# Model Driven Development – The Case for Domain Oriented Programming

Dave Thomas and Brian M. Barry

Bedarra Research Labs
1 Stafford Rd Suite 412
Ottawa, Ontario, Canada K2H 1B9
1-613-860-1163

dave@bedarra.com, brian@bedarra.com

## ABSTRACT

In this paper, we offer an alternative vision for domain driven development (3D). Our approach is model driven and emphasizes the use of generic and specific domain oriented programming (DOP) languages. DOP uses strong specific languages, which directly incorporate domain abstractions, to allow knowledgeable end users to succinctly express their needs in the form of an application computation.

Most domain driven development (3D) approaches and techniques are targeted at professional software engineers and computer scientists. We argue that DOP offers a promising alternative. Specifically we are focused on empowering application developers who have extensive domain knowledge as well as sound foundations in their professions, but may not be formally trained in computer science.

We provide a brief survey of DOP experiences, which show that many of the best practices such as patterns, refactoring, and pair programming are naturally and ideally practiced in a Model Driven Development (MDD) setting. We compare and contrast our DOP with other popular approaches, most of which are deeply rooted in the OO community.

Finally we highlight challenges and opportunities in the design and implementation of such languages.

**Categories & Subject Descriptors:** Add here

**General Terms:** Management, Languages, Documentation, Design, Human Factors.

**Keywords**: Domain Driven Development, Model Driven Development, Domain Specific Languages, End User Programming, Programming By Professional End Users.

## 1. INTRODUCTION

### 1.1 Domain Oriented Programming

In this paper, we offer an alternative vision for domain driven development (3D). Our application model based approach emphasizes the use of generic and specific domain oriented programming (DOP) languages. DOP employs strong specific languages, which allow knowledgeable end users to succinctly express their needs in the form of an application computation.

Most 3D approaches and techniques are targeted at professional software engineers and computer scientists. We argue that DOP is a promising alternative to such 3D approaches. Specifically we are focused on empowering application developers who have extensive domain knowledge as well as sound foundations in their professions, but who may not be trained in computer science.

We note that high-level programming languages have long been found useful in specific domains, and that domain specific languages are also not new. We therefore make no claims that DOP is a new idea; rather it is a good old idea that should be rediscovered.

DOP languages typically provide one or more strong computational metaphors that allow domain concepts to be readily modeled. DOP languages have simple syntax and clean semantics although the underlying semantic account may be extensive. They often achieve their expressiveness by a strict uniformity of operations and types. DOP languages allow the domain developer to map domain abstractions into DOP abstractions.

Most are learned by example or through trial and error rather than with formal training. They lend themselves to this since the majority are interactive languages featuring direct execution. Hence they support *think and execute* as opposed to *design code and test* development.

Domain Specific Languages (DSL) [19] are closely related to DOP. A DSL is a machine-processable language whose terms are derived from a specific domain model, which is used for the definition of components or software architectures supporting that domain. A DSL is tailored for a particular application domain, and captures precisely the semantics of that application domain – no more, no less. A DSL allows one to develop software for a particular application domain quickly and effectively, yielding programs that are easy to understand, reason about, and maintain [21]. DOP languages are frequently used as a base to implement DSLs.

In the next section we provide a survey of experiences with end-user programming, which we regard as precursors to DOP. We follow with a brief discussion of DOP development practices, in which we observe that many of the best practices such as patterns, refactoring, and pair programming are alive and well in these communities. We then compare and contrast DOP with other popular approaches, most of which are rooted deeply in the OO


*OOPSLA '03*, October 26-30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

community. Finally we highlight challenges and opportunities in the design and implementation of DOP languages.

