Motivation and Hypothesis for Comparison between Component Frameworks and DSL Paradigms

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

Abstract

Both component frameworks and domain-specific languages aim to promote reuse and enhance product-line software development, but they approach the problem from different viewpoints. This paper provides motivation and hypothesis for comparison between the two paradigms. As a result we note some initial observations of the paradigms and present research questions for an empirical study.

Introduction

Even though the domain-specific language (DSL) paradigm has been studied in theory for almost twenty years, there are just a few success stories that were applicable only to small, limited domains. The emergence and slow maturization of metaCASE [Lyytinen and Tahvanainen, 1992] tools (e.g. MetaEdit+ from MetaCase Consulting [MetaCase Consulting, 2001b], ObjectMaker [MarkV Systems, 2001]) have made it possible to create and use new visual DSLs easier than before. The development of a DSL can be very costly, but the bullish claims (e.g. [MetaCase Consulting, 2001a], [Kieburtz et al., 1996]) about the productivity increases achievable by using DSL for product-line software development makes it a very attractive option for trying to overcome the infamous software crisis.

The crisis has not been left unattended on the more traditional side of software development either. After inventing object-oriented (o-o) development, the industry has recently tried to tackle the problem with its offspring, frameworks and components [Fayad and Johnson, 1999]. Although component frameworks and DSLs address basically the same issues [Durham, 1996], they approach the problem differently, which makes it interesting to study the similarities and differences between the two paradigms.

Framework development can also be costly because frameworks tend to be large and complex [Fayad et al., 1999], thus requiring well-defined domain knowledge to minimize modifications to a framework. However, the main difference between DSLs and component

frameworks is that a component frameworks relies on some general-purpose language (GPL, such as Java or C++) that loosely defines the "language" [Razavi, 1999], [Goldberg, 1995] (i.e. interfaces and abstract classes components are based on), whereas a DSL typically uses specific code generators (or, if a DSL is derived from some GPL, often a pre-compiler transforming the DSL to a GPL [Kieburtz, 2000]). This study focuses on the differences between the two paradigms and the implications they cause.

The purpose of this paper is to provide motivation for comparison between DSL and component frameworks paradigms. It also serves as a basis of hypothesis for empirical studies. It describes the paradigms and discusses the differences between the two. The paper is organized as follows: First we will discuss how the two paradigms are placed on the Information Resource Dictionary System (IRDS) Framework [International Standard ISO/IEC DIS 10027, 1989], as meta levels of information are a significant part of domain-specific modelling theory. We note some interesting points about component frameworks and, in general, object-oriented programming placement in the IRDS framework. Second, we take a look at the differences between the paradigms and analyze their strengths and weaknesses. As a conclusion, we present directions for future research and present the hypothesis that shall be tested in future empirical studies.

Placing the paradigms in the IRDS framework

The Information Resource Dictionary System (IRDS) is a framework describing the type of information used in information processing systems. It is used here to show how the different development activities in the two paradigms are placed on its levels, and to explain how they are mapped to each other. The framework contains four information levels: the IRD schema level, the IRD definition level, the IRD level, and the application level. Each upper level describes what kind of information is presented on a level below it, thus holding meta-information and serving as a meta-level for the lower level.

The IRD schema level contains meta-meta type information, such as a metamodelling language that is used to define new modelling languages. On the next, IRD definition level, new metamodels are specified using the metamodelling language. Method engineering, such as specifying UML models or building any DSL takes place on this level. The actual development by using a method or a DSL is done on the IRD level. Developers use types defined in a metamodel by instantiating them, e.g. by creating new classes in a class diagram. The fourth, application level is for the outcomes of the development, i.e. the actual instances of developed applications. This level is not interesting for the comparison and thus is not considered further.

Without any domain-specific aspects the information level structure is quite clear and understandable. However, when we add domain-specific information to it, we run into problems. If we make a domain-specific method by extending some general-purpose method, are we still on the same meta-level or on a level below it? If we use a general-purpose method to create a new method, we do not instantiate it (and thus move to lower meta-level as explained in [Zhang and Lyytinen, 2000]) but specialize it. To separate specialization from instantion, we need another dimension besides the information type dimension that represents the generality of information, i.e. its domain-specificity.

