A case study on reusability of a DSL in a dynamic domain

Kalle Korhonen
Information Technology Research Institute
University of Jyväskylä, PL 35, FIN-40351 Jyväskylä, Finland
kaosko@cc.jyu.fi

Abstract

Researchers of domain-specific languages claim that a domain-specific language (DSL), particularly a visual one, can highly increase productivity when the domain is stable and architecture is matured. However, in the real world the domain rarely remains completely unaltered and the software architecture usually goes through series of changes over time. Thus, the promises of DSLs do not always get realized in practice. The costs to change a complete DSL can be high because it contains many different layers and elements that need to work together. This study tries to identify reusable elements of a visual DSL in order to lower these costs and to facilitate construction of a DSL.

Introduction

Frequently changing and loosely defined domain, unstable architecture and ever-changing component framework are common problems in practical domain-specific development. They cause constant pressures for changing the DSL, which can prove out to be challenging [1]. In this paper we study the reusability of a visual DSL and ways to increase its maintainability. Through a case study, we examine possibilities to develop a new DSL using constructs of an experimental visual DSL that was previously built. The domains in these DSLs are very close to each other, but not exactly the same. Furthermore, the domain-specific frameworks differ which limits the reusability and makes it important to identify reusable parts and study approaches to enhance modularity of a DSL.

In [2] the author presented an empirical study on development of an experimental visual DSL. The domain chosen for that study was a programming game called Mikrobotti [3]. The participants of the game create virtual robots to fight against other participants' robots according to the programmed logic in them. The robots are built using the Java language by adding behavior in various hotspots the Java-based robot framework provides and by filling values of several objects and their properties. The author extended the framework and on top of it he created a visual DSL using metaCASE tool MetaEdit+ [4]. The study provided empirical data on the development of a DSL. The time required for implementing different

parts of the DSL was reported. The study also shows the separate tasks, results and skill level required to complete each development phase of a typical DSL. Summary of the development is presented in Table 1.

Table 1: Summary of DSL development phases and tasks [2]

Phase	**Tasks**	**Time used**	**Classes produced**	**Notes**
Rewriting product-line architecture supporting program code	Designing and creating reusable classes	5 weeks	8, many of the original 13 classes at least partially rewritten	Tedious and laborious
Development of the metamodel	Development of metamodel elements (types)	2 days	0	Easy! Drawing symbols took half of the time
Development of the code generator	Tests, writing report templates and fitting Java framework to work with the code generator	2 days for tests, 3 days for the code generator, 2 days for adaptation, total 1 ½ week	5, superclasses for the generator produced classes	Code generator produces 5 classes for every robot
Finishing	General fixes and adjustements	1 week	2 interfaces	
Total		*Ab. 8 weeks*	*28 in the extended framework totally*	

In 2001, Matthew Nelson working for IBM Alphaworks created Robocode [5], a programming game that is strikingly similar to Mikrobotti. That provided a unique opportunity to experiment DSL development in two domains that closely resemble each other. In this study we do not actually completely create a new DSL in practice, we only evaluate the development of it on a theoretical level. However, we examined Robocode framework carefully enough that we are convinced that the time distribution among the development tasks would resemble the presented results of Mikrobotti development. A similar case of two similar domains in practice would be Nokia and Ericsson that both work in the mobile phone software domain, but their software architecture and ways of development differ. It is also analogous to a real world situation when the domain or underlying software architecture rapidly changes because of some external pressure. For example, Nokia is facing similar problems when adapting their architecture from proprietary C++ operating system to standardized Java operating system for hand-held devices. Therefore, studying the effects of domain changes and ways to control them is significant to the evolution and wider adoption of DSLs.

Comparing the two domains, they are surprisingly similar at a first glance. However, Robocode is quite a lot simpler than Mikrobotti when examined closer. The game does not model real world physics so completely as does Mikrobotti and in Robocode you only create the robot behavior but in Mikrobotti you can also affect the robot structure and choose the parts it is built of. The goal and overall robot behavior are the same in both games, which would make it possible to create one unified DSL for both games. It seems that the same kind of tactics and moving logic would suit well in both games. However, mappings between the language and the underlying framework would be almost completely different, because the games are using their own component frameworks and provide their own public classes and interfaces for players to build their robots on. That does not prevent creating a unified framework layer on top for both games, but it would be challenging and impractical, since the game frameworks do not really add to each other. Robocode is just a simpler version of Mikrobotti, whereas many of the functionalities in the extended Mikrobotti framework, like generic path and aiming algorithms could be used in Robocode too. Thus, the extended framework developed for Mikrobotti cannot be reused as a whole and we could not gain much by modifying it to work with both games. However, it would make sense to identify the reusable parts, patterns and more generic parts of a DSL that could be reused in Robocode. The results might be applicable to DSL development in general.