## 2. Experiences With End User Programming

The motivation for DOP draws heavily on our own experiences and those of others with end-user programming. End-user facilities typically are designed for application developers ("blue collar programmers") rather than formally trained software professionals. Application developers are not, as some would suggest, people who "can't learn difficult computer science concepts". They are rather skilled individuals who have chosen to focus their creative energies on a particular application domain, rather than on computing per se. They typically have extensive domain knowledge as well as sound formal foundations in their professions. We provide a brief survey of experiences with end-user programming, from both current and past practice, which have motivated many of our ideas about DOP.

### 2.1 4GLs

The term 4GL was coined by James Martin to refer to non-procedural high-level languages designed to simplify CRUD (Create, Read, Update, Delete) programming and report writing.

RPG exemplifies the success of 4GLs. The IBM System/38 (which evolved into the AS/400) was essentially an RPG machine and its uniformity and robustness allowed ISVs to quickly develop reliable business applications. For many years AS/400 customers, even though they had far fewer development staff than mainframe operations, achieved comparable or greater productivity. This was despite the fact that the programmers often had a professional education in business rather than in computer science. Most practitioners attributed this remarkable productivity directly to the power of RPG to model business domain abstractions, combined with the capability to have OS facilities easily accessible from within the same language.

4GLs have lately (rather unfairly) acquired a somewhat unsavoury reputation compared with more recently popular approaches to application development such as Microsoft .Net or Sun J2EE. We argue strongly that most programmers were likely better off using 4GLs than these more "modern" technologies. Successful 4GLs such as SQL, Focus, RPG-II, Adabas, Powerhouse, Mapper and Synon increased programmer productivity by providing language constructs that modeled the application domain. Further they did so without burdening the user with the complexity of extraneous computer science concepts.

4GLs may not always be well suited to building complex applications, but a major advantage of 4GLs is that simple things can be implemented simply (an advantage we would suggest is not shared by J2EE or .Net). In practice, a surprisingly large number of commercial applications fit one of a few basic architectural patterns, and 4GLs often do an excellent job of capturing and supporting these patterns. RPG, for example, leverages the sort-merge file comparison used in sequential file processing for CRUD and reporting applications.

While it may not be readily apparent, in fact many of today's eBusiness applications perform exactly the same sort of CRUD operations across multiple data sources, with presentation to a web browser instead of a terminal and printer. Developers have observed these patterns in J2EE applications, and have begun to address them with generators, which take XML descriptions. More recently, new languages such as Xquery and Xduce [25] are appearing, which attempt to provide language solutions.

### 2.2 Spreadsheets

Since their introduction in VisiCalc over twenty years ago [13], spreadsheets have become such an accepted part of end-user computing that most spreadsheet users are not even aware that they are programming. Spreadsheets illustrate 3D at its most powerful: users conceive (analyze), organize (design), and record (code) their solutions to computational problems, all within the domain abstraction of financial tables. The abstraction is so natural and powerful that basic spreadsheet programming requires little or no instruction for anyone with even rudimentary accounting skills.

The spreadsheet is an excellent example of a strong, specific solution. It is very good at what it does, and not particularly useful for any problems that don't fit the basic abstraction of financial tables. For example, complex control constructs (like filters or iterators) are difficult to implement in spreadsheets, usually requiring redundant replication of data (but see vector languages below which offer another 3D approach which handles this problem very easily). The Analyst [17], an OO spreadsheet designed to provide diverse computational support for intelligence analysts, provides the most compelling example of just how far the spreadsheet metaphor can be pushed.

### 2.3 Visual Languages

Visual languages (e.g. Prograph [14], LABView [18], IBM VisualAge [24]) allow users to program directly with visual abstractions ("graphics") such as dataflow diagrams. Advocates of visual programming [10] [11] claim that the use of graphics, which provide concrete representations for program abstractions, can dramatically simplify the programming task. Once again, if the underlying application domain fits the graphical abstraction (e.g. the problem really is a dataflow problem in the Prograph case or message flow in VisualAge), users typically experience good results [24]. LABView in particular is likely the most successful VPL and is widely used in process engineering for both modeling and direct execution.

