Domain-Specific Languages in Practice: A User Study on the Success Factors

Felienne Hermans, Martin Pinzger, and Arie van Deursen

Delft University of Technology
{f.f.j.hermans,arie.vandeursen,m.pinzger}@tudelft.nl

Abstract. In this paper we present an empirical study on the use of a domain-specific language(DSL) in industry. This DSL encapsulates the details of services that communicate using Windows Communication Foundation (WCF). From definitions of the data contracts between clients and servers, WCF/C# code for service plumbing is generated. We conducted a survey amongst developers that use this DSL while developing applications for customers. The DSL has been used in about 30 projects all around the world.
We describe the known success factors of the use of DSLs, such as improved maintainability and ease of re-use, and assert how well this DSL scores on all of them. The analysis of the results of this case study also shows which conditions should be fulfilled in order to increase the chances of success in using a DSL in a real life case.

1 Introduction

Domain-specific languages(DSLs) are languages tailored to a specific application domain [12]. DSLs have been described in literature for several decades. They often appear under different guises, such as special purpose [21], application-oriented [15], specialized [4] or task-specific [13] programming languages. An overview of widespread DSLs can be found in [12]. Most authors agree that the use of domain-specific languages has significant benefits, amongst which reduced time-to-market [8] and increased maintainability [2, 19].

However, very little research has been done to the use of DSLs in industry. Are DSLs really as helpful as we think when used within large companies? And if they are, what makes them?

In order to answer this, empirical studies of actual DSL usage are required. In this paper, we report on such a study. It involves the DSL called ACA.NET that is used to create web services that communicate using Windows Communication Foundation (WCF). ACA.NET has been used in over 30 projects all around the world.

In this paper we investigate factors that contribute to the success of this DSL. In particular, we conducted a study among 18 users of ACA.NET, by means of a systematic survey, investigating issues such as usability, reliability, and learnability. With the results of this study, we seek to answer the following research question:

– What are the main factors contributing to the success of a DSLf?

The remainder of this paper is structured as follows. In Section 2 we summarize related work with focus on papers describing known success factors of DSLs. Section 3 introduces ACA.NET, the studied DSL, to the reader. Section 4 presents DSL success factors, the questionnaire and the experimental set-up. The results of the survey can be found in Section 5. In Section 6 the research question is answered, both for ACA.NET as well as for domain-specific languages in general. A summary of our contributions and an outlook towards future work can be found in Section 7.

2 Related work

Several papers discuss advantages and disadvantages of the use of DSLs. For instance, van Deursen and Klint [19] observe that DSLs can substantially ease the *maintenance* , however they also indicate that the cost of extending a DSL for unanticipated changes can be substantial. Kieburtz *et al.* describe that DSLs can increase *flexibility*, *productivity* and *reliability* [9]. *Reusability* is also mentioned as an advantage of the use of DSLs, for instance by Ladd and Ramming [6] and Krueger [10]. The latter furthermore points out that a DSL can reduce the effort to create a system from a specification. From Bell [3] and Spinellis and Guruprasad [17] we learn that DSLs can *ease design and implementation* of a system, by reducing the distance between problem and program.

Spinellis [16] describes *reliability* as an advantage; because of the small domain and limited possibilities of a DSL, correctness of generators or interpreters can be easily verified. However, he also discusses disadvantages, such as training costs for users of the DSL and the lack of knowledge of how to fit the use of a DSL into standard software development processes [16]. Finally, Mernik *et al* [12] mention that a DSL can also be used as a *domain-specific notation*. This way, existing jargon can be formalized.

Most of these papers primarily provide anecdotal evidence for the benefits claimed, often based on a handful of usage scenarios for the language in question. While this provides useful information, more confidence can be gained from rigorous empirical studies. Unfortunately, we only found a few of such studies in the literature. Batory *et al* [2] describe a case study where a DSL is used for simulations. They report improved *extensibility* and *maintainability*. Kieburtz *et al.* [9] describe a series of experiments comparing code generation using a DSL to code generation via templates. Herndon and Benzins [8] report on improvements, amongst which *reduced time-to-market* and *improved maintainability* due to the use of DSLs. Unfortunately they lack to report how they come to their observations. Furthermore, their *Kodiyak* language has been used in only four cases. Both Weiss [20] and Bhanot *et al.* [5] report on a productivity increase of 500%, but is is not made explicit how these numbers were obtained.