As shown in the Figure 1, we can point out how different activities in both paradigms are positioned on the three-level IRD layer architecture. On the left column we see the three information levels and next to it, the two paradigms side-by side. The paradigm columns have a generality dimension from left to right. The different activities are also positioned vertically so that they illustrate how well they fit into some information level, or how much of an activity's development concerns actually take place on that information level.

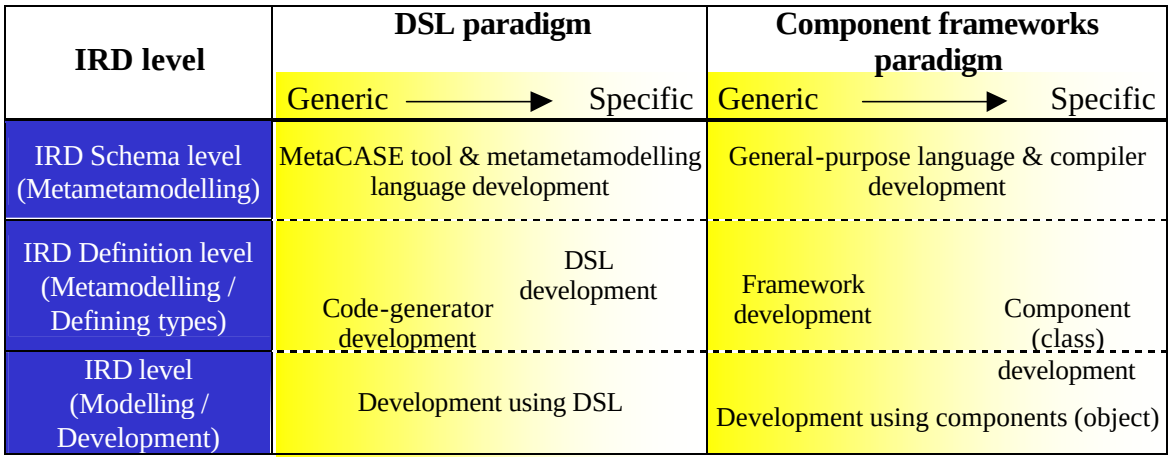

Figure 1: Positioning development concerns of the paradigms in the IRDS framework

Now, some researchers would argue that general-purpose languages should not be on the metametamodelling level as it is positioned in the figure, but on the metamodelling level, because it is used for programming applications, not for defining new programming languages. This is of course completely true with languages that do not allow the definition of new types and interfaces. However, when considering o-o languages, that is not necessarily so. As mentioned before, framework development often concerns development of interfaces and abstract classes, which is similar to metamodelling, because you define new types to be used for some special domain and pay less attention to modelling level issues. This is especially true for hybrid languages or some scripting languages, where an end-developer can use ready-made objects, but may not be able to define new ones at all. Therefore, we claim

that an o-o language is a kind of a metamodelling language for defining new modelling languages when it is used for creating domain-specific frameworks. In Thomas Jay Peckish II' words: "Inside every domain-specific framework, there is a language crying to get out" [Foote and Yoder, 1998].

In common o-o languages, the modelling and metamodelling levels are tightly tied together [Razavi, 1999]. Programmers constantly jump back and forth between metamodelling and modelling leves, when they create new types or classes and then work with an object of that type. Clearly, this is instantion, making o-o programming an activity that happens on both meta levels. It is such an integral part of o-o language principles that it is not often considered this way.

Hypothesis and research questions for an empirical study

To conduct an empirical study, our intention is to build a component framework written in Java for a suitable domain. A domain should be stable, well defined and focused on product-line development for proper comparison. On top of the component framework we plan to create a visual modelling language using the MetaEdit+ method workbench metaCASE tool [MetaCase Consulting, 2001b]. In addition to visual meta-modelling, this setup would need a code-generator to be built, taking care of mapping the domain-specific models to corresponding Java-classes. Code generation is also needed to instantiate objects and populate the attributes with proper values. A code-generator would contain many of the same components a component framework does, plus some more code for supporting the transformation from the models (e.g. state models) to an o-o application.