### 2.4 Rule Programming

There are many business and engineering applications where all or a substantial part of the application is naturally represented using sets of rules. Rules are particularly attractive when the application needs to evaluate a complex set of conditions such as policies. Knowledgeable end users can directly define, modify, test and execute rule programs. Rule languages vary from simple decision tables that have been used for years in IT, to specific rule languages, to expert systems such as Clips [27], to full-blown logic and constraint programming languages such as Prolog [28]. Recently there has been renewed interest in rule programming, reflected in both RuleML [29] and OMG Rule [30].

### 2.5 Mathematical Programming

Symbolic mathematical programming languages such as MATLAB [12], Maple [15] and Mathematica [16] describe and manipulate abstractions such as mathematical equations symbolically rather than numerically. That is, users of these

systems can find exact, closed form solutions to scientific and engineering problems, in exactly the same way that a human would solve them (except that these systems are much better than most human solvers!). Since solutions obtained symbolically are exact, they are, in some sense, "better" solutions than those obtained with numerical techniques. The scientists and engineers who use these systems experience a very easy learning curve. Just as important, symbolic programming systems typically include a number of useful end-user facilities (such as tools for publishing, graphing and charting, report-writing, etc.) that spares their users from any need to learn conventional programming languages such as C or Java. Matlab, for example, includes the capability to produce executable programs from the Matlab model.

## 2.6 Vector Programming Languages

Vector programming, as exemplified by APL [6] and its successor languages such as J and K, provides another example of the power of a strong specific solution. Mathematicians, scientists, engineers, and quantitative business users find it natural to describe systems using vectors and matrices, and have no difficulty transferring this knowledge to enable programming in a well-designed vector language. Vector languages illustrate two other interesting aspects of 3D approaches. First, the strong underlying model (vector math in this case) allows implementers to design both compact data representations, and very efficient algorithms for computing on that data. Kdb applications [32], for example, routinely process many thousands of transactions per second on huge amounts of data, easily outperforming standard relational databases. Second, because the programming abstraction is such a good fit to the problem domain, programs tend to be smaller when compared with, for example, a typical OO language like Java. Less code translates directly to higher productivity and lower maintenance costs.

## 2.7 Dynamic Object Languages

These languages include actor languages, prototype languages such as Self and most familiarly, Smalltalk and CLOS. They are characterized by a pure object model, where the objects rather than the values are strongly typed. Most derive their heritage from Simula 67, which we would assert is the first model based programming language, and arguably defined the beginning of model driven development. Unlike languages from the Pascal and C family, we found that Smalltalk had a very quick uptake by business users and engineers [9]. In particular, the simple keyword syntax and extensive generic class libraries were the major factors which users claim attracted them to Smalltalk.

The keyword syntax in particular provided domain programmers with a simple way to embed their own domain abstractions in Smalltalk, and acted effectively as a mechanism for domain specific syntactic extensions. Successful Smalltalk developers often had a lot of domain modeling experience, typically with a strong bias to using a simulation metaphor for development. The most sophisticated business developers used Smalltalk as a 5GL, by which we mean they implemented their own DOP languages on top of the Smalltalk base. They generally found this to be a more productive and rewarding programming experience than using more computer science oriented languages, which appeared to them to have a great deal of accidental complexity.

## 2.8 Functional Languages

This family of languages, which includes Lisp [26], Scheme, ML and Haskell, has a strong tradition based on functional programming. Functional languages, like dynamic object languages and vector languages, have for many years been used very successfully in challenging business and engineering problem domains [4] [7] [8].

They have also been used extensively for hosting domain specific programming languages, so much so that there was a common saying in the Lisp community that "no one actually programs in Lisp; they use Lisp to implement the language they really want and then program in that". At one point the use of functional languages was so significant that symbolic hardware was developed to support execution with a tagged architecture and assist in garbage collection. In recent years there has been resurgence in functional programming led by the Haskell and Caml communities.

