Empir Software Eng (2012) 17:276–304
DOI 10.1007/s10664-011-9172-x

Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments

Tomaž Kosar · Marjan Mernik · Jeffrey C. Carver

Published online: 4 August 2011
© Springer Science+Business Media, LLC 2011
Editor: Giulio Antoniol

Abstract Domain-specific languages (DSLs) are often argued to have a simpler notation than general-purpose languages (GPLs), since the notation is adapted to the specific problem domain. Consequently, the impact of domain relevance on the creation of the problem representation is believed to improve programmers' efficiency and accuracy when using DSLs compared with using similar solutions like application libraries in GPLs. Most of the common beliefs have been based upon qualitative conclusions drawn by developers. Rather than implementing the same problem in a DSL and in a GPL and comparing the efficiency and accuracy of each approach, developers often compare the implementation of a new program in a DSL to their previous experiences implementing similar programs in GPLs. Such a conclusion may or may not be valid. This paper takes a more skeptical approach to acceptance of those beliefs. By reporting on a family of three empirical studies comparing DSLs and GPLs in different domains. The results of the studies showed that when using a DSL, developers are more accurate and more efficient in program comprehension than when using a GPL. These results validate some of the long-

This work was partially sponsored by the bilateral project "Program Comprehension for Domain-Specific Languages" (code BI-PT/08-09-008) between Slovenia and Portugal.

T. Kosar (✉) · M. Mernik
Faculty of Electrical Engineering and Computer Science, University of Maribor,
Smetanova ulica 17, 2000 Maribor, Slovenia
e-mail: tomaz.kosar@uni-mb.si

M. Mernik
e-mail: marjan.mernik@uni-mb.si

J. C. Carver
Department of Computer Science, University of Alabama,
Tuscaloosa, AL, USA
e-mail: carver@cs.ua.edu

held beliefs of the DSL community that until now were only supported by anecdotal evidence.

Keywords Domain-specific languages · General-purpose languages · Program understanding · Program comprehension · Controlled experiments · Language evaluations

1 Introduction

Software complexity has rapidly increased in recent years. The need to solve compound problems in varied domains is making emerging technologies and general-purpose languages (GPLs) increasingly complex. As a result, programmers need techniques to deal with the difficulties in a domain or set of related domains. Usually, domain functions are implemented in GPLs as an application library accessible through the application programming interface (API). Development of a domain-specific language (DSL) is another way to address domain complexity (Elliott 1999; Thibault et al. 1999; Wile 2001; Mauw et al. 2004). A DSL is specialized for a particular problem domain, provides a notation close to the application domain, and is based only on the concepts and features of that domain (Mernik et al. 2005; Sprinkle et al. 2009). End-users who do not have previous knowledge of general programming typically can work with DSLs (Peyton Jones et al. 2003). However, DSLs are not only for end-users. Many DSLs bring feasible alternatives to application libraries, such as integrated languages (e.g., LINQ; Meijer et al. 2006) and markup languages (e.g., XAML; MacVittie 2006). Although these *little languages* (Bentley 1986) seem to be easier to use than similar solutions, like application libraries in GPLs, there are few empirical studies that validate this belief. One example is a study by Kieburtz et al. (1996), which is discussed in Section 2.

Program comprehension is the process of understanding a program's meaning and behavior. It is an important software development activity (Storey 2005; Varanda Pereira et al. 2008). Although program comprehension is not an independent phase, it is a component of nine different software development phases (Hevner et al. 2005; Webb Collins et al. 2008). This point highlights the importance of program comprehension research.

Modern Integrated Development Environments (IDEs) contain program comprehension tools that facilitate the programmer's work. However, the programmer must still spend considerable time manually performing program comprehension (Hevner et al. 2005). Manual program comprehension involves construction of a mental model of program behavior. While building this model, in addition to domain-specific concepts in the problem space, programmers also have to process and understand their implementation using programming language concepts (the solution space). A DSL helps to reduce the semantic gap between the problem space and the solution space. Therefore, the goal of this paper is to test whether this reduction contributes to better program comprehension when using DSLs vs. using GPLs.

To test the effectiveness of DSLs, we conducted a family of three experiments to compare program comprehension between DSLs and GPLs. Each participant performed two tasks on a DSL and two on a GPL. We used these tasks to evaluate the effects of the DSL on program comprehension with respect to learning, understanding and evolution. Each experiment had at least 34 participants. Because the experiments had students as the participants, we followed Carver et al.'s guidelines for conducting experiments in a classroom environment (Carver et al. 2003, 2010). Using the guidelines, we carefully selected participants based on their knowledge and experience so that the results of the studies would be valid.

The remainder of this paper is organized as follows. Section 2 describes related work on DSLs, program comprehension and experiments. Section 3 highlights the definition and the precise focus of the family of controlled experiments. Section 4 gives the experimental results and data analysis. Section 5 discusses threats to validity. Section 6 summarizes the key findings and outlines future work.

2 Related Work

In a technical report, Hevner et al. debate how function extraction technology assists and significantly improves program comprehension (Hevner et al. 2005). In addition, Hevner et al. and Webb Collins et al. present empirical support for the importance of program comprehension during software development. A study performed in a software company showed that more than 28% of developers' time is devoted to learning and understanding their code or their colleagues' code. This fraction of time is a rough average of the program comprehension effort required in various development activities including system specification, system architecture, component design, component evaluation/selection, component implementation, component correctness evaluation, system integration, system testing, and system maintenance/evolution. The maintenance and evolution phase exhibited one of the highest percentages of programmers' effort spent on understanding system behavior (40%; Hevner et al. 2005; Webb Collins et al. 2008).

The idea for this family of experiments was also influenced by previous work in the DSL community. In particular the following statements were important motivators of the hypotheses posed in this study:

– "Because of appropriate abstractions, notations and declarative formulations, a DSL program is more concise and readable than its GPL counterpart. Hence, development time is shortened and maintenance is improved." (Consel and Marlet 1998)
– "Domain experts themselves can understand, validate, and modify the software by adapting the domain-specific descriptions. Modifications are easier to make and their impact is easier to understand." (van Deursen and Klint 1998)
– "There are lots of advantages to using DSLs, starting with the fact that programs are generally easier to write, reason about, and modify compared to equivalent programs written in general-purpose languages." (Hudak 1998)
– "DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs." (van Deursen et al. 2000)

In a related empirical study comparing a DSL to a GPL, Kieburtz et al. had four engineers construct message translation and validation modules for military command, control, communications and information systems (C3I) using both a DSL and a GPL (Ada). The participants also performed 112 maintenance tasks. The participants solved almost three times as many tasks when using the DSL as they did with the GPL. In addition, the participants developed twice as many correct solutions when using the DSL than when using the GPL (Kieburtz et al. 1996). While these results are promising, they are limited to a small sample (four participants) doing program development on one domain. Our study expands on these results by moving from program development to program understanding, from one domain to multiple domains, and from four participants to more than 70 participants (across the family of experiments).

Ricca et al. (2010) presented a controlled experiment aimed at assessing the comprehension of requirement specifications, which can also be regarded as DSLs. In particular, they were interested in the distinction between graphical and textual notations. Conversely, our experiment focused on textual DSLs and GPLs. The controlled experiment by Ricca et al. indicates a clear improvement in the comprehension of requirements specifications when graphical notations are present.

An earlier study by two of the authors of this paper tested various DSL implementations for the same representative language. The results show that approaches differ in terms of the amount of effort needed to implement DSLs (Kosar et al. 2008). Our current study expands upon these results by focusing on the program comprehension of the end-user of a DSL rather than of the designer of a DSL.

We have already performed some smaller-scale preliminary work in this area (Kosar et al. 2009, 2010). The earlier papers presented the results from only one study rather than from a family of studies, as in the current paper. In addition, in the earlier work, we performed the evaluation using a cognitive dimension framework (Green and Petre 1996) rather than statistical analysis, as in the current paper. While in our earlier work, the focus was on how particular cognitive dimensions contribute to the questionnaires' success for the chosen DSL and GPL (Kosar et al. 2010), in the current paper the focus is on a statistical evaluation of whether DSLs provide developers with better program comprehension compared with GPLs. Kosar et al. studied only one domain (graph descriptions), while in the current paper, we study three domains.

To avoid questionable results and to enable the replication of this research, we have followed published guidelines on conducting a family of experiments (Basili et al. 1999). These guidelines focus on building knowledge about an object of study by organizing a set of related studies (e.g. a family). These studies contribute knowledge about hypotheses that are consistent across the family of studies. Following these guidelines, before the study we defined: the study objective, the experiment hypotheses, the comparison validity, and the measurement framework.

