of their custom annotations). Native support for embedding in a GPL could positively affect IDE reusability for embedded DSLs. They do not further address the problem of IDE reuse for domain experts. Quite a similar approach is proposed by Erdweg et al. [48], who basically focus on native support for embedding directly at the IDE level. They propose to organize editor features into editor libraries that should provide a simple way to extend the IDE to support the embedded language.

Considering that our approach projects the pure embedded DSL sentence into a different presentation syntax (it hides surrounding noise), we could relate it to projectional editors [49]. Usually, the goal of projectional editing is to provide a more flexible environment for language composition and extension. We used a similar idea to provide the non-programming domain expert better syntax for working with the language.

6. Conclusions

In this paper, we experimented with the host IDE reuse and customization to involve non-programmers in using a pure EDSL. We presented several techniques and recommendations that aim to support non-programmers' involvement. File templates can be used to generate the code skeleton (surrounding noise), so that the user does not have to write it herself (which might be an impassable obstacle for a non-programmer). Folding and guarding aimed to reduce syntactic noise by hiding and protecting the surrounding noise from the language user. Additionally, we proposed to use the Nested Functions pattern to utilize smart code completion so that it would prefer domain-specific method calls instead of other host language constructs. Finally, we recommended using error messages with navigational links to increase efficiency in working with pure EDSLs. While the techniques were presented using a case study based on the NetBeans IDE, most features reused by the proposed techniques exist in other IDEs as well, thus we assume wider applicability.

We conducted a controlled experiment to evaluate the proposed techniques.

The experiment compared non-programmers' work with Ruby and non-modified RubyMine IDE versus non-programmers' work with Java and NetBeans IDE customized to use the proposed techniques. The results confirmed that proposed techniques can even outweigh the inherent issues with inflexible syntax of the host language. On the other hand, the experiment indicated that the smart code completion utilization as proposed in section 2.4 brings more problems than benefits for the non-programmers. Of course, drawing conclusions from one experiment is not as certain as from a family of experiments [50], so replications by other researchers are welcome.

To sum up the article, when we evaluate the design of DSLs, we should consider not only their textual expressiveness but also IDE support, since it can improve the user experience of even non-programming domain experts while keeping creation costs relatively low (the main advantage of the EDSL implementation approach).

There are multiple directions for possible future work. The following two options are our priority. First, we need to analyze the applicability of our recommendations and techniques in different combinations of a host language and IDE (e.g., Java with IntelliJ IDEA, Ruby with RubyMine, etc.), since our experiment involved only Java with NetBeans IDE. Next, we would like to compare proposed techniques with stand-alone (external) DSLs. If a pure EDSL could compete with a stand-alone DSL just on the basis of IDE reuse, it could contribute to solving the open problem of selecting the viable implementation strategy for DSLs [51].

One of the EDSL implementation strategies are source code annotations. Annotations along with their processor form a formal language, and they provide a framework for language composition [52]. Source code annotations are one of the most commonly used formats for configuration languages [53], with many different usage domains (for example, we used annotations to annotate the source code with its intent [54]). In the future work, we would like to explore the opportunities to support annotation-based EDSLs using the host IDE.

Acknowledgment

We would like to give our thanks to the director and the teachers of Gymnázium Milana Rastislava Štefánika v Košiciach (`http://www.gmrske.sk/`) for giving us the opportunity to perform the experiment at their school. The thanks also belong to all the students that voluntarily participated in the experiment.

This work was supported by project KEGA No. 019TUKE-4/2014 Integration of the Basic Theories of Software Engineering into Courses for Informatics Master Study Programmes at Technical Universities – Proposal and Implementation.

References

References

[1] T. Kosar, P. E. Martínez López, P. A. Barrientos, M. Mernik, A preliminary study on various implementation approaches of domain-specific language, Information and Software Technology 50 (5) (2008) 390–405. `doi:10.1016/j.infsof.2007.04.002`.

[2] M. Mernik, J. Heering, A. M. Sloane, When and How to Develop Domain-specific Languages, ACM Computing Surveys 37 (4) (2005) 316–344. `doi:10.1145/1118890.1118892`.