As noted, Lisp in particular easily allows the definition of an embedded domain language. Functional programmers must smile when looking at the XML family of standards, which are following this tradition of embedding for descriptive languages. In the context of text and tag processing we should also note in passing the important contributions of the SNOBOL family of languages. These led in turn to the expressive and interesting streaming rule language, Omnimark [5]. The Omnimark language is far more readable and efficient than XSLT, which is popularly advocated for such tag transformations.

# 3. DOP Development Practices

During the past five years we have been interacting with different groups of DOP programmers using the languages discussed above. In particular we were interested in what motivated an actuary or financial analyst to learn a DOP language, especially when some, such as the APL family, are considered difficult to both learn and use by many software engineers. Further, we were interested in understanding their development processes, especially in comparison with the processes advocated for software engineering and business process modeling.

## 3.1 Living In My Data

All of the users of advanced programming languages confessed that they found learning the language difficult, although many also worked in pairs with another professional programmer who was an expert. They also freely admitted that they didn't have full comprehension of the entire language, nor did they have any deep appreciation for the semantic account of the language. A recurring theme was that their selected technology allowed them to live in their data and that unlike what they called "CS languages" they felt that they could not think ahead of their language, and none expressed a need for faster execution at development time. One actuary commented, "It was a pain to learn but until I had this capability I've always been hostage to using Excel and OLAP tools, typically depending on programmers and IT. Now I can live in huge raw data sets and I see things in that data that none of my colleagues can see using their conventional tools".

## 3.2 Domain Analysis Or Domain Programming?

For many years the software engineering community has argued the need for domain analysis [20] to identify the appropriate domain artifacts and abstractions prior to initiating a development activity, or selecting COTS components. Interestingly this is very similar to the approach used by DOP developers, who directly construct and exercise domain abstractions as part of their development process. The essential difference in the DOP approach is that the domain concepts are directly implemented and manipulated in code, rather than being further abstracted into UML and then rendered into an implementation via generators.

## 3.3 Agile Of Course

Many commented that "they really liked the Agile/XP approach" since in many ways it matched what they were already doing. Many teams practiced what we have come to call *ultimate pair programming,* where an expert developer and a domain expert shared a keyboard and mouse. This is the best of XP where the customer actually becomes a co-developer.

Typical development teams were very small, with two to five being a normal size. All development was iterative and incremental. Usually every new function/method was exercised at the time of creation, not deferred for later testing.

Because the languages were known to be more difficult to understand, literate programming was the accepted practice and naming conventions were considered very important. APL programmers pointed to the classic paper by the late Alan Perlis on APL idioms that can be seen as the first publication on patterns [2].

Those unfamiliar with objects asked about refactoring and smiled happily as they talked about how they constantly revised their code to improve it. Functional programmers often mentioned the paper by Hughes [3] that provided inspiration for them to polish and refine their programs.

# 4. Placing DOP in the 3D Context

3D covers a wide range of emerging technologies, including but not limited to Model-Driven Architecture, Product-Line Engineering, Aspect-Oriented Software Development, Generative Programming, Intentional Programming, and various attempts to generalize Design Patterns. What these share is the common goal of bringing solution technologies into better alignment with problem domains. With the exception of Intentional Programming, the majority of these techniques are related to component and object technology. They are designed for software engineers to build either application software, or tools to build such software. Generative Programming and Aspect Oriented Programming in particular have evolved as practical forms of meta-programming, with strong roots in computational reflection. The Patterns work has found application in organizations outside of OO development; however, these are really a distillation of best practices rather than executable artifacts.

## 4.1 The Typical 3D Process

The majority of 3D efforts are focused on developing generic modeling and implementation techniques, to support software development by formally trained computer professionals. Many require a graduate education in computer science as well as considerable practice to master. One only has to look at the complexity inherent in the OMG MDA stack, AOP languages, or Generative Programming to see why they are daunting for many software professionals, let alone engineering, and business professionals.