The use of students as participants in experiments is widely recognized. Sjoeberg et al. report that 50% of the 2,969 experiments in 12 leading software engineering journals and conferences between 1993 and 2002 used undergraduate students as participants (Sjoeberg et al. 2005). Carver et al. define a model for conducting a valid empirical study with students (ESWS). They identify research and pedagogical requirements that need to be managed while preparing and executing an experiment in a university course. In short, researchers have to make sure that the study is well-integrated with the course goals and materials, give realistic time estimates for experimental tasks, properly motivate the participants without revealing the goals, measures and analysis prior to the study, allow students to give feedback, convince the participants of the relevance of what they are learning, avoid conflicts with students' other commitments, and give students feedback on the results of the experiment (Carver et al. 2010).

Besides these requirements, Carver et al. also provide a checklist to explain when the various activities should occur (i.e., before starting the study, as soon as the study begins, during the study, or after the study is completed). The requirements and checklist provide a useful guide for judging how well a study is integrated into the university course and for judging the reliability of the results. We used that checklist to verify the research and pedagogical goals in this study.

3 Experiment Design

To organize the discussion of the experimental design, this section follows the reporting guidelines proposed by Jedlitschka et al. (2008).

3.1 Experimental Goal

The goal of the experiments is to compare programmers' comprehension of programs written in a DSL vs. their comprehension of programs written in a GPL. To provide more validity to the results, we conducted a family of three experiments, each using a different problem domain:

Experiment 1	Feature diagrams (FD) domain: provides the ability to capture the variable parts of application domains and provides ways to express which features make up a given system.
Experiment 2	Graph descriptions (GD) domain: describes directed and undirected graphs, which consist of nodes and edges. Graphs, subgraphs, nodes and edges have various properties (attributes), e.g., color, shape, and style.
Experiment 3	Graphical user interface (GUI) domain: focuses on construction of programs that allow users to visually interact. GUIs are made up of several components, e.g., windows, menus, fields, labels, and buttons.

These domains all have existing DSL and GPL solutions (see Table 1). FDL (Feature Description Language) (van Deursen and Klint 2002) provides a textual description for FD, a technique used in Feature Oriented Domain Analysis (Kang et al. 1990; Czarnecki and Eisenecker 2000). On the GPL side, there is a Java

Table 1 Languages used in experiments

Domain	DSL	GPL
Feature diagrams (FD)	FDL	FD library in Java
Graph descriptions (GD)	DOT	GD library in C
Graphical user interface (GUI)	XAML	Windows forms library in C#

application library for the FD. For the GD domain, the DSL is a plain text description language (DOT) (Gansner et al. 2009), while the GD library in C is the GPL. For the GUI domain, the eXtensible Application Markup Language (XAML) (MacVittie 2006) is the DSL. XAML is an XML-based language for developing graphical user interfaces in the Windows Presentation Foundation and Silverlight applications of the .NET Framework, version 3.5. C# Forms is a comparable GPL for constructing graphical user interfaces.

3.2 Participants

All participants were undergraduate students in the Computer Science program at the University of Maribor. Experiments 1 and 2 were conducted as part of the Compiler Construction course. The focus of this course made it a good fit for the topic of the experiment. Also, most of the application domains used in the experimental tasks were related to the course materials. Experiment 3 was conducted in a combination of two courses: the Object and Functional Programming course and the Evolutionary Computation course. These courses had fewer students so both were necessary to obtain a sufficient number of participants. Both of courses follow the Compiler Construction course in the curriculum and were therefore suitable venues for the experiment.

The students performed the experimental tasks as part of course assignments. Following good practice (Sjoeberg et al. 2005), the participants were rewarded with points for participating in the experiment. Based on the correctness of their answers, the participants earned up to a 10% bonus on their assignment grade for the course.

3.3 Materials

Section 3.3.1 describes the artifacts used in the study. Then, Section 3.3.2 describes the data collection instruments.

3.3.1 Artifacts

Each of the three problem domains contained four specific applications on which the participants answered questions, as shown in Table 2. A brief description of each application is provided here.

– FD domain: provides ways to express how a system is composed and gives the ability to produce all possible configurations of a given problem (Czarnecki and

Table 2 Application used in experiments

	Feature diagrams (FD)		Graph descriptions (GD)		Graphical user interface (GUI)	
	DSL	GPL	DSL	GPL	DSL	GPL
Application 1	Menu	Radio	Compiler parts	UML	Registration form	Picture viewer
Application 2	Software	Compiler construction	Branching	Flowcharts	Product info	Painter

Eisenecker 2000). The following FD applications were used in experimental tests:

  - Menu: provides a customer all possible choices of restaurant menus;
  - Software: shows different options in the software development process (inclusion/exclusion of specific phases, etc.);
  - Radio: shows all possible configurations of a radio with different functionalities; and
  - Compiler construction: represents different options for constructing a compiler.

- GD domain: provides programs to create graphs. We defined the following GD applications for the experimental tasks:
  - Compiler parts: visually presents the compiler, its parts (source code, object code, etc.) and the process of obtaining a result (linguistic analysis, code generation, code optimization, etc.);
  - Branching: implements a special diagram-like mathematical game where branches in a diagram bring many different results of arithmetic equation;
  - UML: creates a UML class diagram; and
  - Flowcharts: creates a figure that represents an algorithm in a flowchart notation.
- GUI domain: provides programs to construct graphical user interfaces for the following applications:
  - Registration form: provides a registration form that can be used to register a new user;
  - Product info: shows basic information about a product for sale (e.g. product name, dimensions, price, and picture);
  - Picture viewer: allows users to view pictures and use different functionalities from menus (File, Edit, View, etc.); and
  - Painter: allows users to draw pictures using different shapes (line, brush, box, etc.) and colors on the Painter's canvas.

3.3.2 Data Collection Instruments

Each experiment consisted of a background questionnaire, two tests and a feedback questionnaire. The remainder of this section describes each of those artifacts in more detail.

Background Questionnaire The background questionnaire measured the participants' prior experience with each domain (FD, GD, and GUI) and with each application. The questionnaire contained ten questions. Each question used a five-point Likert scale (Likert 1932) with 1 representing low and 5 representing high. The background questionnaire is included in Appendix A.

Tests Each of the 12 application domains had its own program comprehension test consisting of 11 questions. The tests focused on the component implementation and system maintenance/evolution phases of the software lifecycle. These phases require the most program comprehension activities (Hevner et al. 2005). From ISO

Standard 9126, three types of cognitive activities are fundamental to the usability and quality of code, especially during modification: *learnability*, *understandability*, and *changeability*. These three aspects are strongly related to the extent to which a programmer's mental model of the software matches and predicts the actual behavior of the software. Each test contained questions that covered each of these cognitive activities. For the remainder of the paper we refer to these questions as *learn*, *understand*, and *evolve* questions.

The *learn* questions relate to learning the notation and meaning of the programs. The *understand* questions relate to program understanding, such as the identification of: the correct meaning of the program, the meaning of language constructs, the meaning of new constructs, or the meaning of programs with comments. The *evolve* questions require the participants to add, delete, or replace functionality. To ensure consistency across all tests, we defined the following questionnaire template. The questions for all tests followed this template. All tests used the same question order.

- Learn
  - Q1 Select syntactically correct statements.
  - Q2 Select program statements with no sense (unreasonable).
  - Q3 Select a valid program with the given result.
- Understand
  - Q4 Select the correct result for the given program.
  - Q5 Identify language constructs.
  - Q6 Select a program with the same result.
  - Q7 Select the correct meaning for the new language construct.
  - Q8 Identify correct meaning in the program with comments.
- Evolve
  - Q9 Expand the program with new functionality.
  - Q10 Remove functionality from the program.
  - Q11 Change functionality of the program.

The *learn* and *understand* question were multiple choice questions. Each question first described a task, such as "select the correct result for the given program." Next the question presented a program excerpt or a description of a program's meaning. Finally, the question contained five choices in terms of programs, program meanings or given answers (see the example of the first XAML *understand* question in Fig. 1).

The *evolve* questions were essay questions where the participants had to add code to a given code excerpt. The first part of the question contained a program for the participants to study. Upon understanding the existing program and the task to be completed, the second part of the question asked the participants to write their solution in a given box (on paper).

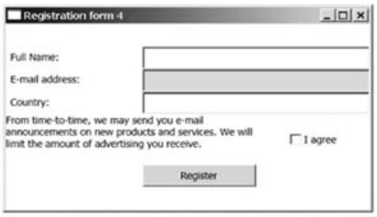
All tests were piloted twice before the actual experiment. First, domain experts reviewed the questions and the answer choices (programs) to obtain code that was as optimal as possible. Second, a small group of students completed the tests and provided feedback to improve and refine the questions before use in the actual experiment.

