Different ways of using Generative Programming to develop an application

Carine Courbis
INRIA Sophia-Antipolis - OASIS project
2004, route des Lucioles - BP 93
06902 Sophia-Antipolis cedex, France
Carine.Courbis@sophia.inria.fr

Didier Parigot
INRIA Sophia-Antipolis - OASIS project
2004, route des Lucioles - BP 93
06902 Sophia-Antipolis cedex, France
Didier.Parigot@sophia.inria.fr

ABSTRACT

With the emergence of the Internet and proliferation of new technologies, the design and programming of complex applications need to take into account standards and notions of code distribution, deployment and reuse. There is a need to change the programming methodologies to take into account these different facets.

This paper lays the foundations for a new way of programming based on generative programming that automatically integrates specific technologies and user specifications (abstract models). This idea was successfully used, at different levels - data representation, interactive environments, semantic treatments and the architecture, in the design and realisation of SmartTools, a software framework for domain-specific languages. In this way, the generated source code makes use of XML technologies for the data representation, object and bean technologies for the views and GUI, aspect-oriented programming and visitor design pattern for semantic treatments and components to obtain an open architecture and a deployable distributed application. This idea is very close to the MDA (Model-Driven Architecture) approach of the OMG consortium that advocates a platform-independent model that can be transformed into one or more platform-specific models.

The main results of this new way of programming is better software quality due to business logic and technology separation, more straightforward code, a rapid addition of new facets and a means that facilitates the portability of applications towards new technologies or platforms.

Keywords

generative programming, data model, separation of concerns, component, GUI, XML schema and DTD, aspect-oriented programming, visitor design pattern, MDA.

1. INTRODUCTION

During this last decade, there were many changes in computer science that have an influence upon the way an application must be developed. Four main reasons of these changes can be identified:

- The first one is the emergence of the Internet that implies applications being no more PC-enclosed but rather distributed ones. Thus, now, data communication between applications and users must be taken into account during the whole application life-cycle. One problem was to choose a well-adapted data exchange format. To solve this problem, the W3C (*World Wide Web Consortium*) has created the XML (*Extended Markup Language*) standard that is application and platform-independent.
- The second reason of these changes is the proliferation of new technologies that makes it difficult to choose the right and most evolution-prone one. For instance, to obtain a component-based application, a developer must choose between, at least, three component technologies (CCM - *CORBA Component Model*, EJB - *Entreprise Java Bean*, or the Web-Services).
- The third reason is the democratization (widespread) of the computer science. That means that the users may have, now, different knowledge, needs, visualization devices, and activity domains that should be considered when developing.
- The last reason is a business reason. Indeed, to be competitive, a company must quickly and cheaply adapt its software to new user needs and technologies.

In software engineering, new programming paradigms occur such as AOP (*Aspect-Oriented Programming*) [24], SOP (*Subject-Oriented Programming*) [17] , IP (*Intentional Programming*) [36], or component programming [37] because the oriented-object programming is not sufficient to handle clear designs and reusable developments of software. For example, concerns can be cross-cut between classes and there can be a mix between functional and none-functional code in a single class making the code difficult to maintain and error-prone. At the specification level, a strong evolution is done towards standards of the W3C for documents or of the

OMG (*Object Management Group*) for design methodologies such as UML (*Unified Modeling Language*) or the MDA (*Model-Driven Architecture*) approach [7, 16].

The goal of this paper is to explain a new way of programming to cope with all these changes using generative programming [10] from models at different levels. More precisely, our approach is close to the MDA approach. We claim that, to develop an open and adaptable application, one must consider three main points:

- the *data model* describing the application structure that should have an application-independent format to abstract away from technology-specific details;
- the different *concerns* that should be separated and modularized to help maintaining the code and to facilitate its reuse;
- the *component* description language that should be as close to the needs as possible to clearly show the provided and required services.

If the application is interactive, a fourth point, the *GUI* (*Graphical User Interface*) and its views, should be considered to avoid device-dependent GUI and views.

Each of these points will, respectively, be presented in the next sections, followed by the conclusion. To enforce and validate our ideas, we have developed a tool generator for DSLs (*Domain-Specific Languages*) [38], named SmartTools [1] [34], based on this new way of programming. The Figure 1 gives an overview of how the generative programming is used in SmartTools. The Figure 2 gives the GUI of the tool showing the four different models used. This paper is thus an experience report, illustrated with concrete examples and screen-shot taken from our tool. These examples are intentionally not technically detailed as it is not the aim of this paper.

2. DATA MODELS

In research works on programming language, there are many formalisms to declare data structures (types). These formalisms are bound to a underlying theory that has either associated systems (such as BNF - *Backus Naur Form* - for parser tools, or algebraic types for rewrite systems), or programming languages.

The most famous theory is certainly the notion of abstract syntax (very often mapped to a tree structure description) that contains the common concepts of those theories, among them the important meta-language concept. Since the advent of distributed programming, the notion of formalism independent from any programming language has occurred such as IDL (*Interface Description Language*) for CORBA (*Common Object Request Broker Architecture*). The motivation of such formalisms was to find a way to exchange structured data between distributed applications.