Identifying reusable elements in the three layers of a DSL

In [2], it was empirically proven that developing the component framework took most of the time that was needed to develop a complete visual DSL. In addition to the component framework and its components there are two other elements in a DSL [6]: the actual language itself and the code generator that connects the language and the framework. A DSL can be viewed as a three-layer structure, as in Figure 1, where each upper layer is dependent on the lower layer. Changing the component framework might cause a change in the code generator and change in the domain most likely causes changes in all of the DSL layers, but changing the language layer does not require changes in any other layer.

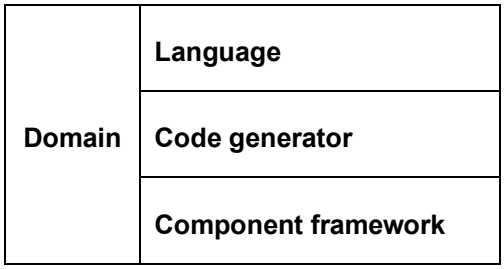

Figure 1: Layers of a DSL

We will examine the different layers of a DSL in order to find out reusable parts for a closely related domain. Generally, reusing the language level should be simple because it does not require changes in any other part of a DSL. In the case of Mikrobotti, it would be possible

reuse its models, symbols and other language elements in Robocode almost completely as they are. We would only have to discard the structure elements that are not needed. However, it must be noted that in practice it would not be as easy, because in the metaCASE tool, MetaEdit+, that we are using the reuse of the symbols between different visual DSLs is somewhat difficult, so we might have to re-draw the symbols. Even so, it would not be such a laborious task, because as can be seen in both Table 1 and Figure 1, the task to create the language itself is very small compared to other tasks of DSL development.

The code generator for Robocode would have to be re-created almost from scratch. Creating a code generator is difficult [7] and we already expressed concerns [2] that it might be difficult to maintain changes in the code generator for Mikrobotti. Especially in MetaEdit+ making the code generator would require extensive knowledge on both the language and the underlying framework, as the code generator is written with report language that MetaEdit+ provides. The code generator was divided in separate reports that were difficult to maintain. To some extent it is a trade-off whether you want to put the functionality in to the framework or the code generator. We came to the conclusion that the code generator should be kept as small as possible for maximum maintainability and flexibility. The code generator should deal with instance level issues as much as possible, i.e. to use ready-made functionalities the framework classes provide and only inherit and implement a small set of super classes for the code generator to function. That way the framework clearly defines the limits of the DSL and needs to modify the code generator can be minimized.

We already noted that the whole framework cannot be reused as is and we do not gain any benefits if we modify it to support both games. However, some parts of the framework could possibly be reused if it was modular. These reusable parts are likely to be applicable to other DSLs too. While the Mikrobotti framework and its extension are not currently built modularly, the different domain aspects could be handled in their separate modules, following the framelet pattern [8]. In Mikrobotti, we built a relatively generic state machine for both making state-based robot behavior possible and facilitating automatic generation of code. That part is clearly reusable and we could refine our state machine so that it would fit in different set-ups. Furthermore, according to [6] [9] many of the visual DSLs implement some kind of state machine, which also advocates developing a complete state machine framelet to be used in DSLs. Such independent, multi-purpose state machine implementations exist already, e.g. open source project Blissed [10], Jacaranda Framework [11] and others.

Of course, the behavior does not necessarily have to be implemented as a state machine. It could be e.g. some kind of directed graph, a self-organising map or any other logical model. In any case, it is important to base the DSL on some well-known, proven model to ease the development of a DSL and make its design clear and enduring. Furthermore, for commonly used models, there usually are open frameworks available that can be used as is or customized for use with a DSL.

Besides behavioral diagrams, there is another diagram type in Mikrobotti, a structure diagram, which has no correspondence at all in Robocode. The structure diagram defines the pieces the robot is built of. Based on that information the code generator creates one method that contains calls to assemble the parts with given properties to robot hull. The framework then calls this method when the robot is initialized. If we could use external text files in Mikrobotti (currently it is prevented), we would not have to generate this method, but we could just read the data from the file to avoid any code generation. More generically, we could say that the underlying framework should be well developed to minimize the need for the code generation. Generating a properties file read by the system is always easier than generating code: it cannot cause run-time failures and does not require extra time for compilation and deploying.