Question 7 **Marks: 0 / 1**

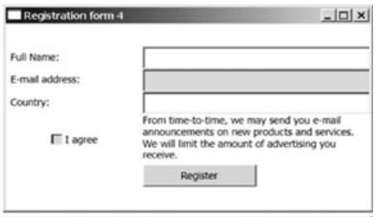
QP011 XAML-Registration: Select the correct result for the given XAML program:

```
<Window x:Class="WpfRegistration.Registration4a"
    Title="Registration form 4" Height="240" Width="400">
    <Grid ShowGridLines="False">
        <Grid.ColumnDefinitions>
         <ColumnDefinition Width="10*"/><ColumnDefinition Width="2*"/>
         <ColumnDefinition Width="10*"/><ColumnDefinition Width="10*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
         <RowDefinition Height="10*"/> <RowDefinition Height="10*"/>
         <RowDefinition Height="10*"/> <RowDefinition Height="10*"/>
         <RowDefinition Height="22*"/> <RowDefinition Height="10*"/>
         <RowDefinition Height="10*"/>
        </Grid.RowDefinitions>
        <Label     Grid.Column="0" Grid.Row="1">Full Name:</Label>
        <TextBox   Grid.Column="2" Grid.Row="1" Grid.ColumnSpan="2"/>
        <Label     Grid.Column="0" Grid.Row="2">
           E-mail address:</Label>
        <TextBox   Grid.Column="2" Grid.Row="2" Grid.ColumnSpan="2"
                   Background="LightGray" />
        <Label     Grid.Column="0" Grid.Row="3">Country:</Label>
        <TextBox   Grid.Column="2" Grid.Row="3" Grid.ColumnSpan="2"/>
        <TextBlock Grid.Column="2" Grid.Row="4" Grid.ColumnSpan="2"
                   TextWrapping="Wrap">
        From time-to-time, we may send you e-mail
        announcements on new products and services.
        We will limit the amount of advertising you
        receive.
        </TextBlock>
        <CheckBox Grid.Column="0" Grid.Row="4" Grid.ColumnSpan="2"
                  VerticalAlignment="Center"
                  HorizontalAlignment="Center"
                  Background="LightGray">
            I agree
        </CheckBox>
        <Button   Grid.Column="2" Grid.Row="5">Register</Button>
    </Grid>
</Window>
```

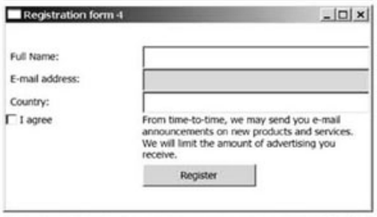
Choose one answer.

○ a.

Registration form 4

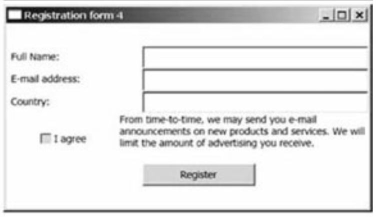
Full Name:
E-mail address:
Country:
From time-to-time, we may send you e-mail announcements on new products and services. We will limit the amount of advertising you receive.
☐ I agree
Register

○ b.

Registration form 4

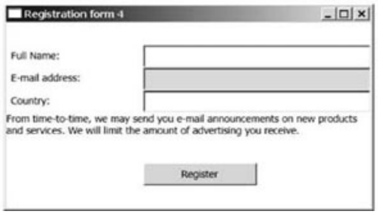
Full Name:
E-mail address:
Country:
☐ I agree
From time-to-time, we may send you e-mail announcements on new products and services. We will limit the amount of advertising you receive.
Register

○ c.

Registration form 4

Full Name:
E-mail address:
Country:
☐ I agree
From time-to-time, we may send you e-mail announcements on new products and services. We will limit the amount of advertising you receive.
Register

○ d.

Registration form 4

Full Name:
E-mail address:
Country:
☐ I agree
From time-to-time, we may send you e-mail announcements on new products and services. We will limit the amount of advertising you receive.
Register

○ e.

Registration form 4

Full Name:
E-mail address:
Country:
From time-to-time, we may send you e-mail announcements on new products and services. We will limit the amount of advertising you receive.
Register

Fig. 1 Example DSL question from the GUI test (XAML)

Feedback Questionnaire The feedback questionnaire measured the participants' individual perspectives on the experiment. The eight questions on this questionnaire used a five-point Likert scale to focus on two issues. First, the participants indicated how well they were able to comprehend the DSL and GPL programs. Second, they rated the experimental tasks relative to clarity, consistency, and DSL vs. GPL comparability. In this paper we analyze only the data relative to the simplicity of use of the DSL and GPL and the participants' opinion on the complexity of the tests.

3.4 Tasks

The following procedure was followed to conduct the experiments:

– Learning 1: The experimenters introduced the participants to the domain through a presentation.
– Learning 2: The experimenters introduced either the GPL API or the DSL notation (depending on which order the participant was following) to the participants with a presentation. Additionally, the experimenters discussed a program example with the participants.
– Experiment step 1: The participants solved either the GPL or DSL experimental tasks (whichever one was covered in Learning 2) using the learning materials and the language/API manual.
– Learning 3: The experimenters introduced either the DSL notation or the GPL API (whichever was not used in Learning 2) to the participants with a presentation. In addition, the experimenters discussed a program example with the participants.
– Experiment step 2: The participants solved either the GPL or DSL experimental tasks (whichever was covered in Learning 3) using the learning material and the language/API manual.
– Questionnaires: After finishing both the DSL and the GPL tests, the participants completed the background and feedback questionnaires.

3.5 Hypotheses and Variables

As the basis for this study, we formulated two one-tailed hypotheses, one on correctness and one on efficiency. These hypotheses were motivated by claims that a notation closer to the problem domain increases programmers' comprehension (Hudak 1998; van Deursen and Klint 1998; Consel and Marlet 1998), as discussed in Section 2. Moreover, because only the relevant domain concepts have to be recognized when using a DSL, the comprehension efficiency should increase. Therefore, the alternate hypotheses are one-tailed suggesting that DSLs will be more effective and efficient that GPLs. The specific hypotheses which were tested individually on each domain are:

$H1_{null}$ There is no significant difference in the correctness of the participants' program comprehension when using a DSL vs. when using a GPL.
$H1_{alt}$ The use of a DSL significantly increases the correctness of the participants' program comprehension over the use of a GPL.
$H2_{null}$ There is no significant difference in the efficiency of the participants' program comprehension when using a DSL vs. when using a GPL.

$H2_{alt}$ The use of a DSL significantly improves the efficiency of the participants' program comprehension over the use of a GPL.

To analyze these hypotheses, we defined five independent variables and five dependent variables. For each variable we provide its definition and levels (where applicable) as follows:

- Independent Variables
  - Language Type:
    - DSL: Domain-Specific Language
    - GPL API: General-Purpose Language with an API
  - Domain:
    - FD: Feature Diagram
    - GD: Graph Descriptions
    - GUI: Graphical User Interface
  - Question Number: 1–11
  - Question Type:
    - Learn: questions related to learning the notation and meaning of the program
    - Understand: questions related to program understanding
    - Evolve: questions that require addition of code
  - Experience: measured with participants' self-evaluation on a 5-point Likert scale (1 = no experience ... 5 = expert)
- Dependent Variables
  - Program Comprehension: the percentage of correct answers on the test
  - Time: number of minutes to complete the exam
  - Efficiency: percentage of correct answers divided by number of minutes
  - Simplicity of Use: measured on a 5-point Likert scale (1 = very hard ... 5 = very easy)
  - Test Complexity: measured on a 5-point Likert scale (1 = very hard ... 5 = very easy)

Figure 2 summarizes the overall study design.

3.6 Procedure

Before the experiments started, some basic rules were defined to rigorously prepare the experiment environment (Basili et al. 1999).We took the actions presented below into consideration when executing the experiments to reduce the threats to validity.

