UPGRADE is the European Journal for the Informatics Professional, published bimonthly at <http://www.upgrade-cepis.org/>

Publisher
UPGRADE is published on behalf of CEPIS (Council of European Professional Informatics Societies, <http://www.cepis.org/>) by Novática <http://www.ati.es/novatica/>, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática,* <http://www.ati.es/>)

UPGRADE monographs are also published in Spanish (full version printed; summary, abstracts and some articles online) by Novática

UPGRADE was created in October 2000 by CEPIS and was first published by Novática and INFORMATIK/INFORMATIQUE, bimonthly journal of SVI/FSI (Swiss Federation of Professional Informatics Societies, <http://www.svifsi.ch/>)

UPGRADE is the anchor point for UPENET (UPGRADE European NETwork), the network of CEPIS member societies' publications, that currently includes the following ones:
• Informatica, journal from the Slovenian CEPIS society SDI
• Informatik-Spektrum, journal published by Springer Verlag on behalf of the CEPIS societies GI, Germany, and SI, Switzerland
• ITNOW, magazine published by Oxford University Press on behalf of the British CEPIS society BCS
• Mondo Digitale, digital journal from the Italian CEPIS society AICA
• Novática, journal from the Spanish CEPIS society ATI
• OCG Journal, journal from the Austrian CEPIS society OCG
• Pliroforiki, journal from the Cyprus CEPIS society CCS
• Pro Dialog, journal from the Polish CEPIS society PTI-PIPS
• Tölvumál, journal from the Icelandic CEPIS society ISIP

Cover page designed by Concha Arias Pérez
"Golden Ratio" / © ATI 2008
Layout Design: François Louis Nicolet
Composition: Jorge Llácer-Gil de Ramales

Editorial correspondence: Llorenç Pagés-Casas <pages@ati.es>
Advertising correspondence: <novatica@ati.es>

UPGRADE Newslist available at
<http://www.upgrade-cepis.org/pages/editinfo.html#newslist>

ISSN 1684-5285

Monograph of next issue (June 2008)

"Next Generation Technology-Enhanced Learning"

(The full schedule of **UP**GRADE is available at our website)

CEPIS UPGRADE

The European Journal for the Informatics Professional
http://www.upgrade-cepis.org

Vol. IX, issue No. 2, April 2008

* This monograph will be also published in Spanish (full version printed; summary, abstracts, and some articles online) by **Novática**, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*) at <http://www.ati.es/novatica/>.

The Domain-Specific IDE

Steve Cook and Stuart Kent

Years of pursuing efficiencies in software development through model-driven development techniques have led to the recognition that domain-specific languages can be an effective weapon in the developer's armoury. But these techniques by themselves are necessarily limited; only by assimilating them into the overall context of a domain-specific development process and tools can their real power be harnessed.

Keywords: Domain-Specific Languages (DSL), Domain-Specific Tools (DST), Model-Driven Architecture (MDA), Model-Driven Development (MDD).

1 Introduction

The development of information systems is getting increasingly complex as they become more and more distributed and pervasive. Today's advanced software developer must be familiar with a wide range of technologies for describing software, including modern object-oriented programming languages, eXtensible Markup Language (XML) and its accessories (schemas, queries, transformations), scripting languages, interface definition languages, process description languages, database definition and query languages, and more. Translating from the requirements of a business problem to a solution using these technologies requires a deep understanding of the many architectures and protocols that comprise a distributed solution. Furthermore, end-users expect the result to be fast, available, scaleable and secure even in the face of unpredictable demand and unreliable network connections. It can be a daunting task.

In areas other than software development, such as electronic consumer products (TVs and HiFis), cameras, cars and so on, we have come to expect a high degree of reliability at low cost, coupled increasingly in many cases with the ability to have items customized to satisfy individual needs. These expectations are met because of advances in industrial manufacturing processes made over many decades. Building a car or a television involves the coordination of a complex chain of manufacturing steps, many of which are wholly or partially automated.

We would like to apply similar principles to the construction of software. The main difficulty in doing so is that we have not yet developed techniques for software description that allow different concerns within the software development process to be effectively separated and effectively integrated. Although we increasingly use different languages for different tasks (programming languages for writing application logic, XML for transmission of data between application components, Structured Query Language (SQL) for storing and retrieving data in databases, Web Services Description Language (WSDL) for describing the interfaces to web-facing components) there are many complexities involved in getting these languages to work effectively together.