If we use an independent implementation of a state machine framework for Robocode, we only need to create the language itself and a code generator that implements the particular state machines modeled in Robocode diagrams to create a complete visual DSL. In Mikrobotti, the code generator was directly built into the reports of MetaEdit+, thus making it an integral part of the language and difficult to maintain. In that approach the code generator handles two tasks: interpreting the models using query language MetaEdit+ provides and the actual code generation. It might be better to separate the two tasks, so that we only generate an open computer-readable format of the model (MetaEdit+ also provides another kind of representations that we could use directly) inside the language and the tool. The code generator can then be written directly with the target language. The benefit is that reflective languages, such as Java, do not have to generate source code as an intermediate format but they can directly generate compiled code or, as in case of Java, intermediate bytecode as an output. Following these guidelines we have separated three parts of the code generator: a model interpreter, a code generator and an independent logical model, in our case a state machine framework. Modularizing the code generator this way should result in enhanced maintainability and increased reusability of the DSL and give it a better, more controlled and generalizeable structure.

Conclusion

In this study we have theoretically created a new visual DSL using previously built DSL for a similar domain. We have identified several reusable parts of a DSL and argued for the modular structure and the development principles of a DSL. We believe that at least part of these results can be generalized and used in DSL development in general.

A DSL consists of three parts: a language, a code generator and a component framework. We have previously shown empirically that creating the language is the smallest task of developing a DSL and argued here that if the domain changes we usually have to modify all parts of the DSL. We claim that a DSL should be based on some proven logical model, such as state machines. We might be able to reuse the symbols and other language elements but reuse possibilities are case-specific since the language elements are domain-specific and their

reusability depends on how the domain changes. Even so, the cost of creating a language is very small and reusability won't thus help significantly in this task.

We have made some interesting observations on the development of the code generator and its reusability. Most of all, we recommend that the code generator should be kept as small as possible to avoid maintenance problems. We argue that extending the framework is a better approach than adding the functionality into the code generator. We identified several reusable parts in the code generator depending on its structure: a model interpreter, an independent logic framework that corresponds to the logical model the language is based on and a code generator that uses the output of the model interpreter and generates instances of the framework defined by the logical model.

Examining the framework development affirms our conclusion that it is the most demanding and time consuming part in creating a functional DSL. We mentioned the possibility to build a framework modularly, constructed of several framelets. This observation is, of course no different from the known fact that increased modularity improves reusability. Depending on the domain and how it changes, we noted that there might be several parts of the framework that we could use as-is or with little modifications.

Analyzing the results, we can notice some research directions that could bring the DSL research forward and broaden its applicability. Generic logic frameworks that can be used in code generators are very interesting area that can quite quickly result in wider development and use of DSLs. Looking further, it would be interesting to see a metaCASE tool that is tuned in building DSLs. The kind of metaCASE tool could incorporate some code generator and model frameworks that are half-integrated with tool and generic language elements. In the coming years, it is also easy to predict that open formats, such as XML most likely gain more ground in the DSL field too, since more generic tools and frameworks need formats that can be easily transformed from one structure to another. There are a lot of tools to process and transform XML documents, which creates possibilities to create generic code generators for some particular logic model, e.g. a code generator that generate state machines. Yet another research direction, which might be a bit far fetching for the present, is the evolution of reflective languages. In theory a language that can change itself, might remove the need for the code generators as such completely. The research in these areas is under way.

References

1. Korhonen, K. *Motivation and Hypothesis for Comparison between Component Frameworks and DSL Paradigms*. in *OOPSLA Workshop on Domain-Specific Visual Languages (DSVL'01)*. 2001. Tampa, FL, USA.
2. Korhonen, K. *An empirical study on an experimental DSL for product-line development*. in *The 6th World Multiconference on Systemics, Cybernetics and Informatics*. 2002. Orlando, FL, US.
3. Mikrobotti, *Mikrobotti*. 2000. *http://www.mbnet.fi/mikrobotti*. Last accessed 15.8.2001
4. MetaCase Consulting, *MetaCase Consulting website*. 2001. *http://www.metacase.com*. Last accessed 15.8.2001
5. Nelson, M., *Robocode*. 2001, IBM Alphaworks. *http://alphaworks.ibm.com/tech/robocode*. Last accessed 18.9.2002
6. Kelly, S. and J.-P. Tolvanen. *Visual domain-specific modelling: Benefits and experiences of using metaCASE tools*. in *ECOOP 2000*. 2000. Sophia Antipolis and Cannes, France.
7. Kieburtz, R.B., *Defining and Implementing Closed, Domain-Specific Languages*. 2000, Oregon Graduate Institute of Science & Technology: Beaverton, Oregon USA.
8. Fayad, M.E., D.C. Schmidt, and R.E. Johnson, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. 1999: John Wiley & Sons, Inc.
9. Harel, D., *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming, 1987. **8**(3): p. 231-274.
10. McWhirter, B., *blissed*. 2002, The Werken Company. *http://blissed.werken.com/*. Last accessed 18.9.2002
11. Pedrazzini, S., *The Jacaranda Framework*. 2000. *http://a.die.supsi.ch/~pedrazz/jacaranda/*. Last accessed 18.9.2002