Order of the Tests Each experiment was conducted twice, at a different time, on different participants and in a different course (see Table 3). During the first execution, the participants first completed the DSL test and then completed the GPL test. Conversely, during the second execution the participants first completed the GPL test and then completed the DSL test. In such a way, we avoided the

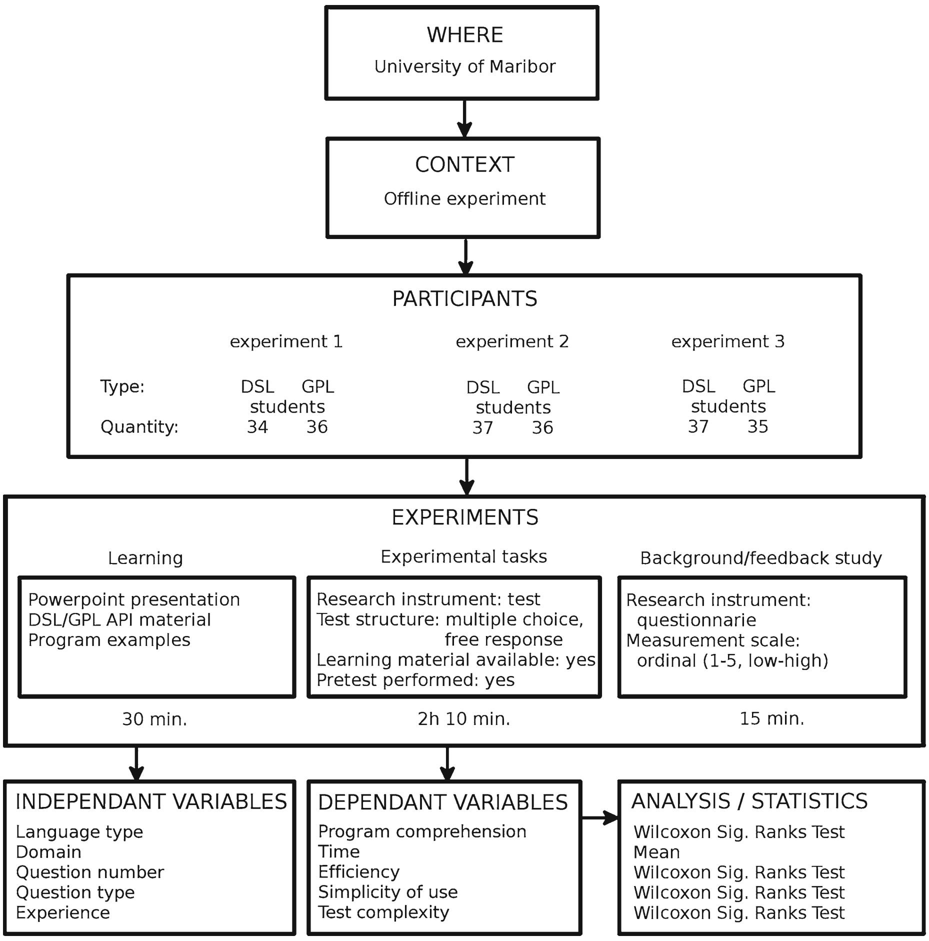


Fig. 2 General overview of experiments

introduction of an order effect related to the language type (DSL/GPL). Students from courses 1 and 2 participated in two experiments (FD and GD), while students in courses 3 and 4 participated only in the GUI experiment. All participants used both a DSL and a GPL with API. The only difference is the order in which the languages were presented.

In each case (DSL and GPL), the participants completed tests on two application domains at the same time. For example, in the FD domain for DSL the participants answered 11 questions about the Menu program and 11 questions about the Software

Table 3 Overview of experiment operation

	Procedure	Experiment 1	Experiment 2	Experiment 3
First execution	DSL $\rightarrow$ GPL	Course 1	Course 2	Course 3
Second execution	GPL $\rightarrow$ DSL	Course 2	Course 1	Course 4

program for a total of 22 questions. To preserve the question ordering, the questions from the two application domains were interleaved. That is Q1 from application 1 was followed by Q1 from application 2, and so on. The experiment tests can be found on the project homepage.[1]

Experiment Duration Both tests (DSL and GPL) in a single experiment were conducted in the same day. Course meetings last for three hours. The courses in which the experiments were conducted occurred at the end of the day. We reserved two hours of additional time giving the participants a total of five hours to complete the study including learning, questionnaires and both tests. On average the participants finished both tests in two hours and ten minutes. The participants did not feel any pressure to finish, because there was plenty of time remaining.

Presentations These presentations occurred before the participants began the experimental tasks. First, we presented the participants with a short tutorial on the problem domain (FD, GD, or GUI) including its basic constructs and details about the importance of the domain. Then we illustrated the domain with an example application. Next, we gave the participants a tutorial in the language type they were going to use (either DSL or GPL API) along with an example program. The slides, programs and experiment tests were in English. But, we presented all of the tutorials in the participants' native language (Slovenian). The participants received the slides, the tutorial (domain, DSL, and GPL API), and the case study applications on paper to use as a reference during the experiment. By providing the participants with these presentations, we sought to reduce the potential impact of experience and knowledge on program comprehension. By making the participants more familiar with the application domain and the DSL or GPL API, there was less chance that the lack of knowledge would negatively affect their program comprehension. Even with these presentations, we could not completely eliminate the effect of experience and knowledge. As a result, we investigated the impact of these factors with the background questionnaire described in Section 3.3.2.

Test Completeness When the participants submitted their tests, first we checked to ensure that the test was completed within the assigned time period. Most participants completed all 22 questions (11 on each test). If one or more questions were unanswered and the participant was still in the room, we asked whether the participant intentionally left the questions unanswered. If not, we asked the participant to complete the missing questions. When several participants submitted their tests at the same time it was impossible to check all of them in real-time. We removed the data from any participant who was missing two or more answers. If only one answer was missing, and it was not intentional, we invited the participant to our office and gave them an opportunity to complete the missing answer. In no case was a participant forced to answer a question. Using this approach, we only eliminated 10 submissions out of the 108 received.

Relative to the time taken for the tests, we noticed that some participants did not consistently note the start and end times for the program comprehension tests.

[1] http://lpm.uni-mb.si/index.php?page=ProjectPC

Because it was easy to observe if the start or end time was missing from the test (first and last page of the program comprehension test), we were able to advise most of the participants to complete the missing data. If the information was still missing, we were able to accurately fill in the missing data ourselves. The starting time was easy to complete because all participants started at the same time. The ending time was a bit more complex to fill in because the participants ended at different times. We kept the tests in the order in which they were submitted, so we approximated the missing end time for a participant by assigning him the completion time for his immediate predecessor. Again, five participant's submissions at most were missing those two pieces of information across all three experiments.

4 Experiment Results

This section compares the program comprehension results for the DSL and the GPL language types. It also presents the results from the background and feedback questionnaires. All the observations were statistically tested with a $\alpha = .05$ as a threshold for judging significance. *SPSS Statistics 17.0* was used to perform the statistical analysis.

Overview of Results Table 4 summarizes the basic data from the experiments: the background questionnaire, the program comprehension tests for the DSLs and the GPLs, and the feedback questionnaires. The "Experience" row shows the results of the participants' self-evaluation of their knowledge (questions B2 and B3 in Appendix A on a 5-point Likert scale: 1 = never used, . . . 5 = expert). These results indicate that the participants were more experienced in the GPLs than they were in the DSLs. The "Correctness" row shows that, based on the program comprehension tests (questions Q1–Q11 in Section 3.3.2), the participants performed better when using DSLs than when using GPLs. The "Time" row shows that the participants finished the tasks faster when using the DLSs. Finally, the "Simplicity of Use" row shows that the participants found the DSLs easier to use than the GPLs (questions F1 and F2 in Appendix B on a 5-point Likert scale: 1 = very hard, . . . 5 = very easy). Note, that because some incomplete tests were eliminated, the number of participants (column N) is different for the GPL and the DSL program comprehension tests. Also, not all participants submitted the feedback and background questionnaires, so the number of participants N for "Experience" and "Ease of Use" is less than for "Correctness" and "Efficiency".

The remainder of this section provides the statistical analysis of these results for each experiment and relates those results back to the hypotheses defined in Section 3.5.

Table 4 Descriptive statistics

Measures	GPLs (C, C#, Java)				DSLs (FDL, DOT, XAML)			
	N	Median	Mean	Std. dev.	N	Median	Mean	Std. dev.
Experience	91	2.0	2.27	1.38	92	1.0	1.39	0.73
Correctness	107	50.0%	53.00%	20.75%	108	71.59%	70.18%	16.68%
Time	106	1:17:00	1:17:43	0:27:07	108	0:52:00	0:56:50	0:20:53
Simplicity of use	91	3.0	2.88	0.92	92	3.0	3.32	0.93

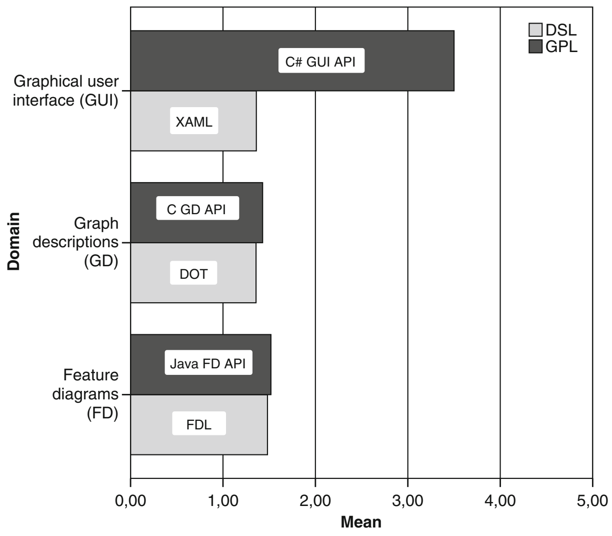


Fig. 3 Prior experiences in the domains

4.1 Participants' Experience

Participant experience level, gathered with the background questionnaire, is an important factor in these experiments. Specifically, the participants' domain experience, which was elicited with questions B5 and B6 in Appendix A, was important. Figure 3 shows the average level of experience the participants reported for each domain. The participants had low experience in the FD and GD domains. In the GUI domain, the participants had considerably more experience with C# Forms (the GPL) than they did with XAML (the DSL).