Empirical work in the area of model-driven engineering in general is somewhat more common. For example, Baker *et al.* [1] describe a large case study, in which source code and test cases were generated from models. They present numbers on *increased productivity*, *quality* and *maintainability*. White *et al* [22] also describe a case study in which code is generated. Their paper reports on reduced effort on development and improved quality, but they only describe the results of one case. We have found one account where a questionnaire was used to study the ideal situations for model-driven development [18]. This questionnaire, however, addressed model-driven engineering in

general, rather than the specific merits of the domain-specific notation used in a software project. To the best of our knowledge, no user study like ours has been performed before.

3 About ACA.NET

ACA.NET,[1] Avanade Connected Architectures for .NET, is a visual DSL developed by Avanade.[2] It is used to build web services that communicate via Windows Communication Foundation.[3] Developers from Avanade noticed that for many projects in which a service oriented application had to be created, the same simple, but time consuming tasks had to be repeated for each project. Typical tasks include creating classes for service contracts, data contracts, writing service configuration, writing endpoint definitions and creating service clients. Because these tasks appeared very similar for each project, Avanade decided to create an abstraction for these tasks.

With ACA.NET a large part of the development of service oriented applications can be automated. ACA.NET enables the user to draw a model of a service oriented application on the Visual Studio-integrated design surface. This model consists of server and client objects and the data contracts between them. From this model, a large part of the C#-code is generated. Only the business logic that describes the behavior of the service has to be implemented by hand, which can be done through C# partial classes.

ACA.NET is built with Microsoft DSL Tools [7]. The code generation is implemented using Microsoft's Text Template Transformation Toolkit (T4) that is part of the DSL Tools suite.

4 Experimental Design

To measure the success of ACA.NET we conducted a survey amongst ACA.NET developers. The survey was set up according to the guidelines of Pfleeger and Kitchenham [14]. Their guidelines propose to start by setting the survey objective. The objective of our study is to provide answers to the following ACA.NET specific research question:

– What are the main factors contributing to the success of ACA.NET?

4.1 DSL Success Factors

To reason about the success of ACA.NET, we identified a number of success factors of DSLs. We obtained these factors from the related work in the field which has been presented in the Section 2. We aimed at making this list of factors specific to the use of DSLs. Thus general success factors, such as commitment from higher management or the availability of skilled staff were not taken into consideration, as they are not directly affected by the use of a DSL. The resulting factors under consideration are:

[1] See `http://www.avanade.com/delivery/acanet/`

[2] Avanade is a joint venture between Accenture and Microsoft. See `www.avanade.com`

[3] See `http://en.wikipedia.org/wiki/Windows_Communication_Foundation`

Reliability (I) [16, 9] In addition to reducing development costs, automation of large parts of the development process leads to fewer errors.

Usability (U) [3, 17] Tools and methods supporting the DSL should be easy and convenient to use.

Development costs (C) [8] The DSL helps developers to model domain concepts that otherwise are time-consuming to implement. The corresponding source code is generated automatically. This lowers development costs and shortens time-to-market.

Learnability (L) [16] Developers have to learn an extra language, which takes time and effort. Furthermore, as the domain changes the DSL has to evolve and developers need to stay up-to-date.

Expressiveness (E) [12] Using a DSL, domain specific features can be implemented compactly, however, the language is specific to that domain and limits the possible scenarios that can be expressed.

Reusability (R) [6, 10] With a DSL, reuse is possible at model level, making it easier to reuse partial or even entire solutions, rather than pieces of source code.

With the reliability we mention we do not mean the number of bugs per line of code or other objective measures. We did not use that kind of measures for a few distinct reasons. Firstly, these measurements were not available for all of the projects in which ACA.NET was used. Secondly, since large parts of the code are generated, the amount of lines of code is not comparable to projects without ACA.NET. We believe the *perceived* reliability we use is a good measure, because developers often have a good feeling for improved quality of the software. They know whether the number of bugs is reasonable with respect to both the complexity and the size of the project. The same goes for the reduced costs. Since we did not have project data like lines of code or hours spent on it, we asked the developers to estimate it. We believe developers have a good idea on how much time they spent on their projects.

4.2 Questionnaire to measure DSL success factors

Every question in the questionnaire relates to one or more of these factors of a DSL, because to cite [14] *it's essential that the survey questions relate directly to the survey objectives.* In the following we review the success factors and describe the questions that we use to measure them. Every success factor is covered by at least one Likert question, so it is possible to measure it. Open questions are added to the questionnaire to obtain more insight into the results. Table 4.2 provides an overview of the questionnaire. A pdf version of the questionnaire can be downloaded from `http://www.st.ewi.tudelft.nl/~hermans/`.