[3] A. van Deursen, P. Klint, J. Visser, Domain-specific languages: an annotated bibliography, SIGPLAN Notices 35 (6) (2000) 26–36. `doi:10.1145/352029.352035`.

[4] D. Spinellis, Notable design patterns for domain-specific languages, Journal of Systems and Software 56 (1) (2001) 91–99. `doi:10.1016/S0164-1212(00)00089-3`.

[5] P. Hudak, Domain-specific languages, Handbook of Programming Languages 3 (1997) 39–60.

[6] S. Zawoad, M. Mernik, R. Hasan, Towards Building a Forensics Aware Language for Secure Logging, Computer Science and Information Systems 11 (4) (2014) 1291–1314. doi:10.2298/CSIS131201051Z.

[7] M. Gouseti, C. Peters, T. van der Storm, Extensible language implementation with object algebras (short paper), ACM SIGPLAN Notices 50 (3) (2015) 25–28. doi:10.1145/2775053.2658765.

[8] S. Ristić, S. Aleksić, M. Čeliković, I. Luković, Generic and Standard Database Constraint Meta-Models, Computer Science and Information Systems 11 (2) (2014) 679–696. doi:10.2298/CSIS140216037R.

[9] T. Frťala, V. Vranić, Animating Organizational Patterns, in: Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE '15, IEEE Press, 2015, pp. 8–14. doi:10.1109/CHASE.2015.8.

[10] M. Woźniak, D. Połap, C. Napoli, E. Tramontana, Graphic object feature extraction system based on Cuckoo search algorithm, Expert Systems with Applications 66 (C) (2016) 20–31. doi:10.1016/j.eswa.2016.08.068.

[11] S. Chodarev, J. Kollár, Extensible host language for domain-specific languages, Computing & Informatics 35 (1) (2016) 84–110.

[12] M. Mernik, An object-oriented approach to language compositions for software language engineering, Journal of Systems and Software 86 (9) (2013) 2451–2464. doi:10.1016/j.jss.2013.04.087.

[13] InfoQ, Developing a Complex External DSL, https://www.infoq.com/articles/External-DSL-Vaughn-Vernon, accessed: 2016-11-20.

[14] M. Fowler, Domain-Specific Languages, 1st Edition, Addison-Wesley Professional, 2010.

[15] W. R. Cook, Erik Meijer started a discussion on Domain Specific Languages, http://lambda-the-ultimate.org/node/4560, accessed: 2016-11-20.

[16] J. Kurš, M. Lungu, R. Iyadurai, O. Nierstrasz, Bounded seas, Computer Languages, Systems & Structures 44, Part A (2015) 114–140. doi:10.1016/j.cl.2015.08.002.

[17] P. Hudak, Building domain-specific embedded languages, ACM Computing Surveys 28 (4es) (1996) 1–6. doi:10.1145/242224.242477.

[18] H. C. Cunningham, A little language for surveys: constructing an internal DSL in Ruby, in: Proceedings of the 46th ACM Southeast Conference, ACM-SE 46, ACM, New York, NY, USA, 2008, pp. 282–287. doi:10.1145/1593105.1593181.

[19] T. Grust, M. Mayr, A Deep Embedding of Queries into Ruby, in: IEEE 28th International Conference on Data Engineering (ICDE), 2012, pp. 1257–1260. doi:10.1109/ICDE.2012.121.

[20] S. Freeman, N. Pryce, Evolving an embedded domain-specific language in Java, in: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, OOPSLA '06, ACM, New York, NY, USA, 2006, pp. 855–865. doi:10.1145/1176617.1176735.

[21] D. Ghosh, DSL for the uninitiated, Communications of the ACM 54 (7) (2011) 44–50. doi:10.1145/1965724.1965740.

[22] L. M. do Nascimento, D. L. Viana, P. A. S. Neto, D. A. Martins, V. C. Garcia, S. R. Meira, A Systematic Mapping Study on Domain-specific Languages, in: Proceedings of the 7th International Conference on Software Engineering Advances, ICSEA'12, 2012, pp. 179–187.