Table 5 shows the results of the Wilcoxon Signed Ranks Test (Wilcoxon 1945) to evaluate the experience gap between the GPL and the DSL language type for each domain. Because we did not have a hypothesized direction of effect, these tests were two-tailed. As expected from Fig. 3, there is no significant difference for the FD and GD domains, meaning that is unlikely that domain experience had an impact on the results. Conversely, for the GUI domain, the participants are significantly more familiar with the GPL API than with the DSL (the mean rank for the GPL was 17 and for the DSL was 0). In the GUI domain, if domain familiarity did impact effectiveness, it would bias the results against the study hypotheses (i.e. DSLs make

Table 5 Comparison of background experience in DSLs vs. GPLs

		N	Mean	St. Dev.	Median	Mean rank	Z	p–value
FD	FDL	27	1.48	0.75	1	6	−0.577	0.564
	FD API	27	1.52	0.80	1	4.5		
GD	DOT	29	1.34	0.77	1	1.5	−0.816	0.414
	GD API	28	1.43	0.92	1	2.25		
GUI	XAML	36	1.36	0.68	1	0	−5.082	<**0.001**
	GUI API	36	3.5	1.11	4	17		

Significant p-values provided in bold

developers more effective and efficient than GPLs) rather than towards it. Therefore, this difference does not present a serious threat to validity.

4.2 Analysis of Results

This section analyzes the DSL and GPL program comprehensions tests. For the purpose of statistical analysis, all questions come from the list of questions Q1–Q11 described in Section 3.3.2. The data were analyzed separately for each experiment.

For program comprehension, the box-plot in Fig. 4 shows the percentage of questions the participants correctly answered in each domain. The upper whiskers of the box-plots indicate that at least one participant was able to answer all questions correctly in the FD and GD domains. Conversely, no one correctly answered all questions for either language type in the GUI domain. In all three experiments, the participants using the DSL performed better than those using the GPL. The correctness scores were very similar for the FD and GD domains. While for the GUI domain, the overall correctness scores were lower, but the magnitude of the difference between the DSL and the GPL language types was similar to the magnitude in the FD and GD domains. The results for the GUI domain are a bit surprising because the participants reported higher experience with the GUI domain than with other two domains (see Fig. 3). We expected that more experience would result in higher correctness. It is possible that this unexpected result is influenced by the fact that the GUI domain is larger than the FD and GD domains and the GUI programs included in the experimental tasks were larger. For example, the average lines of code in the programs for the DSL tests were 4, 13 and 22 for the FD, GD and GUI domains respectively.

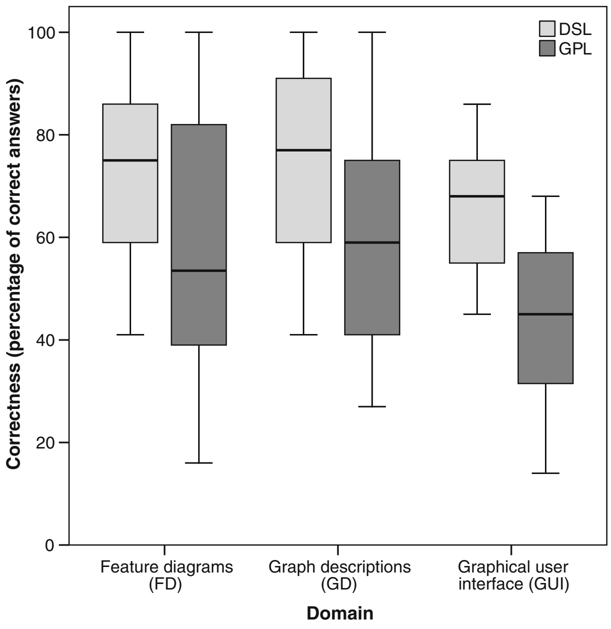


Fig. 4 Correctness by domain

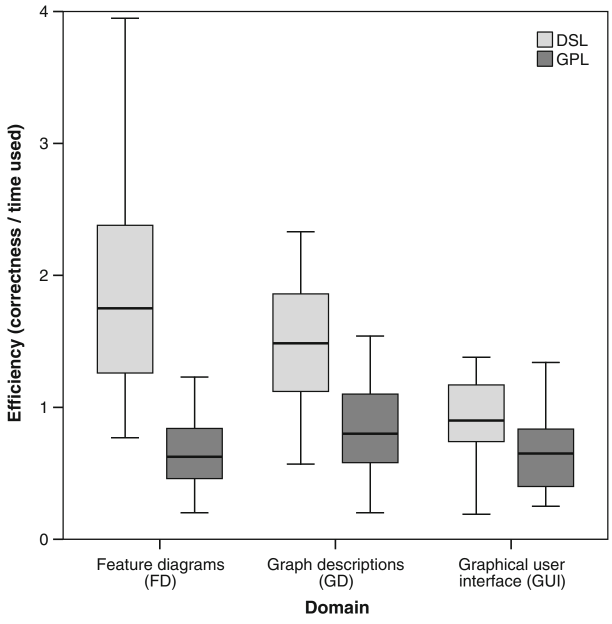


Fig. 5 Efficiency by domain

The most intuitive way to measure efficiency would be by observing the amount of time that participants spent on the tests. However, this approach is questionable, because some participants could randomly fill-in the test and spend far less time than those who perform the tasks properly (Nugroho 2009). As a result, another common way to measure efficiency is to compute the ratio of correct answers to time (Cruz-Lemus et al. 2009; Otero and Dolado 2004). Following this advice, in our study, efficiency is measured as the ratio of the percentage of correctly answered questions to the amount of time spent answering the questions. Figure 5 shows that, for all three domains, the participants' comprehension efficiency on DSLs is higher than their comprehension efficiency on GPLs.

To have confidence in the observations made from Figs. 4 and 5, we performed a statistical analysis to evaluate whether the observed differences were significant. Each participant provided data for both the DSL treatment and the GPL treatment, necessitating the use of between-subjects analyses. Prior to conducting the analysis, we tested the sample for normality using the Shapiro-Wilkes test. The results indicated that the overall sample was not normally distributed. As a result, we used the Wilcoxon Signed Ranks test to evaluate the observed differences. Because our hypotheses were in favor of DSLs, we used one-tailed tests in all cases. The results in Table 6 show that for all three domains, the participants using the DSL were significantly more effective and significantly more efficient than those using the GPL. These results allow us to reject both null hypotheses $H1_{null}$ and $H2_{null}$ and accept the alternative hypotheses: ($H1_{alt}$): *the use of domain-specific language significantly improves participants' program comprehension correctness* and the second alternative hypothesis $H2_{alt}$: *the use of domain-specific language significantly improves participants' program comprehension efficiency*.

Table 6 Wilcoxon signed ranks test for correctness and efficiency by domain

			N	Mean	Std. Dev.	Mean rank	Z	p–value
FD	Correctness	DSL	34	72.79%	16.56	18.20	−4.058	<**0.0005**
		GPL	36	56.50%	22.87	6.83		
	Efficiency	DSL	34	1.84	0.72	17.00	−4.918	<**0.0005**
		GPL	36	0.69	0.31	1.00		
GD	Correctness	DSL	37	73.71%	17.21	18.32	−4.559	<**0.0005**
		GPL	36	58.84%	19.77	9.00		
	Efficiency	DSL	37	1.44	0.44	18.94	−5.078	<**0.0005**
		GPL	36	0.81	0.33	2.50		
GUI	Correctness	DSL	37	64.25%	14.97	18.90	−4.613	<**0.0005**
		GPL	35	43.38%	16.02	7.00		
	Efficiency	DSL	37	0.94	0.28	20.24	−4.455	<**0.0005**
		GPL	35	0.67	0.29	7.17		

Significant p-values provided in bold

4.3 Program Comprehension Tests Individual Question Analysis

Table 7 presents the results from Q1–Q11 for the DSL and GPL language types. Each column is the average correctness for the two applications in the domain (see Table 2). By averaging the correctness values across both applications within a domain, the results for the individual questions are more reliable.

Q1 ("Select the syntactically correct statements") tested the participants' knowledge of GPL and DSL syntax in each domain. In the FD domain, the participants performed slightly better when using the GPL than when using the DSL. In the other two domains, the participants performed much better when using the DSL. The unexpected result for the FD domain can be explained by the participants' experience. Senior-level students are almost professionals and it was easier for them to find the syntax error in Java, which they knew, than in an FDL language, which they did not know before the experiment.