These approaches can be summarized as mapping domain concepts into abstract modeling concepts (often UML), then applying sophisticated tools and processes to generate code. They can work well in restricted domains but are difficult to apply in many others such as business intelligence and bioinformatics. Since the target platform is typically J2EE or .NET there is a strong emphasis on generation to translate domain representations into middleware execution artifacts. The inherent complexities of the process are obvious and the need for tools to implement them is acute**.** We can summarize the process as –

***Domain => UML => Program***

## 4.2 The DOP Process

In contrast, DOP seeks to address the problem by using a higher level programming language that has strong computational foundations to directly model the domain problem, or to host a domain specific language for that purpose. Rather than employ a separate modeling abstraction like UML, we model the problem domain directly in an executable programming language. We argue that for many application developers this is a much more natural approach –f

***Domain => Domain Language => Program***

U.S. Supreme Court Justice Potter Stewart once famously remarked that "I shall not today attempt further to define the kinds of material I understand to be embraced within that shorthand description; and perhaps I could never succeed in intelligibly doing so. *But I know it when I see it*"[22]. Our ideas about DOP are immature, and like Justice Stewart we find it difficult at this stage to crisply define exactly what is, and is not, DOP. But we know it when we see it, and we can draw on the lessons learned from end-user programming, and from our interactions with nascent DOP developers, to identify some of the features and benefits of DOP that distinguish it from other 3D approaches:

- language abstractions which map directly onto the problem domain
- interactive, incremental development style
- programming by example
- bias towards dynamically typed object languages
- use of agile development practices such as pair programming, refactoring and test first
- DOPs seem easier to learn; partial understanding is usually enough to do useful work
- smaller programs leading to increased productivity
- domain-driven implementation efficiencies

Certainly none of these features is unique to DOP, but taken together they begin to characterize DOP's place within the 3D universe.

# 5. Challenges and Opportunities in DOP Design and Implementation

## 5.1 Strong Specific Versus Strong Generic

It might seem intuitive that domain specific languages should be more attractive to users than potentially more complex generic domain oriented languages, which typically have a much higher learning threshold. However, in practice, users of DOPs express a strong preference for the ability to easily embed a DSL in a DOP. We need to understand a great deal more about the tradeoffs between DSLs and generic DOPs, computational models for embedding, and the impact of these on programmer productivity and program complexity.

## 5.2 Space and Speed Issues

As noted earlier, we have a few examples of strong specific domain knowledge allowing implementers to make assumptions and impose constraints which lead to speed and space efficiencies, e.g. with vector programming languages. Our intuition is that there are many more opportunities with DOPs for such domain-driven performance tuning.

In addition, most of DOP's antecedent technologies (e.g. functional and dynamic object languages) typically have their own portfolios of performance optimization techniques which have been developed over the last twenty or so years. However, many of these are only understood by their expert communities, and not well known to outsiders.

There may also be some unique performance opportunities obtained by optimizing at a very high level using function composition, substitution etc. Our conclusion is that it is likely that there are potential synergies, both between the domain-driven and language-driven approaches, and also between the various language communities.

## 5.3 Readability and Writeability

Our experience, and the experience of others, is that once coached over the threshold, business and engineering students feel empowered by DOP. However, terminology can present a significant barrier: lexical closures, continuations, higher order functions, range, domain, monads are just a few examples of terms which are not in the lexicon of most application developers and many software professionals. The challenge for DOP advocates is how to reduce these syntactic and vocabulary barriers that often come from the domains or technical disciplines in which the languages have evolved, which can create significant intellectual impedance.

End user tooling which generates DOP code, combined with teaching examples, which provide a gentle introduction to concepts for those who lack the formal background, may mitigate this. But at another level this problem can be seen as the dark side of the strong affinity that exists between DOPs and their associated foundation disciplines. Regardless, it is clear we need better ways to communicate the deeper meaning of domain specific terminology implicit in DOP languages to application developers.