The questionnaire basically consists of three parts. The first part, questions Q1 and Q2, concerns the background of the subject. The second part, questions Q4–Q10, contains the questions related to *one* specific ACA.NET project. For all subjects we investigated the set of projects for which they were listed as contact person. The third part of the survey, questions Q11–Q20, comprises questions on ACA.NET in general. In this part, we limited the answer-space to two five-point Likert scales to facilitate the measurement of the various success factors. The first one ranges from strongly disagree,

ID	Question	Factor
Background Questions		
Q1	How many years have you worked as a professional software developer?	L
Q2	How much experience do you have with ACA.NET	L
Project specific questions		
Q3	Was this a new ACA.NET project or built on an existing version?	R
Q4	If you start a new ACA.NET project, how do you proceed?	R
Q5	Did the ACA.NET user interface help you modeling?	U
Q6	Did you use other tools for modeling in this project, next to the ACA.NET interface?	U
Q7	Can you estimate the percentage of time that would be spent on the following tasks if ACA.NET was not used for this project?	C
Q8	Can you estimate the percentage of time that you actually spent on the following tasks?	C
Q9	Estimate the percentage of code that was generated	C
Q10	How many lines of code did this project consist of?	C
General ACA.NET questions		
Q11	How many days did it take you to get to know ACA.NET?	L
Q12	How many hours a month does it take you to stay up to date on ACA.NET?	L
Q13	Did you ever consider to use ACA.NET but decided against?	U
Q14	In case you answered Yes to the previous question, please indicate why.	U,E
Q15	Indicate your agreement with	
Q15a	The code is more readable	I
Q15b	Fewer errors occur	I
Q15c	The product complies better with the customers requirements	I
Q16a	ACA.NET makes designing easier	U
Q16b	ACA.NET makes implementing easier	U
Q16c	ACA.NET is powerful	U,E
Q17	Did you ever deny a customer a feature because you knew you would not be able to implement it using ACA.NET?	E
Q18	Did you ever have to write extra code (other than custom code for business logic) to implement features?	E
Q19	Indicate your agreement with	
Q19a	ACA.NET is difficult to use	U
Q19b	ACA.NET restricts my freedom as programmer	E
Q19c	ACA.NET doesn't have all features I need	E
Q20a	I look into the generated code in order to be able to understand the underlying models	E
Q20b	I look into the generated code in order to be able to be able to write custom code	E

Table 1. Overview of the questionnaire used for the ACA.NET survey.

disagree, neutral, agree, to strongly agree. The second Likert scale ranges from very often, often, sometimes, seldom, to never.

Reliability of ACA.NET solutions (I) Because parts of the development process are automated, software constructed using a DSL is expected to be less error prone. To measure the reliability of ACA.NET we ask the subjects whether they think that the use of ACA.NET increases the quality of the delivered code in the following ways: the code is more readable (Q15a), fewer bugs occur (Q15b), and the product complies better with the customer requirements (Q15c). The possible answers to each question are defined by a five point Likert scale. We expect that DSLs help to communicate requirements better to the customer and developers. Furthermore, ACA.NET code is assumed to be more readable and easier to understand. Both aspects are expected to lead to fewer bugs in ACA.NET web services. We want to stress again that we measure *perceived* reliability, meaning we measure how users feel about the increased reliability.

Usability of ACA.NET (U) Another factor is ease of using the DSL and the tools that support it. We included several questions dedicated to the usability of the ACA.NET toolkit for developing web services. For instance, does the ACA.NET user interface help in modeling web-services (Q5) and were other tools used in the project (Q6). We asked whether subjects decided against the use of ACA.NET (Q13) in any project, and, if yes, reasons why they did so (Q14). Descriptions of reasons could be provided in free-text. We also added questions to assess whether ACA.NET eases designing (Q16a) and implementing web services (Q16b), and summarizing that ACA.NET is a powerful DSL (Q16c). Question Q19a is used to obtain the level of agreement on the statement that ACA.NET is difficult to use.