Table 7 Average correctness by question

Question	FD		GD		GUI	
group	DSL (N = 34)	GPL (N = 36)	DSL (N = 37)	GPL (N = 36)	DSL (N = 37)	GPL (N = 35)
Q1	89.71%	91.67%	83.78%	69.44%	72.97%	48.57%
Q2	79.41%	66.67%	44.59%	20.83%	35.14%	38.57%
Q3	75.00%	40.28%	75.68%	65.28%	64.86%	35.71%
Q4	81.62%	60.42%	87.84%	75.00%	77.03%	70.00%
Q5	57.35%	48.61%	72.97%	76.39%	64.86%	48.57%
Q6	63.24%	45.83%	62.16%	33.33%	39.19%	27.14%
Q7	76.47%	65.28%	85.14%	55.56%	75.68%	62.86%
Q8	39.71%	52.78%	63.51%	56.94%	62.16%	45.71%
Q9	77.94%	50.00%	72.97%	56.94%	79.73%	28.57%
Q10	73.53%	61.11%	90.54%	69.44%	68.92%	41.43%
Q11	86.76%	38.89%	71.62%	68.06%	66.22%	30.00%
Average	72.79%	56.50%	73.71%	58.84%	64.25%	43.38%

As mentioned in Section 4.2, the average correctness value for each domain (the bottom row in Table 7) indicates that the FD and GD domains had very similar results, while overall the participants were less effective in the GUI domain. A second reason for this difference (the first was mentioned in Section 4.2) is that the correctness for Q2 and Q6 was quite low in the GUI experiment (below 40%). Q2 ("Please select GUI program with no sense (unreasonable—incorrect GUI)") was difficult for the participants (note the poor results in both GUI tests). In question Q2, all given programs were correct (executable), however one program produced a semantically incorrect graphical user interface. Because the question did not involve the correct constitution of a GUI, it was very difficult for the participants to identify the incorrect program. There was a similar problem with Q6. While these observations are interesting, they ultimately do not threaten the validity of the overall experimental results because the poor performance on the GUI problems affected both the GPL and the DSL similarly. Comparing the results across all three domains, there are only 4 of 33 questions where the participants' correctness when using the GPL was better than when using the DSL: Q1 (FD), Q8 (FD), Q5 (GD), and Q2 (GUI).

These examples indicate why drawing general conclusions on the basis of individual questions can be extremely risky. Even when the questions are designed to be equivalent and evaluated by experts, some questions can be unequal between the DSL and GPL test, posing a threat to the validity of the results. Therefore, grouping questions is crucial to obtain more reliable results. Before the study, we defined three sets of questions: *learn*, *understand* and *evolve* (Section 3.3.2).

Table 8 presents the results of the Wilcoxon Signed Ranks test for each group of questions. Similar to the previous analysis, because our hypotheses were directional, we used one-way tests. In all cases, the results confirm the hypotheses that program

Table 8 Wilcoxon signed ranks test for correctness on learn, understand and evolve questions

			N	Mean (%)	Std. dev.	Mean rank	Z	p−value
FD	Learn	DSL	34	81.37	18.24	13.31	−3.107	**0.001**
		GPL	36	66.20	25.35	7.30		
	Understand	DSL	34	63.68	21.93	19.50	−1.634	**0.05**
		GPL	36	54.58	29.33	12.64		
	Evolve	DSL	34	79.41	23.95	14.75	−3.979	<**0.0005**
		GPL	36	50.00	28.73	8.00		
GD	Learn	DSL	37	68.02	24.34	17.38	−3.832	<**0.0005**
		GPL	36	51.85	24.81	8.00		
	Understand	DSL	37	74.32	22.18	16.04	−3.296	**0.0005**
		GPL	36	59.44	21.77	11.00		
	Evolve	DSL	37	78.38	19.19	16.88	−2.369	**0.009**
		GPL	36	64.81	33.04	9.10		
GUI	Learn	DSL	37	57.66	22.08	17.23	−3.033	**0.001**
		GPL	35	40.95	22.99	10.75		
	Understand	DSL	37	63.78	19.34	15.65	−3.107	**0.001**
		GPL	35	50.86	18.69	12.50		
	Evolve	DSL	37	71.62	20.36	16.93	−4.578	<**0.0005**
		GPL	35	33.33	26.20	10.00		

Significant p-values provided in bold

Table 9 The Wilcoxon signed ranks test on participants' feedback on simplicity of use

		N	Mean	St. dev.	Median	Mean rank	Z	p−value
FD	FDL	27	3.44	1.09	3	7.00	−3.213	**0.001**
	FD API	27	2.48	0.96	2	0.00		
GD	DOT	29	2.97	0.79	3	7.45	−2.140	**0.032**
	GD API	28	2.64	0.68	3	5.50		
GUI	XAML	36	3.50	0.85	3.5	10.80	−0.756	0.449
	GUI API	36	3.36	0.83	3	14.25		

Significant p-values provided in bold

comprehension in terms of *learn*, *understand* and *evolve* is better for DSL programs than for GPL programs.

4.4 Participant Feedback

The feedback questionnaires (Appendix B) asked the participants about certain aspects of the program comprehension tests, such as simplicity of use of the DSL and the GPL (question F1 and F2), the complexity of the tests (question F3 and F4), and the clarity of programs used in the tests (question F5–F8). In this paper, we focus our analysis on questions F1–F4 because they reveal the most important information related to the study focus.

After completing both tests, the participants used the feedback questionnaire to express their opinion about the simplicity of the DSL and the GPL. The last row of Table 4 shows that the participants' only rated the DSLs one-half point simpler than the GPLs (GPL Mean = 2.88 vs. DSL Mean = 3.32). This result was surprising because we expected the difference to be much larger, especially considering the difference in effectiveness discussed in Section 4.2. Table 9 separates the results from Table 4 by domain. The participants preferred the DSL in all three domains, but the difference was only significant in the FD and GD domains.

Questions F3 and F4 focused on the participants' comparison of the complexity of the DSL and GPL tests for each domain. Although, during the planning phase, we devoted a lot of effort to this issue (test constitution, equal experimental tasks complexity, order of tests, etc.), these questions provided the participants' point of view on the complexity of tests. The statistical analysis in Table 10 confirms that there are no significant differences in test complexity for the GD and GUI domains. However, for the FD domain the DSL was rated as significantly more complex than the GPL ($p = 0.002$). Since the results for correctness and efficiency were

Table 10 The Wilcoxon signed ranks test on participants' feedback on tests complexity

		N	Mean	St. dev.	Median	Mean rank	Z	p−value
FD	FDL	27	3.15	1.13	3	8.93	−3.038	**0.002**
	FD API	27	2.33	0.92	2	5.50		
GD	DOT	28	3.00	0.80	3	7.75	−0.632	0.527
	GD API	28	2.89	0.86	3	7.17		
GUI	XAML	36	3.00	0.89	3	7.30	−0.267	0.790
	GUI API	36	2.92	1.03	3	10.50		

Significant p-values provided in bold

significantly better for the DSLs, this difference in complexity does not present a serious threat to the validity of the results in the FD domain. Also, it is possible that some participants did not understand the meaning of the question "to compare DSL and GPL tests' complexity" (F3/F4). In the future, we will combine questions F3 and F4 so the participants will have to clearly decide whether the complexity of the DSL and the GPL tests are equal.

5 Threats to Validity

In this section we discuss the threats to validity of the results of the experiments. We discuss the construct validity, internal validity and external validity threats that we addressed as well as those that we were unable to address.

5.1 Construct Validity

There are two main issues that affected the construct validity of these experiments: the applications we chose for each domain and the specific questions we included on the tests for those applications.

First, for each domain, we chose two applications for the DSL language type and two different applications for the GPL language type. This decision addresses one validity threat but creates another one. By averaging the results from the two applications for each domain and language type combination (i.e. FD DSL), we have reduced the threat that the observed results were caused by the specific applications rather than by the language type. Conversely, because we used different applications for the DSL and GPL language types, we cannot be sure that the observed differences between the language types were due solely to the language type rather than to the specific applications. This threat is mitigated by the fact that across the three domains there are six applications for DSL and six for GPL. It is unlikely that all six DSL applications would somehow be different from all six GPL applications.

Second, for question complexity, there were four applications for each domain (two DSL applications and two GPL applications as shown in Table 11). The test for

Table 11 Number of components involved in the GD domain questions

	DSL		GPL	
	Compilers	Branching	UML	Flowcharts
Q1	5	7	3	7
Q2	5	10	3	9
Q3	5	12	5	9
Q4	12	10	5	13
Q5	16	9	3	14
Q6	13	17	3	19
Q7	9	10	4	10
Q8	13	10	13	16
Q9	14	7	15	13
Q10	15	24	16	16
Q11	17	24	13	20

each application had 11 questions. Our major concern was ensuring that the question complexity was similar so as not to introduce a threat to validity. LOC (Lines of Code) is usually not a good measure of complexity when comparing programs written in different languages, especially when they have a different abstraction level. Instead, we decided to measure something similar to function points, the functionality of programs. To measure functionality, we counted the total number of included components, the number of assignments and the number of different components. For instance, we tried to define the questions so that corresponding questions would have a similar number of components across all applications within a domain.