## 5.4 Generation vs. Direct Execution

Most current DOP approaches have tended to favor direct execution over generative techniques. There is no doubt that the immediacy of the direction execution experience, the capability to interact with both program and data in real-time, has tremendous appeal to the application developer.

However, it is a reality of today's heterogeneous computing environment that many platforms are not easily accessible, perhaps because they are remote, or have limited interaction and presentation facilities (e.g. embedded systems). Deployment in such circumstances requires a capability to generate solutions based on some kind of program description or metadata.

In addition, generative techniques can offer opportunities for both performance improvement and reuse that are not available with direct execution. The history of OTI's efforts to deploy Smalltalk on embedded computers and mainframes offers some hints as to partial solutions [9][23], but this is a large open area, which requires further investigation.

# 6. Summary

In this paper we have sketched a vision for Domain Oriented Programming, a model driven approach to 3D. What sets DOP apart from other 3D alternatives is that it is aimed squarely at application developers, rather than computer professionals. It addresses problems that require deep application knowledge and the ability to build complex applications directly from domain models. We do not claim to have invented DOP, as much as to have rediscovered it in the cultures of various end-user programming communities. In a practical sense, DOP is alive and well, and is already being used in several niche application domains. But the application developers who currently employ DOP languages and techniques are not doing so with any explicit self-awareness. They do not, as yet, have a label to describe what they are doing, and names are important.

There is currently little or no effort devoted to studying best practices, categorizing and cataloguing techniques, developing new technology for embedding DSLs in DOPs, or designing new and better DOP languages. We would like to change this, by calling attention to a real success story that is hidden in plain view, one that should be legitimized and encouraged. DOP methods like ultimate pairs provide an ideal vehicle for non-software professionals to work in concert with talented developers.

The current communities are often isolated islands centered on specific applications, languages, or vendor products. We feel that there is a need for more interaction between the language communities, as well as a need to educate young professionals with respect to the power and limitations of various DOP approaches [31]. At a minimum we would expect to see such efforts improve best practices, and evolve existing languages or new hybrid languages such as Loops (objects, rules, functions and lists), or F-Script (Smalltalk and Arrays) [33]. Our hope is that by promoting this vision we can help to build a community around DOP concepts, and encourage research that furthers the state of the art**.**

We would like to close with a note of caution. The attempt to turn a good strong specific language into a universal language and "conquer the world" has taken many languages outside the problem space for which they are best suited. The language wars that follow often destroy the use of the languages in the very domains for which they worked best. DOP languages need to stay domain-oriented, and not fall prey to this temptation. After all,

every true craftsman knows the importance of using the right tool for the job.

DOP, while not a panacea, is a proven approach, which merits further definition, investigation and popularization. It has already enjoyed limited success and presents a clear alternative to current 3D approaches for application development.

## 7. ACKNOWLEDGMENTS

Our thanks to all of those talented application developers who continue to build great applications despite the limitations imposed on them by the implementation technology; to the designers of semantically clean high level languages; and finally to the reviewers for their constructive comments.

## REFERENCES

[1] Philip Cox and Trevor Smedley, End-User and Domain-Specific Programming (EUP) Symposium, 2002 IEEE Symposia on Human Centric Computing Languages and Environments (HCC'02).

[2] Alan J. Perlis and Spencer Rugaber, Programming with idioms in APL, International Conference on APL Proceedings, Pages: 232 - 235, 1979.

[3] J. Hughes, Why Functional Programming Matters, Computer Journal, 32, 2, pp 98-107, 1989.

[4] Simon Peyton Jones, Jean-Marc Eber and Julian Seward, Composing Contracts: An Adventure in Financial Engineering, ICFP 2000.

[5] Mark Baker, Internet Programming with OmniMark, 2000.