Reduction of development costs (C) With the use of ACA.NET, developers can focus on the business logic while other web-service related source code is generated by ACA.NET. Therefore time spent on tasks related to web-services is assumed to be shorter and development costs are assumed to be lower. For measuring the effect of ACA.NET on development costs we formed a set of questions related to the experiences with the selected project. For instance, we ask each subject to estimate the percentage of time that would have been spent on the following tasks if ACA.NET was *not* used: write data contracts, write service configuration, and write business logic (Q7). Next, we ask the subjects to estimate the percentage of time they spent on actually: design contracts, generate the source code, and write the business logic with ACA.NET for the selected project (question Q8). In addition, we ask the subjects to estimate the percentage of source code that has been generated with ACA.NET (Q9).

Learnability of ACA.NET (L) The time invested in actually learning and staying up-to-date represents our first success factor for DSLs. For measuring the learnability of ACA.NET we first ask the subjects for their level of experience in terms of years worked as professional software developer (Q1) and in terms of years worked with ACA.NET (Q2). Later on in the questionnaire we ask for the detailed effort numbers.

In particular, we were interested in the number of days of 8 working hours invested in learning ACA.NET (Q11) and the number of hours invested in staying up-to-date on ACA.NET (Q12).

Expressiveness of ACA.NET DSL (E) To measure the expressiveness of ACA.NET we asked the subjects how often they had to deny a customer a feature, because it could not have been implemented with ACA.NET (Q17) and how often they had to write extra code to implement a feature (Q18). Answers to both questions are given by a five-point Likert scale. Furthermore we investigated whether respondents feel that ACA.NET restricts their freedom (Q19b) and whether ACA.NET provides all the features needed to develop web services (Q19c). The answers to the latter two questions are also given with a five-point Likert scale.

To obtain a deeper insight into the expressiveness of ACA.NET we added questions Q20a and Q20b. We ask whether developers look into the source code to understand the models defined with ACA.NET (Q20a). Question Q20b assesses whether developers use the generated source code instead of the models to add custom code. Frequent use of the generated code indicates the model does not express all properties of the domain.

Reusability of ACA.NET models (R) As with traditional software engineering, one goal of a DSL is to reuse existing solutions. We addressed the reusability of ACA.NET models in question Q3. We ask the subjects whether they reuse models of existing projects. For instance, when they start a new project do they start from existing assets or from scratch.

4.3 Survey set-up

We conducted our survey online, in a Sharepoint environment, making it cost-effective and also appropriate, because our target group is used to this kind of surveys. We choose a self-control study [14], comparing user experience with and without the use of ACA.NET. The fact that the subjects are not able to see each others results makes the survey more resilient to bias. The fact that the survey is cost-effective, appropriate and resilient to bias, makes it *efficient* according to Pfleeger and Kitchenham [14]. Furthermore, automation reduces the contact between subjects and researchers, giving the researchers less opportunity to bias responders.

In total we invited 48 people to participate in this survey. Of 21 subjects we knew for sure they used ACA.NET and of 27 people we thought they might have experience with it. 28 people responded, of which 10 indicated they did not use ACA.NET, or their experience was too limited to answer the questions. We got 18 meaningful results, giving our survey an effective response rate of 38%. Since our target population is small, we did not use any form of sampling.

With the invitation for the survey, developers received an email explaining them the purpose of the survey; helping to improve the tool set they work with everyday. We expect this to be a good motivation for them to participate, especially since there has been no opportunity to give official feedback, other than bug reports on ACA.NET.

By testing the survey, we estimated the time needed to fill out the questionnaire at about 60 minutes, which is appropriate for a self-administered survey on a subject important to responders. As recommended by Pfleeger and Kitchenham [14], we added a neutral option to all Likert-scaled [11] questions.

We believe there is little risk of researcher bias, because the researchers are not part of the users or designers of ACA.NET. When creating this survey, we ensured that subjects get the possibility to reflect on both, the positive and the negative aspects of ACA.NET.

5 Results

In this section we present the results of the survey, grouped by success factor.

5.1 Reliability (I)

Developers clearly believe that the use of ACA.NET increases the quality of the delivered code, since 40% of the respondents agree and 50% strongly agree with Question 15b as shown in Figure 1. As one of the respondents put it: *"The application becomes less error prone since lots of tasks are automated"*. Note that only one respondent disagrees with this statement.

5.2 Usability (U)

Over 75% of the developers indicate that ACA.NET aids them in modeling by giving them a good overview of the whole connected system of servers and clients (Figure 2). The reasons indicated by the respondents include that *"using ACA.NET gives us a better overview at higher abstraction"*, and that *"the DSL design surface helps to model the services even before business logic has been designed"*. Furthermore, the ACA.NET tools were considered easy to use (*"ACA.NET provides an easy to use interface that can be taught to others very quickly."*). Note that none of the respondents agrees to the statement that ACA.NET is difficult to use (Question 19a) as shown in Figure 3.