The number of components involved in the GD domain, as a representative example of all domains, can be seen in Table 11. In the case of the GD domain, we counted graphical elements (nodes, relationships and sub-graphs). Table 11 shows the number of components contained in the question or in the correct answer choice (if the question does not contain a program). Note, that the first applications in the DSL and the GPL were less complex with respect to number of components than the second applications. Q1 for the first DSL application contained five components, while Q1 for the first GPL application contained three components. The second DSL and GPL applications in question Q1 have the same number of components. Observing each question with respect to the first/second application in the GPL and the DSL tests reveals that the programs are comparable based on the number of components present. However, there are some deviations that resulted from creating reasonable questions/programs. Note, that the number of components is just a rough measure of complexity. Some components, such as labels, are much easier to use than others, such as menus.

Overall, we tried to ensure similarity in logical complexity (control and data flow) and physical complexity (number of constructs/components) between the DSL and GPL questions. Whenever we could not ensure this similarity due to the nature of a problem, we opted, in most cases, to bias the study towards the GPL by making the DSL programs more complex. Based on the participants' feedback, it appears that we successfully accomplished these goals. Participants did not indicate a significant difference in complexity for the GD and GUI domains. For the FD domain, the participants indicated that the DSL questions were more complex than the GPL questions (see Table 10). Since correctness and efficiency were higher for the DSLs than for the GPLs, the difference in question complexity does not introduce a serious threat to validity.

Questions Interaction Issue Additionally, we ensured that there was no interaction among the questions on a test. For example, if two successive questions are focused on the same concept, like a registration form in the GUI domain, we ensured that the correct answer for question 1 did not appear in question 2 either as part of the question or as an answer choice. By using this approach, we reduced a threat to validity that the participants answers to the questions would be based on other questions or answers on the test.

5.2 Internal Validity

Some of the participants also expressed their disapproval that the experiment was performed on paper rather than on the computer. They missed the basic, supplemental comprehension tools provided by IDEs like syntax highlighting, code treeviews, and inspectors that help with code analysis. We chose this offline approach because we wanted to measure the participants' program comprehension and avoid the influence of their experience with IDEs. However, it would be interesting to measure how program comprehension would change with the help of the IDE tools.

A GPL test has 54 pages on average while a DSL test has only 31 pages on average. As a result of including programs that were not trivial, there is a lot of code on each test. Taking into account that the largest GPL programs were only around 100 LOC, these programs can be considered as small programs. Using even larger programs would have resulted in splitting programs over several pages which could have influenced the results.

As discussed earlier, the experiments were an optional assignment in the course. In each experiment, some students did not participate (11 out of 45 in the FD experiment, 9 out of 45 in the GD experiment, and 11 out of 46 in the GUI experiment). The fact that all students did not participate represents a selection threat, because the effect on the final results if the rest of the students had participated is unknown.

From our notes, most of the students who chose not to participate also never showed up in the classroom (nine student in the first and second experiment and five in the third experiment). Most of these students, were already working in industry and a few of them were in the international student exchange program. We do not think that the students' absence was related to their ability to understand the course materials. The other students who did not participate in the experiment (eight in the FD experiment, seven in the GD experiment, and six in the GUI experiment) were most likely satisfied with their course grade and did not have a need for the extra points awarded for participation. We have no reason to believe that the students who chose not to participate significantly changed the sample. Therefore, excluding those students should not cause a problem with the results.

Another threat is related to the introductory lessons given to the participants. In both experiments, the instructors gave lessons about the DSL and the API (in the specific GPL). The instructor did not give an introduction to the GPL because the participants were students from later years of study and had already gained experience in C++, Java, and C#. Although the APIs and DSLs were similar in their extent of domain coverage, an API is likely more complex to learn than a DSL. Therefore only allowing 15 min for the introduction to each one may be questionable. We compensated for this lack of introduction by providing more explanation of the API during the experiment. These explanations were given to individual participants and were strictly limited to comprehension of the API. Even with this compensation, this situation still presents an internal threat to validity.

5.3 External Validity

The participants were students which typically poses a threat to external validity. In this case, most of the students were in their later years of study and already had industrial experience. In Slovenia, where the experiments were conducted, most computer science students start working early in their university career, because their employment is encouraged by the government. In fact, those students are quite experienced. Also, during the study, the students gained deep knowledge in certain domains through the experimental tasks. While these circumstances reduce the threat to external validity, they do not eliminate it, because the participants were not actual industrial practitioners using the DSL and GPL languages.

The size of the experimental tasks is another threat to external validity. The DSL and GPL programs used in the experiments were relatively small (the DSL programs contained less than 32 LOC and the GPL less than 104 LOC) and simple but not trivial. It is possible that if larger, more complex programs were used, the benefits of the DSL may not be as pronounced as in our results. This issue will require further study.

The representativeness of the chosen domains (FD, GD, and GUI) may pose a threat to external validity. We chose different sized domains to see if domain size had an influence on the results. The FD domain was fairly small, followed by the GD domain and finally the quite large GUI domain. Because the results were similar across domains, domain size does not seem to present a serious threat. Size is just one criteria by which domains can be compared. Further experimentation is needed to determine how other factors like domain completeness or existent formal analysis (e.g. defined terminology) affect the results.

Another related threat is whether the chosen DSLs and APIs are good implementations of the domains. For instance, a DSL can be implemented in various ways. Each implementation method has its own weaknesses and benefits for the end-users (Kosar et al. 2008). Also, APIs can be different in design, performance, documentation, etc. Additional experiments are required to test other DSLs and APIs for the selected domains.

The first row of Table 4 indicates that the participants are more experienced in the GPLs than in the DSLs. We were not surprised by this distribution of experience. One open issue with the results of the experiments is what the impact would have been if the participants had more experience with DSLs. However, we believe, that it would be difficult to find a representative sample of programmers for these experiments who would have equal experience in both DSLs and GPLs techniques. Therefore, this threat to validity was not addressed in our experiment. But, because the results indicated that participants performed better when using DSLs, we can only assume that if we had used participants with equal experience in DSLs and GPLs the results would have been even more favorable for DSLs.

Finally, there is a potential threat to validity that arises from the fact that the tests were taken on paper rather than on the computer. Our goal in the study was to test the comprehensibility of DSLs and GPLs without the help of tools. The reason for this choice was that the DSL tools are not as mature as the GPL tools. As a result, using tools support would provide a confounding variable for the study of the underlying differences between DSLs and GPLs. Therefore, we chose to use paper-based tests. It is possible that the results may differ if participants are allowed to use IDEs and other computer-based tools. This threat will be addressed by future studies that include IDEs and other tools.

6 Conclusions

This paper presents a family of three controlled experiments that analyze whether developers have better program comprehension when using a DSL compared with using an API in a GPL. The experiments evaluated a set of common hypotheses regarding correctness and efficiency of program comprehension. The main findings of these experiments are:

Comprehension correctness	Across all three experiments, the participants' correctness on the DSL programs was significantly better than their correctness on the GPL programs (see Table 6). We found a similar significant result when the *learning*, *understanding*, and *evolution* tasks were analyzed separately (see Table 8). The participants also indicated that the DSL was simpler in all cases. This result was significant in the FD and GD domains.
Comprehension efficiency	In all three experiments, the participants took less time to complete the DSL tests than they did to complete the GPL tests (see Table 4). In terms of efficiency (percentage of correct answers divided by the time required to complete the test), the participants were significantly more efficient when using a DSL than when using the corresponding GPL API (see Table 6).

The results of these experiments have important implications for the DSL community. The commonly-held belief that DSL programs are easier to understand and maintain than GPL programs had not previously been empirically validated in a controlled experiment. Hence, the aforementioned advantages of DSLs could be easily attacked by DSL skeptics as being mere speculation. This study provides clear empirical evidence to support the community belief about the benefits of DSLs. Due to the scale of the artifacts used, the results of this study should be valid for relatively small applications (the GD DSL programs did not exceed 32 LOC and the GD GPL programs were less than 104 lines of code). But, this size is typical for DSL programs, since they are tailored to narrow domains.

As with any empirical study, these results must be taken with caution. Additional replications of this study are needed, especially in industry. Larger domains and applications from real projects are also needed to strengthen the validity of the conclusions. In addition, we need to compare the program comprehension of DSLs and GPLs when writing programs from scratch. Tool support is also important for program comprehension. It would be interesting to see how the use of development tools affects the results of the family of experiments.