[23] T. Kosar, S. Bohra, M. Mernik, Domain-Specific Languages: A Systematic Mapping Study, Information and Software Technology 71 (C) (2016) 77–91. doi:10.1016/j.infsof.2015.11.001.

[24] StackOverflow, What Ruby IDE do you prefer?, http://stackoverflow.com/questions/16991/what-ruby-ide-do-you-prefer, accessed: 2016-06-30.

[25] T. Dybå, V. B. Kampenes, D. I. Sjøberg, A systematic review of statistical power in software engineering experiments, Information and Software Technology 48 (8) (2006) 745 – 755. doi:10.1016/j.infsof.2005.08.009.

[26] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Publishing Company, Incorporated, 2012.

[27] J. Carver, L. Jaccheri, S. Morasca, F. Shull, Issues in using students in empirical studies in software engineering education, in: Proceedings of the 9th International Symposium on Software Metrics, METRICS '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 239–249. doi:10.1109/METRIC.2003.1232471.

[28] J. C. Carver, L. Jaccheri, S. Morasca, F. Shull, A checklist for integrating student empirical studies with research and teaching goals, Empirical Software Engineering 15 (1) (2010) 35–59. doi:10.1007/s10664-009-9109-9.

[29] R. L. Berger, D. D. Boos, P Values Maximized Over a Confidence Set for the Nuisance Parameter, Journal of the American Statistical Association 89 (427) (1994) 1012–1016. doi:10.1080/01621459.1994.10476836.

[30] A. A. Neto, T. Conte, Threats to validity and their control actions – results of a systematic literature review, Technical Report TR-USES-2014-0002, Universidade Federal do Amazonas (Mar. 2014).

[31] J. Siegmund, J. Schumann, Confounding parameters on program comprehension: a literature survey, Empirical Software Engineering 20 (4) (2015) 1159–1192. doi:10.1007/s10664-014-9318-8.

[32] R. Polách, J. Trávníček, J. Janoušek, B. Melichar, Efficient determinization of visibly and height-deterministic pushdown automata, Computer Lan-

guages, Systems & Structures 46 (2016) 91–105. doi:10.1016/j.cl.2016.07.005.

[33] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel, MontiCore: a framework for the development of textual domain specific languages, in: Companion of the 30th international conference on Software engineering, ICSE Companion '08, ACM, New York, NY, USA, 2008, pp. 925–926. doi:10.1145/1370175.1370190.

[34] E. Vacchi, W. Cazzola, Neverlang: A framework for feature-oriented language development, Computer Languages, Systems & Structures 43 (2015) 1–40. doi:10.1016/j.cl.2015.02.001.

[35] E. Torlak, R. Bodik, Growing Solver-aided Languages with Rosette, in: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013, ACM, New York, NY, USA, 2013, pp. 135–152. doi:10.1145/2509578.2509586.

[36] M. Pfeiffer, J. Pichler, A Comparison of Tool Support for Textual Domain-Specific Languages, in: Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling, 2008, pp. 1–7.

[37] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, J. van der Woning, The State of the Art in Language Workbenches, in: Software Language Engineering, Vol. 8225 of Lecture Notes in Computer Science, Springer International Publishing, 2013, pp. 197–217. doi:10.1007/978-3-319-02654-1_11.

[38] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina,

M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, Evaluating and comparing language workbenches: Existing results and benchmarks for the future, Computer Languages, Systems & Structures 44, Part A (2015) 24–47. doi:10.1016/j.cl.2015.08.007.

[39] D. Méndez-Acuna, J. A. Galindo, T. Degueule, B. Combemale, B. Baudry, Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review, Computer Languages, Systems & Structures 46 (2016) 206–235. doi:10.1016/j.cl.2016.09.004.

[40] S. Adam, U. P. Schultz, Towards Tool Support for Spreadsheet-based Domain-specific Languages, SIGPLAN Notices 51 (3) (2015) 95–98. doi:10.1145/2936314.2814215.

[41] H. V. Nguyen, C. Kästner, T. N. Nguyen, Varis: IDE Support for Embedded Client Code in PHP Web Applications, in: Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 693–696. doi:10.1109/ICSE.2015.225.