Authors

Steve Cook works at Microsoft, and is the software architect of the Domain-Specific Language Tools which are part of Microsoft Visual Studio. He is currently working on future versions of these tools. Previously he was a Distinguished Engineer at IBM, which he represented in the UML 2.0 specification process at the OMG. He has worked in the IT industry for more than 30 years, as architect, programmer, author, consultant and teacher. He is a member of the Editorial Board of the Software and Systems Modeling Journal, a Fellow of the British Computer Society, and holds an Honorary Doctor of Science degree from De Montford University (United Kingdom). <steve.cook@microsoft.com>.

Stuart Kent is a Senior Program Manager on the Visual Studio team in Microsoft. Stuart joined Microsoft in 2003 to work on tools and technologies for visual modelling. This culminated in the Domain-Specific Language Tools, which are now part of the Visual Studio core tooling platform and are described in a recent book (Domain-Specific Development with Visual Studio DSL Tools) that he co-authored. Before joining Microsoft, Stuart was an academic and consultant, with a reputation in modelling and model driven development. He has over 50 publications to his name and made significant contributions to the UML 2.0 and MOF 2.0 specifications. He is a member of the editorial board of the Software and Systems Modeling journal, and on the steering committee for the MoDELS series of conferences. He has a PhD in Computing from Imperial College, London. <stukent@microsoft.com>.

2 Domain Specific Modelling Languages

2.1 Model Driven Development (MDD)

Model driven development is an approach to software development where the main focus of attention shifts from writing code by hand to dealing with higher level abstractions (models). The approach aims to increase productivity, improve reliability and be more predictable.

Typically, a model driven development solution develops incrementally in stages, as follows.

In the first stage, a solution starts out as a way of getting an initial boost in productivity by generating code that is duplicated within and between software applications, instead of writing it by hand. In this situation, the model provides the information that is variable in amongst the duplicated code, and the code generators merge this with boilerplate code to produce the final result. As it is unlikely that all the required code can be generated from the model, the architecture of the software application may need to be adjusted to ensure that the generated code is kept separate from any hand written code.

In the next stage, as the code generators become more complex, it is realized that much of the duplication can be removed by creating a framework using constructs, where available, in the underlying programming language. This generally won't remove all the duplication, but it will remove bloat from the code generators and make them easier to maintain.

In a third stage, it may be possible to remove the need for code generation altogether, and write the framework so that it directly interprets the model.

In subsequent stages, once models have become a first class citizen in the software development process, they can then be treated as the target of transformations from yet more abstract models.

However, there is a lot to consider, even in the first stage, including:

- What language should the model be expressed in?
- What's the best way to write the code generators?
- How do we expose the code generators to the users of the tools, for example when a ship-blocking bug needs to be fixed?
- How do we ensure that generated code can be mixed with non-generated code so that regeneration does not overwrite the non-generated code?
- How do we ensure that generated code builds and exhibits the correct behaviour?
- How is the generated code tested?
- How does a developer debug through the generated code? Should he need to?

There aren't easy answers for all these questions. Indeed, in the next section we argue that questions such as these require us to think in terms of making the whole tooling environment domain specific, with models and code generators being only a part of a more holistic, integrated environment. Nevertheless, when models are an important part of the overall solution, the most burning question is the first: what language is used to express the models? That's the focus of the remainder of this section, and, when considering an answer, it's worth noting that answers to the other questions involve writing tools, including code generators, which must be able to access the models programmatically. The range of situations in which model driven development provides a productive approach depends directly on how easy it is to build and test those tools.

2.2 UML

A language often associated with MDD is the Unified Modeling Language (UML), which is a standard of the Object Management Group (OMG)[1] .The development of the UML started during the early 1990s, when it emerged as a unification of the diagramming approaches for object-oriented systems developed by Grady Booch, James Rumbaugh and Ivar Jacobson. First standardized in 1997, it has been through a number of revisions, most recently the development of version 2.

UML is large and complicated, version 2 especially so. To understand UML in any depth it is important to understand how it is used. We follow the lead of Martin Fowler, author of "UML Distilled," one of the most popular introductory books on UML. Martin divides the use into UMLAsSketch, UMLAsBlueprint, and UMLAs ProgrammingLanguage (for more details see <http://martinfowler.com/bliki>).