5.3 Development costs (C)

Based on the results of the survey, we can conclude that the use of ACA.NET indeed reduces programming time. One of the respondents says: *"ACA.NET speeds up the implementation of trivial tasks"*. From the answers to Question 7 and Question 8 we can conclude that time spent on actually coding the services is reduced from 46% to only 18%, as shown in Figure 6. The shift in focus to the more important business logic is also underlined by a subject who responded: *"We don't think too much about Windows Communication Foundation services or the Data Access Layer anymore as we are able to concentrate on the business requirements."*

Time is not the only measure for reduced costs: we also take the amount of generated code into account. The respondents estimate that on average 40% of the code is generated, distributed as shown in Figure 4.

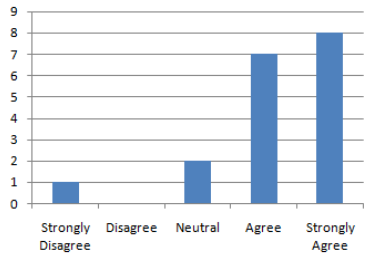

Fig. 1. Question 15b. Agreement with the statement that "fewer errors occur"

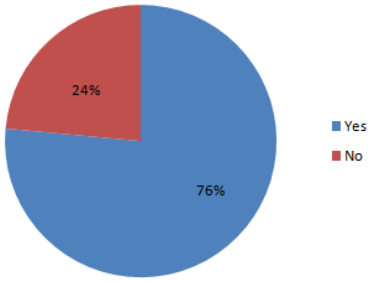

Fig. 2. Question 5. Did the ACA.NET user interface help you in modeling?

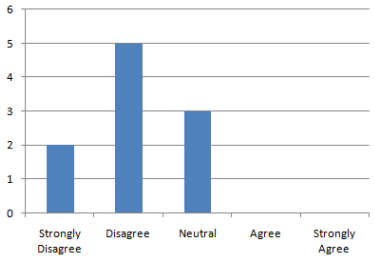

Fig. 3. Question 19a. ACA.NET is difficult to use.

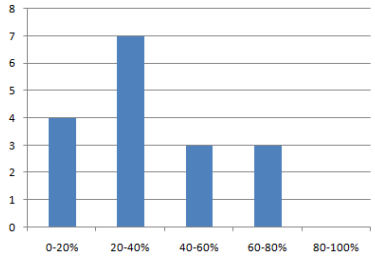

Fig. 4. Question 9. Estimate the percentage of code that was generated

According to the answers on Question 16a and Question 16b, developers also feel that ACA.NET eases the design and implementation phases (Figure 5), which is likely to result in less time (and lower costs) for these tasks.

5.4 Learnability (L)

The respondents indicate that it took them quite some time to learn the basics of ACA.NET, as shown in Figure 7. Most respondents were able to learn ACA.NET within one week, while the maximum time mentioned was 15 days. Apart from learning ACA.NET, it also takes time to stay up to date, as shown in Figure 8.

5.5 Expressiveness (E)

The developers turn out to be satisfied with the expressive power of ACA.NET: 60% of them agrees that ACA.NET is powerful (Figure 9). Furthermore, we see that the limited scope is not considered a problem; only few developers indicate their freedom is restricted (Figure 10). There are however some developers that indicate they miss features (Question 19c, Figure 10).

The model is a good representation of the code, since developers do not have to look into the code to understand or complete their own code (see Figure 11). However,

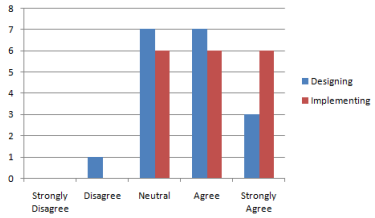

Fig. 5. Question 16. ACA.NET makes designing and implementing easier

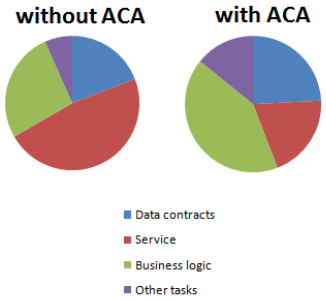

Fig. 6. Question 7 and 8. Please estimate the percentage of time you spent on typical development tasks

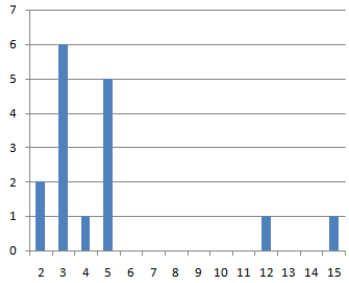

Fig. 7. Question 11. How many days did it take you to get to know ACA.NET?