[42] S. Chiba, M. Horie, K. Kanazawa, F. Takeyama, Y. Teramoto, Do We Really Need to Extend Syntax for Advanced Modularity?, in: Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development, AOSD '12, ACM, New York, NY, USA, 2012, pp. 95–106. doi:10.1145/2162049.2162061.

[43] J. Porubän, M. Nosáľ, Leveraging Program Comprehension with Concern-oriented Source Code Projections, in: 3rd Symposium on Languages, Applications and Technologies, Vol. 38 of OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2014, pp. 35–50. doi:10.4230/OASIcs.SLATE.2014.35.

[44] L. Renggli, T. Gîrba, O. Nierstrasz, Embedding languages without breaking tools, in: Proceedings of the 24th European conference on Object-oriented

programming, ECOOP'10, Springer, 2010, pp. 380–404. doi:10.1007/978-3-642-14107-2_19.

[45] T. Dinkelaker, M. Eichberg, M. Mezini, Incremental concrete syntax for embedded languages with support for separate compilation, Science of Computer Programming 78 (6) (2013) 615–632. doi:10.1016/j.scico.2012.12.002.

[46] M. Bravenboer, R. de Groot, E. Visser, MetaBorg in action: examples of domain-specific language embedding and assimilation using Stratego/XT, in: Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering, GTTSE'05, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 297–311. doi:10.1007/11877028_10.

[47] M. Scherr, S. Chiba, Almost First-class Language Embedding: Taming Staged Embedded DSLs, SIGPLAN Notices 51 (3) (2015) 21–30. doi:10.1145/2936314.2814217.

[48] S. Erdweg, L. C. Kats, T. Rendel, C. Kästner, K. Ostermann, E. Visser, Growing a Language Environment with Editor Libraries, SIGPLAN Notices 47 (3) (2011) 167–176. doi:10.1145/2189751.2047891.

[49] M. Voelter, J. Siegmund, T. Berger, B. Kolb, Towards user-friendly projectional editors, in: 7th International Conference on Software Language Engineering, SLE 2014, Springer, 2014, pp. 41–61. doi:10.1007/978-3-319-11245-9_3.

[50] T. Kosar, M. Mernik, J. C. Carver, Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments, Empirical Software Engineering 17 (3) (2012) 276–304. doi:10.1007/s10664-011-9172-x.

[51] A. H. Bagge, V. Zaytsev, Open and Original Problems in Software Lan-

guage Engineering 2015 Workshop Report, SIGSOFT Software Engineering Notes 40 (3) (2015) 32–37. doi:10.1145/2757308.2757313.

[52] M. Nosáľ, M. Sulír, J. Juhár, Language Composition Using Source Code Annotations, Computer Science and Information Systems 13 (2016) 707–729. doi:10.2298/CSIS160114024N.

[53] M. Nosáľ, J. Porubän, XML to Annotations Mapping Definition with Patterns, Computer Science and Information Systems 11 (2014) 1455–1477. doi:10.2298/CSIS130920049N.

[54] M. Sulír, M. Nosáľ, J. Porubän, Recording Concerns in Source Code Using Annotations, Computer Languages, Systems & Structures 46 (C) (2016) 44–65. doi:10.1016/j.cl.2016.07.003.

Vitae

Milan Nosáľ is currently working as a full-time iOS developer at Svagant, Košice. He received his PhD. in Computer Science in 2015 for his work "Leveraging Program Comprehension with Concern-oriented Projections" from Technical university of Košice, Slovakia. In his free time, he works on research in the field of attribute-oriented programming (source code annotations), program comprehension, projectional editors, and domain-specific languages.

Jaroslav Porubän is an Associate professor and the Head of Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in Computer Science in 2000 and his PhD. in Computer Science in 2004. Since 2003 he is a member of the Department of Computers and Informatics at Technical University of Košice. Currently, the main subject of his research is the computer language engineering concentrating on design and implementation of domain specific languages and computer language composition and evolution.

Matúš Sulír is a PhD student at the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University

of Košice. He graduated with a master's degree in Computer Science in 2014. His current research is focused on program comprehension, source code annotations, and empirical methods in software engineering.