UMLAsSketch is very popular. Sketches using UML can be found on vast numbers of whiteboards in software development projects. To use anything other than UMLAs Sketch as a means of creating informal documentation for the structure of an object-oriented design would today be seen as perverse. In this sense, UML has been extremely successful, and entirely fulfilled the aspirations of its creators who wanted to eliminate the gratuitous differences between different ways of diagrammatically depicting an object-oriented design.

UMLAsProgrammingLanguage is an initiative supported by a rather small community, which is unlikely to gain much headway commercially, and which we will not dwell upon.

UMLAsBlueprint characterizes the use of UML in MDD. In this role, UML suffers from two problems:

- *Bloat*. In most MDD solutions, it is likely that only a small subset of the language will be applicable. So, although tools implementing UML do generally provide rich programmatic access to models created using them, the surface area of the API against which tools - such as code generators - are written is large, which just makes it that much harder to code against. Also, unless the chosen graphical UML editor allows significant parts of the language to be 'switched off', the user of the MDD solution needs to have external guidance on how to use the UML tool within the context of that solution.

- *Not domain specific*. In order to drive code generators in MDD, it's necessary to develop models that fit the domain, that can supply the exact information required by the code generators to fill the gaps in the boilerplate and produce fully executable code, and which conform to the

[1] The OMG also uses the trademarked term *Model Driven Architecture (MDA)* for its particular take on MDD.

necessary validation rules that ensure the code generated is well-formed and builds correctly. UML does support an extension mechanism, called UML Profiles, which allows additional data and validation rules to be added to a model. But this cannot do anything about fundamental conceptual mismatches between a domain and UML: at best, profiles allow UML to be used for MDD where there is a reasonable conceptual match between the domain and UML, and all that is required is some additional data and validation rules to make the models precise enough to drive code generators.

2.3 Domain-Specific Modelling Languages

An alternative to using the UML is to define a domain-specific language specially designed for the MDD solution.

At first sight, this seems like a significant undertaking, especially when you consider that in an MDD context the language needs to be supported by an editor, often graphical, which validates models and delivers them in a machine-readable form, as well as providing rich API access to those models. Indeed, a reason that implementers of MDD solutions may have opted for the sub-optimal UML-based approach is that it means they don't have to build their own graphical editor!

However, this is changing. There are now environments available such as Microsoft's Domain Specific Language Tools (DSL Tools) [1], Eclipse Graphical Modeling Framework (GMF) [2] and MetaCase's MetaEdit+ [3] which significantly reduce the cost of creating your own graphical, domain specific modelling language and editor. These tools generate a clean API for accessing models, and also include support for writing code generators. The editors that are created support graphical modelling using custom graphical notations, and allow rich model validation constraints to be included.

One criticism aimed at using DSLs is that you can end up creating a range of slightly different languages, one for each MDD solution. This can be confusing to users, leading to a lot of very similar looking but subtly different languages and editors, and also confusing to tool builders who might end up writing against similar but subtly different APIs. In contrast, in the UML approach you have a base language which can be shared between different solutions, and then have an extensibility mechanism which allows the base language to be customized for each MDD solution. The problem with the UML approach is not the principle of having a base language that is then extended and customized, but the fact that the UML has not been architected well to support this. For this approach to be effective, the UML should have been defined as a collection of small, loosely coupled, unconstrained base languages capturing specific modelling styles (class diagram style, component style, state diagram style, sequence style etc.), where any detailed content was defined through extensions (e.g. the Java, .Net and Object-Oriented analysis class diagram extensions).

3. The Domain-Specific IDE

We've suggested that model-driven development can be made more efficient by designing and implementing domain specific modelling languages aimed at specific software development problems. But there's much more to software development than modelling, and we'd like to make the entire software development process more efficient, not just the modelling part of it.

Models form one aspect of the software development experience. We must integrate this aspect with others across the entire lifecycle: envisioning, architecture, design, coding, debugging, testing, deploying and managing. We are increasingly discovering that this entire lifecycle is domain-specific, and that implementing domain-specific software development languages goes hand-in-hand with implementing domain-specific processes.