In the scenario describe above, we have two development environments sharing an identical domain, but based on two different paradigms. It allows us to easily conduct an empirical study on them. Basically, there are two fundamental differences between the paradigms:

1. In a DSL (developed with a metaCASE tool), metamodelling and modelling levels are clearly separated from each other, while in the component frameworks paradigm they are inherently tangled [Razavi, 1999].

2. The DSL paradigm raises the abstraction level of development. With a DSL, you can hide many of the low level issues that must be taken care of explicitly when using a framework based on some GPL. Thus, it can be said that a DSL is somewhat "closer" to the domain model; or, the elements of a DSL are closely related to the real domain elements [Kelly and Tolvanen, 2000].

In addition to these differences, there is also one additional benefit when using a DSL developed with MetaEdit+: It is visual. It allows a developer to see more at once, which should directly translate to better and quicker system understanding in its entirety and easier perception of object interactions.

As noted, a code-generator plays a big part in the DSL paradigm. While it is a challenging task to build a code-generator, it handles many of the same, if not more, responsibilities a framework does in a component frameworks paradigm. This should result in a faster development time, less error-prone development and less time needed in debugging. However, debugging in a DSL can be tedious; while there are some examples of working debuggers in DSL environments (such as [Faith, 1997], [Faith et al., 1997]), often bugs have to be nailed down by looking at generator-produced code (as in [Korhonen, 2000]). The situation is similar to the move from assembler languages to higher-level languages; while you could write a program with C, you still had to use the assembler debugger to trace the program execution to see where it went wrong.

Clear separation between metamodelling and modelling levels, using a metaCASE tool [Kelly and Tolvanen, 2000] for building a DSL, has both strongnesses and weeknesses. The development is more controlled, because end developers cannot introduce new types, but only use the power of expression allowed in a metamodel. This should result in less errors and more understandable models. It supports easier delegation of work: the gurus can be appointed to develop a code-generator and more experienced developers can take care of the metamodel, while novice developers may only be allowed to work with modelling issues. Logically, this also stiffens modifications to a metamodel, which can become troublesome if a domain is young and rapidly evolving.

It is probably safe to say that starting costs for DSL-based paradigm are even higher than for a component frameworks -based paradigm. However, if the claims noted in the beginning of this paper about enormous productivity increases when using DSL are true, it may suggest that if it is justified to build a component framework, it is also always justified to build a DSL. One of the most interesting research questions for empirical study is to find out where the productivity increase actually comes from.

In the planned empirical study the additional difference was that a DSL would be visual while the framework paradigm will not be. While there are many graphical IDEs for Java, most of the programming is still done textually, not graphically. However, many researchers in the field share the opinion that visual development offers potentially major productivity increases. This is also one of the possible reasons for productivity gains that a DSL may offer over framework-based development.

Conclusion and future research

In this paper we discussed and compared two paradigms - component frameworks and DSLs - against each other. We noted the similarity between the paradigms, as both are most suitable for product-line development, where high levels of reuse can be achieved. We also clarified how the development activities of the paradigms are positioned in the IRDS information meta-level framework. We noted that besides metamodelling levels, methods and models could also be differentiated in another dimension based on their generality. To finish, we presented an environment to conduct an empirical study and developed hypothesis and research questions for a comparative study.

Clearly, in some environments (e.g. [Kieburtz et al., 1996] [MetaCase Consulting, 1999]) a DSL can be proved as being highly beneficial for improving and speeding up development. One of the most interesting research questions for future research is to find out where the benefits of a DSL come from. It is also interesting to compare exactly how much faster development could be with a DSL when compared to an otherwise identical component frameworks paradigm.

Another research direction concerns how these two paradigms can benefit each other while studying how development of a DSL based on a component framework evolves. One reasonable and engrossing possibility (as already proposed by [Durham, 1996]) is that at first, when the domain is still young, you could develop software based on a light component framework, then iteratively enhance your framework. Finally, when domain matures, build a DSL on top of the framework.