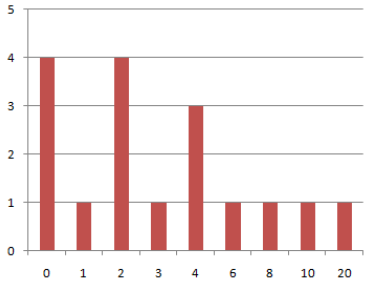

Fig. 8. Questions 12. How many hours a month does it take you to stay up to date on ACA.NET?

respondents mention that it is very hard to evolve the models along with the code, which indicates lack of expressiveness. *"When the models get more complicated, such as for the web factory where you can set a lot of properties, the model loses its value - its not practical to maintain or set a lot of properties using the visual tool."* and *"For the more complex, it was to time-consuming to maintain the graphical details between updates, and you lost the overview."*

5.6 Reusability (R)

A somewhat surprising result is that *reuse* hardly plays a role in ACA.NET. The answers to Question 3, Figure 12, tell us ACA.NET models are never reused. Even conceptual designs are hardly ever reused, in only 10% of the cases. One possible explanation is that the current ACA.NET implementation does not directly support exporting or importing models. In particular, respondents indicated that they would like to be able to import parts of earlier models, to reuse standard architectures for services across projects, and to compose services from multiple earlier defined models. As one respon-

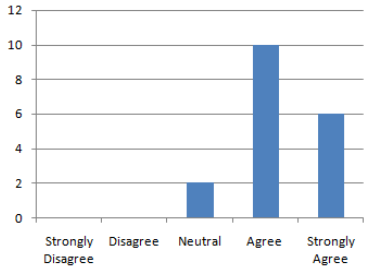

Fig. 9. Question 16c. Is ACA.NET powerful?

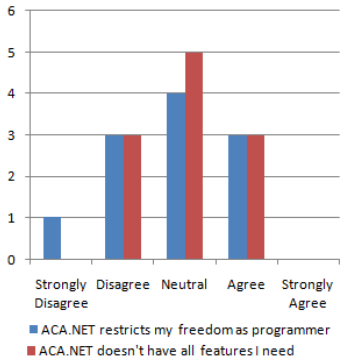

Fig. 10. Question 19. Is ACA.NET restrictive / feature-incomplete?

der said: *"ACA.NET could be improved by providing import mechanisms which allow the importation of other ACA-files into the model."*

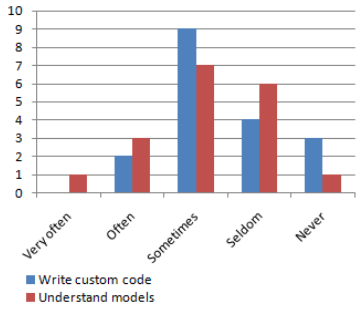

Fig. 11. Question 20. Inspection of generated code for different purposes

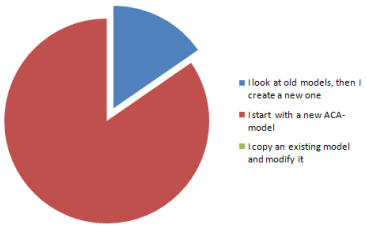

Fig. 12. Question 3. If you start with a new ACA-project, how do you proceed?

6 Discussion

6.1 Lessons Learned Concerning ACA.NET

Based on our study we can draw several lessons concerning ACA.NET. First, the developers indicate ACA.NET helped in reducing development costs by reducing time spent on programming services. Because project managers at Avanade indicate that programming a web service accounts for a large part of the development process, we conclude that ACA.NET also lowers time-to-market. The results also show that ACA.NET helped

in increasing reliability, the vast majority of developers agree that the use of ACA.NET helps to avoid faults.

Underlying success factors were usability of the language and tool set (which was rated as positive), the learnability of the language (several days initial learning, and several hours per month to stay up to date), and the expressiveness of the language which was focused specifically towards the web services domain (and which was rated as powerful). Much to our surprise, reuse of models did not contribute to the success of ACA.NET.