Narrowing the domain means building more tools. These are not simply modelling tools, or domain-specific extensions to programming languages. The domain-specific lifecycle also requires domain-specific commands, user interfaces, processes and guidance. Elsewhere we've described these requirements as "software factories" [4]. We've found that this can be an overloaded term, referring as it does to several distinct concepts:

- an organization designed to develop a particular kind of software;
- a set of processes that execute to deliver a particular kind of software;
- an integrated set of interactive software tools designed to support such processes.

In this article we are mainly interested in the third of these, which we'll call here the Domain-Specific Interaction Development Environment, or Domain-Specific IDE. We'll look now at some of the requirements for this.

3.1 Agility

Agile programming embraces evolutionary change throughout the software development lifecycle and is increasingly recognized as best practice for software development. The Domain-Specific IDE must support agile software development practices. The IDE must avoid processes or commands that force the developer into premature commitments that are expensive and time-consuming to reverse. Examples of these are code generation steps or "wizards" that cannot be repeated at a later time. The use of models within the IDE can help with this, as illustrated by Microsoft's Web Services Software Factory: Modeling Edition [5].

3.2 Integrating Multiple Languages

The definition of software inevitably involves a combination of languages. Even in the most traditional environment there will be distinct languages for programming and for defining and manipulating data. Today's popular software stacks involve multiple languages: for .NET they in-

[2] See <http://www.martinfowler.com/bliki/InternalDslStyle.html>.

clude C#, Visual Basic, SQL, and many dialects of XML e.g. XAML, with domain-specific languages added to the mix.

As soon as the artefacts implemented by these languages cross-reference each other, as they must, we find breakdowns in agility. For example, renaming a class or a namespace in a code file can cause compilation errors in a XAML file which refers to the class or namespace. Finding and correcting such errors can be costly, and a better solution would be a refactoring engine that spans multiple languages. This means that new domain-specific languages should be able to participate in such a refactoring engine, in order to be first-class citizens in the IDE.

One approach to alleviating such breakdowns is to expand the scope of one language to cover more of the lifecycle. Good examples of this approach are the latest versions of C# and VB .NET, which incorporate LINQ (Language-Integrated Query) giving strongly-typed integrated capabilities for accessing data stores. Further down this route would be more facilities in these languages for enabling embedded DSLs[2]. Few truly multi-paradigm languages exist today, though. LISP was the first example of a multi-paradigm language, and has fallen out of widespread use in favour of the strongly-typed object oriented programming languages which are commonly in use today. We're sure that these languages do not represent the ultimate destiny of programming: new developments in programming languages such as F# [6] enable considerable increases in expressive power and the design of rich abstractions. F# is complicated, though, and truly mastering it may be out of the reach of many software developers.

But however powerful the language, it is unlikely ever to be the case that a single language can successfully span the entire software lifecycle. Whether through limitations in languages or through limitations in their users, techniques for integrating multiple languages will inevitably be required.

3.3 Code Generation and "Reverse Engineering"

Code generation is an increasingly prevalent feature of modern IDEs that allows designs to be expressed in domain-specific terms. Code is often generated from models, defined in a domain-specific modelling language or in UML. Code can also be generated from other artefacts such as XML dialects, domain-specific textual languages, data held in databases, data captured from a What You See Is What You Get (WYSIWYG) editor (*e.g.* Windows Forms) or via User Interface (UI) automation (wizards, snippets etc). The advantages of code-generation are a substantial reduction in cost and errors for the creation of repetitive boilerplate code. Code generation does, however, have several potential disadvantages:

- *Non-repeatability*. A badly-designed code generation system can lose the developer's input after generation, forcing the developer to commit to a set of values and making it difficult to change their mind.
- *Hand-modification*. A system that requires code to be modified by hand after it has been generated may prevent a re-generation without erasing the hand-modifications, unless markers are placed in the generated code to indicate the hand-written parts, which can make the code unpleasant to read.
- *Difficulty of interfacing generated and hand-written code*. Languages features such as C#'s partial classes, in which a single class can be partially defined in multiple files, are essential to enable the combination of generated and hand-written code.
- *Difficulty of modification*. If the generated code is not what is required, for example it needs to be modified to introduce additional aspects such as logging or events, then it may be necessary to change the code generator itself to make such modifications. Such changes can be error-prone and incur a considerable test burden.