To conclude, this paper provides motivation and initial observations to study these research problems further:

- What are the benefits of using a DSL over a component framework in product-line development?
- What is the trajectory to build a DSL?
- Is it possible to use a component framework as an evolutionary to a DSL?
- How much does a component framework need to be modified if it is used as a foundation for a DSL?
- What kind of models should be developed for a DSL and how to best transform them to a GPL to minimize modifications to an existing framework?

There is an empirical study currently going on to find out answers to these questions.

References

Durham, A. M. (1996) In *OOPSLA'96, A Framework for Run-time Systems and its Visual Programming Language*, San Jose, CA, USA.

Faith, R. E. (1997) *Debugging Programs After Structure-Changing Transformations* ,Ph.D. Dissertation.The University of North Carolina.

Faith, R. E., Nyland, L. S. and Prins, J. F. (1997) In *USENIX Conference on Domain-Specific Languages, Khepera: A System for Rapid Implementation of Domain-Specific Languages*, Santa Barbara, CA, USA.

Fayad, M. E. and Johnson, R. E. (1999) *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, John Wiley & Sons, Inc.

Fayad, M. E., Schmidt, D. C. and Johnson, R. E. (1999) *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, John Wiley & Sons, Inc.

Foote, B. and Yoder, J. (1998) In *PLOP'98, Metadata and Active Object-Models*, Monticello, Illinois.

Goldberg, A. (1995) In *OOPSLA '95, What Should We Teach?*, Austin, TX, USA.

International Standard ISO/IEC DIS 10027 (1989) *ISO Information processing systems: Information Resource Dictionary System (IRDS) Framework*, .

Kelly, S. and Tolvanen, J.-P. (2000) In *ECOOP 2000, Visual domain-specific modelling: Benefits and experiences of using metaCASE tools*, (Eds, Bezivin, J. and Ernst, J.) Sophia Antipolis and Cannes, France.

Kieburtz, R., McKinney, L., Bell, J., Hook, J., Kotov, A., Lewis, J., Oliva, D., Sheard, T., Smith, I. and Walton, L. (1996) *A Software Engineering Experiment in Software Component Generation*, Oregon Graduate Institute of Science & Technology, Portland, OR.

Kieburtz, R. B. (2000) *Defining and Implementing Closed, Domain-Specific Languages*, Oregon Graduate Institute of Science & Technology, Beaverton, Oregon USA.

Korhonen, K. (2000) *Build Your Own Lego: Components, Architecture and Processes in Component-Based Development* ,Master's Thesis.University of Jyväskylä, Jyväskylä, Finland.

Lyytinen, K. and Tahvanainen, V.-P. (1992) *Introduction: Towards the Next Generation of Computer Aided Software Engineering (CASE)*, In *Next Generation CASE Tools*, Vol. 3 (Eds, Lyytinen, K. and Tahvanainen, V.-P.) IOS Press, , pp. 1 - 7.

MarkV Systems (Accessed: 15.8.2001) *MarkV Website*, Access Media: http://www.markv.com/.

MetaCase Consulting (1999) *MetaEdit+ Revolutionized the Way Nokia Develops Mobile Phone Software*, MetaCase Consulting, Inc.

MetaCase Consulting (2001a) *Domain-Specific Modelling: 10 Times Faster than UML*, MetaCase Consulting, Inc., Jyväskylä.

MetaCase Consulting (Accessed: 15.8.2001) *MetaCase Consulting website*, Access Media: http://www.metacase.com.

Razavi, R. (1999) In *OOPSLA'99, Building an End-user-oriented Application Framework by Meta-programming A Case Study*, Denver, Colorado, USA.

Zhang, Z. and Lyytinen, K. (2000) *A Framework for Component Reuse in a MetaCASE Based Software Development*, In *Information Systems Engineering: State of the Art and Research Themes*(Eds, Brinkkemper, S., Lindencrona, E. and Solvberg, A.) Springer.