Conducting the study also resulted in several suggestions for improving the ACA.NET language and tool set. A first observation is that adding the possibility to import or export partial models would make it possible to actually reuse (parts of) models, adding even more benefits to the use of ACA.NET.

Second, some of the developers observed that the abstraction that is made in ACA.NET for web services is not specific to .NET/WCF services: *"ACA.NET is very good for modeling the service layer of an enterprise application structure, but only code for the .NET based services can be generated. Unfortunately it is not possible to generate code for Java or SAP platform based services."* By appropriately extending the code generators, ACA.NET could be used to integrate services from different platforms.

Furthermore, several respondents indicated that ACA.NET was not used as often as possible, because customers do not want to make use of or be dependent on proprietary software. To cite one of the respondents: *"Customers don't use ACA.NET in quite some cases since it's an Avanade specific tool."* A way out of this could be to give customers access to the source code of the underlying code generation infrastructure.

Last but not least, several of the developers would have liked access themselves to the generator and underlying meta-models, as this would enable them to build in customer-specific features in an easier way. This actually calls for proper extension points and hooks in the language, and suggests that the level of expressiveness of the language could be further improved.

6.2 Beyond the Case at Hand

An interesting question is which general lessons we can learn from the case at hand.

First of all, the case provides further evidence that the use of a domain-specific language can reduce time-to-market and development costs, and can improve system reliability. The evidence, in this case, not just comes from the creators of the language, but from the people who are actually using the language to deliver working software to their customers.

Second, the case suggests that reuse is not a critical success factor. Reuse is a notoriously hard problem, involving the identification, adaptation, and integration of existing parts. For many application domains, light-weight, copy-paste based forms of reuse may be enough, having the additional benefit of full flexibility.

Another lesson we can draw from the study is that the questionnaire itself is a useful instrument for, e.g., identifying opportunities for improving the language. In fact, we would recommend engineers involved in the design of a new domain-specific language to compose a questionnaire as part of their design effort. This questionnaire, for which

ours can form a starting point, can then be used in a later stage to evaluate whether the language has met its design goals.

6.3 Threats to Validity

Content Validity One of the threats to *content validity* when conducting an (online) survey is the fact that respondents could be influenced by other replies [14]. Therefore we made sure that it was not possible for respondents to view each others results. Furthermore, responses came from different divisions of Avanade, making it less likely that responders spoke to each other about the survey.

Another threat to content validity is the fact that respondents have to estimate the percentage of generated code and time spent on different tasks. This is the case because corresponding data were not collected during the development process. Developers could be imprecise in their memory. Because all results show similar numbers, we believe the results are sufficiently reliable.

The survey was pre-tested on a focus group, consisting of domain experts — the developers that created ACA.NET — and members of the target population. The survey questions were also reviewed by university staff with experience in empirical research. Their feedback helped in further assuring content validity.

Internal Validity The calculations used to manipulate the data were all very simple, and constitute no threat to internal validity.

An issue of concern could be that the respondents have a commercial interest in putting up a bright picture, thus giving answers that are too positive. While we cannot exclude this possibility, we do not believe this is the case. We explicitly announced the questionnaire as an opportunity to suggest improvements for ACA.NET, encouraging them to be as critical as possible.

Survey Reliability In order to ensure repeatability of the experiment, the full questionnaire including answer options and descriptions is available online.[4] Unfortunately we were not able to make Avanade's answers available too, for reasons of confidentiality.

External Validity Some of the issues concerning external validity were discussed in Section 6.2, where we addressed the implications of our study beyond ACA.NET. Furthermore, we have no reason to believe that our results are specific to the *web services* domain. One characteristic of this domain, however, is that it is a "horizontal" domain, applicable in many different settings, and aimed at developers as language users. This has clearly had some influence on our questionnaire, which is tailored towards developers.

Another issue may be that the results were obtained in a *commercial* setting: we have no reason to believe that they would be different for, e.g., open source projects.

[4] See `http://www.st.ewi.tudelft.nl/~hermans/`

7 Conclusions

The goal of the present paper is to obtain a deeper understanding of the factors affecting the success of a domain-specific language in practice. To that end, we have analyzed experiences of developers that made use of the ACA.NET DSL in over 30 projects around the world.

The key contributions of this paper are as follows:

- The identification of a number of DSL success factors;
- A questionnaire that can be used to assess these factors in concrete DSL projects.
- The ACA.NET empirical study, in which we use the proposed questionnaire to evaluate success factors in the use of ACA.NET.