Acknowledgements Special thanks to Matej Črepinšek and Dainela da Cruz who helped us prepare the third experiment on graphical user interfaces. Also thanks to Portuguese partners (Pedro Rangel Henriques, Maria João Varanda Pereira, and Nuno Oliveira) for their constructive comments on questionnaires that helped us increase the quality of this work.

Appendix A: Background Questionnaires

B1 How would you rate your programming skill level?
B2 How would you rate your programming skills level in Java/C/C#?
B3 How would you rate your experience with domain-specific languages before the experiment?

B4	Are you familiar with domain1/domain2/domain3?
B5	Are you familiar with DSL of domain1/domain2/domain3?
B6	Are you familiar with the application library of domain1/domain2/domain3?
B7	Are you familiar with DSL application 1?
B8	Are you familiar with GPL application 1?
B9	Are you familiar with DSL application 2?
B10	Are you familiar with GPL application 2?

Appendix B: Feedback Questionnaires

F1	How simple for use does DSL 1/2/3 seem to you?
F2	How simple for use does GPL 1/2/3 seem to you?
F3	How would you grade complexity of the DSL 1/2/3 questionnaire?
F4	How would you grade complexity of the GPL 1/2/3 questionnaire?
F5	How well have you understood programs on DSL application 1?
F6	How well have you understood programs on GPL application 1?
F7	How well have you understood programs on DSL application 2?
F8	How well have you understood programs on GPL application 2?

References

Basili V, Shull F, Lanubile F (1999) Building knowledge through families of experiments. IEEE Trans Softw Eng 25(4):456–473

Bentley J (1986) Little languages. Commun ACM 29(8):711–721

Carver J, Jaccheri L, Morasca S, Shull F (2003) Issues in using students in empirical studies in software engineering education. In: METRICS '03: proceedings of the 9th international symposium on software metrics. IEEE Computer Society, Washington, DC, USA, p 239

Carver J, Jaccheri L, Morasca S, Shull F (2010) A checklist for integrating student empirical studies with research and teaching goals. Empir Softw Eng 15(1):35–59

Consel C, Marlet R (1998) Architecturing software using a methodology for language development. In: Proceedings of the 10th international symposium on programming language implementation and logic programming, vol 1490, pp 170–194

Cruz-Lemus JA, Genero M, Manso ME, Morasca S, Piattini M (2009) Assessing the understandability of UML statechart diagrams with composite states–a family of empirical studies. Empir Softw Eng 14(6):685–719

Czarnecki K, Eisenecker U (2000) Generative programming: methods, tools and applications. Addison-Wesley, Reading

van Deursen A, Klint P (1998) Little languages: little maintenance. J Softw Maint 10(2):75–92

van Deursen A, Klint P (2002) Domain-specific language design requires feature descriptions. J Comput Inf Technol 10(1):1–17

van Deursen A, Klint P, Visser J (2000) Domain-specific languages: an annotated bibliography. ACM SIGPLAN Not 35(6):26–36

Elliott C (1999) An embedded modeling language approach to interactive 3D and multimedia animation. IEEE Trans Softw Eng 25(3):291–309

Gansner ER, Koutsofios E, North S (2009) Drawing graphs with *dot*. Tech Rep AT&T Bell Laboratories, Murray Hill, NJ, USA. http://www.graphviz.org/pdf/dotguide.pdf

Green TRG, Petre M (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. J Vis Lang Comput 7(2):131–174

Hevner AR, Linger RC, Webb Collins R, Pleszkoch M, Prowell S, Walton G (2005) The impact of function extraction technology on next-generation software engineering. Tech Rep CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University

Hudak P (1998) Modular domain specific languages and tools. In: Proceedings: fifth international conference on software reuse. IEEE Computer Society Press, Los Alamitos, pp 134–142

Jedlitschka A, Ciolkowski M, Pfahl D (2008) Reporting experiments in software engineering. In: Shull F, Singer J, Sjæberg DIK (eds) Guide to advanced empirical software engineering. Springer, London, pp 201–228. doi:10.1007/978-1-84800-044-5_8

Kang K, Cohen S, Hess J, Novak W, Peterson S (1990) Feature-oriented domain analysis (FODA) feasibility study. Tech Rep CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University

Kieburtz RB, McKinney L, Bell JM, Hook J, Kotov A, Lewis J, Oliva DP, Sheard T, Smith I, Walton L (1996) A software engineering experiment in software component generation. In: ICSE-18: proceedings of the 18th international conference on software engineering. IEEE Computer Society, pp 542–553

Kosar T, Martínez López PE, Barrientos PA, Mernik M (2008) A preliminary study on various implementation approaches of domain-specific language. Inf Softw Technol 50(5):390–405

Kosar T, Mernik M, Črepinšek M, Henriques PR, da Cruz D, Varanda Pereira MJ, Oliveira N (2009) Influence of domain-specific notation to program understanding. In: Proceedings of the international multiconference on computer science and information technology, WAPL 2009—2nd workshop on advances in programming languages, pp 675–682

Kosar T, Mernik M, Črepinšek M, Henriques PR, da Cruz D, Varanda Pereira MJ, Oliveira N (2010) Comparing general-purpose and domain-specific languages: an empirical study. Comput Sci Inf Syst 7(2):247–264 (Extended version of CORTA'09 paper: comparison of XAML and C# forms using cognitive dimension framework)

Likert R (1932) A technique for the measurement of attitudes. Arch Psychol 22(140):1–55

MacVittie LA (2006) XAML in a nutshell. O'Reilly Media, Inc

Mauw S, Wiersma W, Willemse T (2004) Language-driven system design. Int J Softw Eng Knowl Eng 6(14):625–664

Meijer E, Beckman B, Bierman G (2006) Linq: reconciling object, relations and xml in the .net framework. In: Proceedings of the 2006 ACM SIGMOD international conference on management of data. ACM, New York, NY, USA, pp 706–706

Mernik M, Heering J, Sloane A (2005) When and how to develop domain-specific languages. ACM Comput Surv 37(4):316–344

Nugroho A (2009) Level of detail in UML models and its impact on model comprehension: a controlled experiment. Inf Softw Technol 51(12):1670–1685

Otero MC, Dolado JJ (2004) Evaluation of the comprehension of the dynamic modeling in UML. Inf Softw Technol 46(1):35–53

Peyton Jones S, Blackwell A, Burnett M (2003) A user-centred approach to functions in Excel. In: ICFP '03: proceedings of the eighth ACM SIGPLAN international conference on functional programming. ACM Press, New York, NY, USA, pp 165–176

Ricca F, Scanniello G, Torchiano M, Reggio G, Astesiano E (2010) On the effectiveness of screen mockups in requirements engineering: results from an internal replication. In: Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement (ESEM 2010). ACM, New York, NY, USA, pp 17:1–17:10

Sjoeberg D, Hannay J, Hansen O, Kampenes V, Karahasanovic A, Liborg NK, Rekdal A (2005) A survey of controlled experiments in software engineering. IEEE Trans Softw Eng 31(9): 733–753

Sprinkle J, Mernik M, Tolvanen JP, Spinellis D (2009) What kinds of nails need a domain-specific hammer? IEEE Softw 26(4):15–18

Storey MA (2005) Theories, methods and tools in program comprehension: past, present and future. In: IWPC '05: proceedings of the 13th international workshop on program comprehension. IEEE Computer Society, pp 181–191

Thibault S, Marlet R, Consel C (1999) Domain-specific languages: from design to implementation—application to video device drivers generation. IEEE Trans Softw Eng 25(3):363–377

Varanda Pereira MJ, Mernik M, da Cruz D, Henriques PR (2008) Program comprehension for domain-specific languages. Comput Sci Inf Syst 5(2):1–17

Webb Collins R, Hevner AR, Walton GH, Linger RC (2008) The impacts of function extraction technology on program comprehension: a controlled experiment. Inf Softw Technol 50(11): 1165–1179

Wilcoxon F (1945) Individual comparisons by ranking methods. Biom Bull 1(6):80–83

Wile DS (2001) Supporting the DSL spectrum. J Comput Inf Technol 9(4):263–287

Tomaž Kosar received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research is mainly concerned with design and implementation of domain-specific languages. Other research interest in computer science include also domain-specific modelling languages, empirical software engineering, software security, generative programming, compiler construction, object oriented programming, object-oriented design, refactoring, and unit testing. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also a visiting professor at the University of Alabama at Birmingham, Department of Computer and Information Sciences, and at the University of Novi Sad, Faculty of Technical Sciences. His research interests include programming languages, compilers, domain-specific (modelling) languages, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Jeffrey C. Carver received the PhD degree in Computer Science from the University of Maryland. He is an assistant professor in the Department of Computer Science at the University of Alabama. His main research interests include software engineering for computational science and engineering, empirical software engineering, software quality, software architecture, human factors in software engineering and software process improvement. He is a senior member of the IEEE Computer Society and the ACM. Contact him at carver@cs.ua.edu.