In the early days of model-driven development there was a strong tendency amongst the vendors of model-driven tools to claim the capability to do "reverse engineering", i.e. to create a model by reading and parsing a codebase. It is wrong to think of such a procedure as the inverse of code generation. Useful tools can be created for visualizing a large codebase, for example showing namespaces, dependencies, class hierarchies, call graphs etc. These tools operate at the same level of abstraction as the code and are used to help understand it at that level. By contrast, code generation schemes transform representations from a domain to another, more general domain. It is possible to reverse the generation process to visualize generated code in terms of the source artefacts from which it is generated, but it is not generally feasible to extract domain-specific representations from an arbitrary hand-written codebase.

3.4 Validation, Debugging and Testing

One of the most compelling advantages of domain-specific representations is the ability to validate software artefacts at an appropriate level of abstraction. It is much easier, for example, to establish that communication paths in a layered architecture conform to the architecture's rules in a model of layers, components and connections, than it is by trying to deduce the communication paths from the eventual code. Simple constraint violations, such as non-uniqueness of property names or cycles in supposedly acyclic graphs, can be detected and fixed as early as possible in the lifecycle.

Validation can be "hard", enforced by a user experience that simply does not allow invalid configurations to be created, or "soft", i.e. evaluated on demand with error and warning messages referring the developer to the source of the problem. Hard validations are usually more expensive to implement, but offer a more habitable user experience; soft validations rely on carefully written error messages that effectively direct the user to the root problem. It is important to be able to save invalid representations: the IDE should never force the developer to re-organize their life in order simply to save their work.

Debugging should also offer a domain-specific experience. If the software is described in terms of domain-specific concepts then its execution may be observed and stepped through in terms of the same concepts. Implementing a domain-specific modelling language, for completeness, should include implementing debugger plugins that enable program execution to be visualized and controlled in domain-specific terms.

Domain-specific development also impacts testing. The designer of a domain-specific language has two kinds of testing to think about: testing that the DSL does what it is supposed to do, and providing an environmen, e.g. a generated test harness[3] , for users of the DSL to test the systems that they build using it.

3.5 Building the Domain Specific IDE

We've observed that narrowing the domain can improve productivity, but involves building more tools, which in turn means that it must be relatively cheap to do so. We tackle this by applying domain-specific techniques to the tool-building problem. We analyze the domains involved in tool-building, and build tools that support development in those domains.

Work on model-driven development over the past decade or so has involved a lot of thought about "meta-modelling". This subject is fraught with misunderstanding and confusion. Popular definitions are simply wrong: for example, the oft-heard definition of a meta-model as "a model of a model" is both unhelpful and incorrect. If a meta-model is anything, it is a model of the concepts expressed by a modelling language. Discussions about meta-modelling habitually degenerate into a kind of cultish mysticism almost entirely unconnected with the business of making software.

In practice, a meta-model is a model which is used (usually through code-generation) to build some aspects of a modelling tool. The result is that the tool can be built more cheaply, and thus bring extra efficiencies to solving the actual problem at hand. We'd like to generalize this principle to the entire development environment. We don't just model modelling concepts, we model all aspects of how the IDE is constructed, extended and deployed. We're especially interested in how all of the pieces that constitute a domain-specific IDE extension can be modelled: these include languages, commands, extra data carried by the project system, forms and toolwindows to interact with that data, and so on.

4 Conclusion

This article has proposed that an important step forward in software development tools is the development of domain-specific interactive development environments. Such tools recognise the advantages of model-driven development and domain-specific languages, while assimilating these techniques into an overall development experience tuned for the specific problem at hand.

[3] In software testing, a test harness or automated test framework is a collection of software and test data configured to test a program unit by running it under varying conditions and monitor its behavior and outputs. <http://en.wikipedia.org/wiki/Test_harness>.

References

[1] Steve Cook, Gareth Jones, Stuart Kent, Alan Wills. "Domain-Specific Development with Visual Studio DSL Tools", Addison-Wesley 2007. Also see <http://msdn.com/vsx>.

[2] Eclipse Foundation. Graphical Modeling Framework (GMF). <http://www.eclipse.org/gmf/>

[3] MetaCase. MetaEdit+. <http://www.metacase.com/>.

[4] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent. "Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools", John Wiley 2004.

[5] Microsoft. Web Service Software Factory: Modeling Edition. MSDN Library. <http://msdn2.microsoft.com/en-gb/library/bb931187.aspx>.

[6] Microsoft Research. <http://research.microsoft.com/fsharp/fsharp.aspx>.