The outcomes of the study indicate that in the given case study the DSL helped to improve reliability, and to reduce costs. Furthermore, conducting the survey resulted in a number of suggestions for improving the DSL under study, such as increasing the level of reuse.

We see several areas for future work. One direction is to conduct a similar survey in a DSL from a less technical (horizontal) domain, but from a vertical, highly specialized DSL. The challenge here will be to find such a DSL in industry and the corresponding industrial partner that is willing to collaborate in such a survey. A second direction is to compare the results we obtained from interviewing with "hard" data obtained from, e.g., measurements on code or the software repository used. One of the challenges here will be the availability of accurate data on, e.g., reliability of projects conducted with the DSL under study.

Acknowledgements We owe our gratitude to all responders that took the time to fill out our survey. Special thanks go out to Gerben van Loon and Steffen Vorein, for reviewing the questionnaire itself extensively.

References

1. P. Baker, S. Loh, and F. Weil. Model-driven engineering in a large industrial context – motorola case study. In *Proceedings 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *Lect. Notes in Comp. Sc.*, pages 476–491. Springer-Verlag, 2005.
2. D. Batory, C. Johnson, B. MacDonald, and D. von Heede. Achieving extensibility through product-lines and domain-specific languages: A case study. In *Software Reuse: Advances in Software Reusability; Proceedings 6th Int. Conf. on Sw. Reuse (ICSR)*, volume 1844 of *Lect. Notes in Comp. Sc.*, pages 117–136. Springer-Verlag, 2000.
3. J. Bell, F. Bellegarde, J. Hook, and R.B. Kieburts. Software design for reliability and reuse: a proof-of-concept demonstration. In *Proceedings Conference on TRI-Ada*, pages 396–404. ACM Press, 1994.
4. T. J. Bergin, Jr. and R. G. Gibson, Jr., editors. *History of programming languages—II*. ACM, New York, NY, USA, 1996.

5. V. Bhanot, D. Paniscotti, A. Roman, and B. Trask. Using domain-specific modeling to develop software defined radio components and applications. In *Proceedings of the 5th OOPSLA Workshop on Domain-Specific Modeling (DSM05)*. Computer Science and Information System Reports, Technical Reports, 2005.
6. D. L. Christopher and J.C. Ramming. Two application languages in software production. In *USENIX Symposium on Very High Level Languages Proceedings*, pages 169–187. USENIX, 1994.
7. S. Cook, G. Jones, S. Kent, and A. Cameron Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Microsoft .NET Development Series. Addison-Wesley, 2007.
8. R.M. Herndon and V.A. Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14:803–809, 1988.
9. R.B. Kieburtz, L. McKinney, J.M. Bell, J. Hook, A. Kotov, J. Lewis, D.P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *International Conference on Software Engineering (ICSE'96)*, pages 542–552. IEEE Computer Society, 1996.
10. C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.
11. R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140), 1932.
12. M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.
13. B. A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT Press, 1993.
14. S. Pfleeger and B. Kitchenham. Principles of survey research. *ACM SIGSOFT Software Engineering Notes*, 26:16–18, 2001.
15. J. E. Sammet. Programming languages: history and future. *Communications of the ACM*, 15(7):601–610, 1972.
16. D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56:91–99, 2001.
17. D. Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. In *Proceedings of the Conference on Domain-Specific Languages (DSL'97)*, pages 67–76. USENIX, 1997.
18. M. Staron. Adopting model driven software development in industry: A case study at two companies. In *Proceedings 9th Int. Conf. on Model-Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 of *Lect. Notes. in Comp. Sc.*, pages 57–72. Springer-Verlag, 2006.
19. A. van Deursen and P. Klint. Little languages: little maintenance. *Journal of Software Maintenance*, 10(2):75–92, 1998.
20. D. Weiss. Creating domain-specific languages: the fast process. In *First ACM-SIGPLAN Workshop on Domain-specific Languages: DSL97*. University of Illinois, Technical Reports, 1997.
21. R. L. Wexelblat, editor. *History of programming languages I*. ACM, New York, NY, USA, 1981.
22. J. White, D. C. Schmidt, and A. Gokhale. Simplifying autonomic enterprise java bean applications via model-driven development: A case study. In *Proceedings 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *Lect. Notes in Comp. Sc.*, pages 601–615. Springer-Verlag, 2005.