[6] APL 2002, Proceedings of the 2002 International Conference on APL: Array Processing Languages: Lore, Problems, and Applications, July 22-25, 2002, Madrid, Spain. ACM, 2002.

[7] Philip Wadler, Functional Programming in the Real World, http://www.research.avayalabs.com/user/wadler/realworld/.

[8] David B. Lamkins, Appendix A - Successful Lisp Applications, Successful Lisp: How to Understand and Use Common Lisp, http://www.psg.com/~dlamkins/sl/appendix-a.html.

[9] Dave Thomas, Ubiquitous Applications: Embedded Systems to Mainframe, Communications of the ACM Volume 38, Issue 10 (October 1995).

[10] Visual Language Research Bibliography http://cs.oregonstate.edu/~burnett/vpl.html.

[11]http://wwhttp://cs.oregonstate.edu/~burnett/vpl.html#V2A4w.usenix.org/publications/library/proceedings/dsl99/technical.html

[12] Steven T. Karris, Signals and Systems with MATLAB Applications, Orchard Publications, 2001.

[13] http://www.bricklin.com/visicalc.htm.

[14] http://www.pictorius.com/.

[15] http://www.maplesoft.com/products/.

[16] http://www.wolfram.com/.

[17] W. Piersol, Object Oriented Spreadsheets: The Analytic Spreadsheet Package, Proceedings of ACM OOPSLA, September 1986, 385-390.

[18] R. Jamal, L. Wenzel, The Applicability of the Visual Programming Language LabVIEW to Large Real-World Applications, 11th International IEEE Symposium on Visual Languages, September 05 - 09, 1995.

[19] DSL'97, First ACM SIGPLAN Workshop on Domain Specific Languages, January 18, 1997.

[20] Domain Engineering and Domain Analysis, http://www.sei.cmu.edu/str/descriptions/deda.html.

[21] P. Hudak, Building Domain-Specific Embedded Languages, ACM Computing Surveys, 28(4es): 196-198, December 1996.

[22] http://caselaw.lp.findlaw.com/cgi-bin/getcase.pl?court=US&vol=378&invol=184

[23] Kim Clohessy, Brian M. Barry, and Peter Tanner, "New Complexities in the Embedded World – The OTI Approach", Object-Oriented Real-Time Systems Workshop, ECOOP 97, Jyväskylä, Finland, June 9-13, 1997. Also in Lecture Notes in Computer Science 1357 Springer 1998, ISBN 3-540-64039-8, Pp. 472-481.

[24] Dave Thomas, Visual Application Development – Lessons from the IBM VisualAge Experience, Keynote IEEE Symposium on Visual Languages, 1997.

[25] Haruo Hosoya and Makoto Murata. Validation and boolean operations for attribute-element constraints. In *Programming Languages Technologies for XML (PLAN-X)*, pages 1-10, 2002.

[26] Richard P. Gabriel and Guy L. Steele, The Evolution of Lisp, ACM Conference on the History of Programming Languages II, for an unabridged version see http://www.dreamsongs.com/NewFiles/HOPL2-Uncut.pdf.

[27] CLIPS A Tool for Building Expert Systems, http://www.ghg.net/clips/CLIPS.html.

[28] ICLP'03, the Nineteenth International Conference on Logic Programming, http://www.tcs.tifr.res.in/~iclp03/.

[29] The Rule Markup Initiative, http://www.dfki.uni-kl.de/ruleml/.

[30] OMG Business Rules in Models RFI, http://www.omg.org/techprocess/meetings/schedule/Business_Rules_in_Models_RFI.html.

[31] Dave Thomas: "Computational Diversity, Practice and a Passion for Applications", in *Journal of Object Technology*, vol. 2, no. 3, May-June 2003, pp. 7-12, http://www.jot.fm/issues/issue_2003_05/column1.

[32] Kdb for DBAs, Kx Systems Inc., 2001.

[33] http://www.fscript.org/