Separating Concerns with Domain Specific Languages

Steve Cook

Microsoft UK Ltd, Cambridge
steve.cook@microsoft.com

Abstract. I'll talk about the separation of concerns in the development of large distributed enterprise systems, how to manage it using domain specific languages, and how to build these languages. This brief note outlines some of the topics I'll cover.

1 Separation of Concerns

Most developments in programming language design are intended to improve the ability of the programmer to separate the expression of different concerns. This has progressively led to the development of language features such as procedures, abstract data types, objects, polymorphic type systems, aspects, and so on.

We're now moving into an era when the normal case of software development is distributed and heterogeneous, with the internet playing a pivotal role. It's simply not practical today to use a single programming language to create all aspects of a large and complex computing system. Different technologies are used to implement user-interfaces, business subsystems, middleware, databases, workflow systems, sensors, etc. Enterprise programming stacks include as first-class participants a variety of inter-related programming and scripting languages, databases, metadata and configuration files. Most programming projects involve interoperating with what is already there, which requires interfacing to existing technology stacks.

In such a world, concerns such as the structure and organization of business data and processes inherently span multiple technologies. A given business concept will show up in the user interface, in the formats used to communicate between components, in the interfaces offered from one component to another, in the schemas for databases where business data is stored, and in the programming logic for manipulating all of the above. Even the simplest change, such as changing the name of the business concept, impacts all of these pieces. Such concerns cannot possibly be effectively separated by improving programming language design. How then can we approach this problem?

2 Development Using Domain Specific Languages

A promising approach has been described variously as "Language-Oriented Programming" [1], "Language Workbenches" [2], "Generative Programming" [3] and "Model Driven Engineering" [4]. All of these phrases essentially describe the same

D. Lightfoot and C. Szyperski (Eds.): JMLC 2006, LNCS 4228, pp. 1 – 3, 2006.

pattern: given a class of problems, design a special-purpose language – a Domain Specific Language or DSL - to solve it.

A simple (and old) example of this pattern is the language of regular expressions. For example, using the .Net class System.Text.RegularExpression.Regex, the regular expression "(?<user>[^@]+)@(?<host>.+)" applied to a string of characters will find email addresses in it, and for each address found, appropriately extract the user and host variables. Programming such a procedure directly in a general-purpose language is a significantly larger and more error-prone task.

In developing complex enterprise systems, it is increasingly the case that graphical languages can be used to express certain concerns most effectively. Business workflows, business data, and system, application and data centre configuration are obvious candidates for graphical representation. Also textual languages, while effective for inputting large programs, may not be the most effective medium for displaying, analyzing and interpreting these programs.

Putting these ingredients together provides the motivation for an emerging class of graphical language-processing tools, which includes the DSL Tools from Microsoft [5], the Generic Modeling Environment (GME) from Vanderbilt University [6], and commercial examples from MetaCase, Xactium and others. These tools enable the language author to design and implement the abstract and concrete syntax for a DSL, together with the ancillaries needed to integrate the language into a development process.

Of course it is not sufficient simply to design what a DSL looks like; it is also necessary to give its expressions meaning, which in practical terms means to generate executable artifacts from it: these will most likely be programs in more general-purpose languages, together with configuration files, scripts and metadata, that can be deployed together to implement the intention of the developer.

As soon as generation is introduced into the development process, there is the possibility of developers changing the generated artifacts. Uncontrolled, this will break the process: the source form of the DSL will become out of date and useless. Alleviating this issue is one of the main challenges of making DSLs successful. Not all artifacts can be generated from DSLs, so it is essential to be able to interface generated artifacts with hand-coded ones: various language techniques such as partial classes [7] can enable this.

3 Software Factories

A DSL can provide a means to simplify the development of one area of concern. But the development of large distributed applications involves the integration of multiple areas of concern, with multiple stakeholders manipulating the system via multiple different viewpoints.

Managing the complexity of such a development involves delivering appropriate languages, tools and guidance to individual participants in the process at the right place and time. Enabling this is the province of Software Factories [8], an approach

to software development that focuses on the explicit identification of viewpoints in the development process, the definition of DSLs, tools and guidance to support these viewpoints, and the delivery of these capabilities to individuals during the enactment of the process.

References

1. Dimitriev, S. Language-Oriented Programming: The Next Programming Paradigm, http://www.onboard.jetbrains.com/is1/articles/04/10/lop/
2. Fowler, M. Language Workbenches: The Killer App for Domain Specific Languages? http://martinfowler.com/articles/languageWorkbench.html
3. Czarnecki, K. and Eisenecker, U.W. Generative Programming – Methods, Tools and Applications. Addison-Wesley (2000).
4. Bézivin J., Jouault F, and Valduriez P. On the Need for Megamodels. Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (2004).
5. DSL Tools Workshop. http://msdn.microsoft.com/vstudio/DSLTools/
6. Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J. and Volgyesi, P. The Generic Modeling Environment. Proceedings of WISP'2001, May, 2001. http://www.isis.vanderbilt.edu/Projects/gme/GME2000Overview.pdf
7. C# programming guide, http://msdn2.microsoft.com/en-us/library/wa80x488.aspx
8. Greenfield, J., Short, K., Cook, S., Kent, S. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. Wiley (2004).