Since some years, the OMG (*Object Management Group*) and the W3C (*World Wide Web Consortium*) consortia play

[1] http://www-sop.inria.fr/oasis/SmartTools/

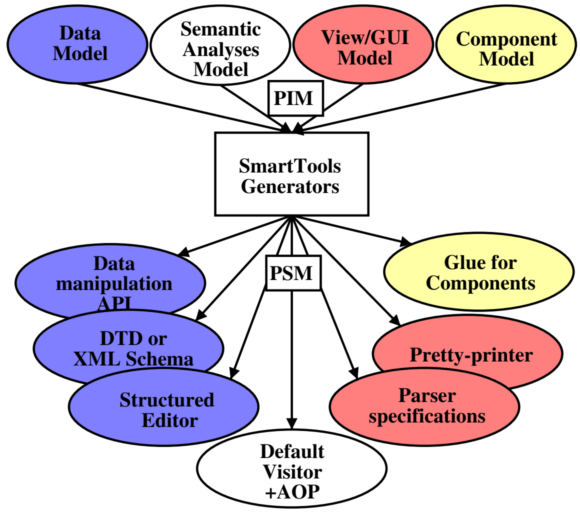

Figure 1: Overview of how generative programming, with a MDA approach, is used in SmartTools (PIM - *Platform Independent Models* - and PSM - *Platform Specific Models* -)

major roles in the data or model integration problematic with their standardization efforts. To fit with new needs, there are strong evolutions of these standard formalisms. For instance, to improve document data validity, the document meta-language (DTD - *Document Type Definition*) has been replaced by more complex and rich data type formalisms such as XML schema or RDFS (*Resource Description Framework Schema*).

In object modeling domain, the formalisms have also evolved. For example, the UML approach (object model) has evolved toward domain-specific models definition based on the MOF (*Meta-Object Facility*) meta-formalism [15]. It is important to notice that these two evolutions (XML schema and MOF) have introduced the main notion of meta-formalism. Additionally, these formalisms make the models independent from any programming language. This independence certainly explains the XML technologies success and their use in any domain.

In this way, it is possible to instrument the models with manipulation (transformation, browsing or selection) tools written in any programming language. This feature of programming language independence can, most probably, explained the success of the Web-Services with the SOAP (*Simple Object Access Protocol*) protocol. With the MOF and its new version[2] of exchange data format, XMI (*XML Metadata Interchange*), the bridge between the object modeling domain and the DSLs can be even more clearly identified as a XML schema associated with the meta-model (defined with the MOF) can be automatically generated.

But, it is important that these formalisms do not be considered only as a new exchange data format. Indeed, the applications built on the top of these models (e.g. the CASE - *Computer Aided Software Engineering* - tools) must also

[2] XMI 2.0 is based on XML Schema.

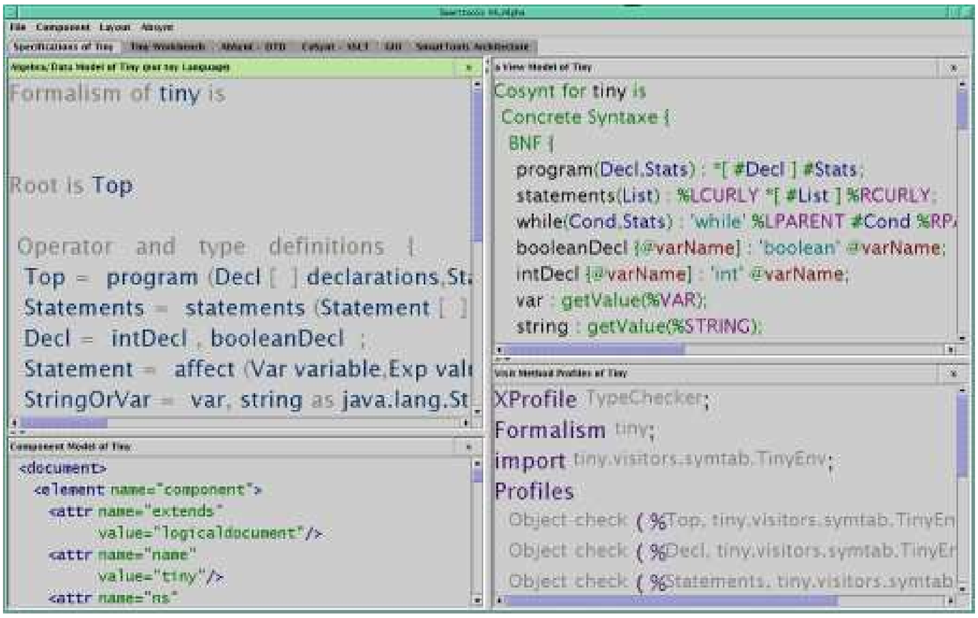

Figure 2: The SmartTools GUI showing the four models useful to generate tools for the tiny language. Anticlockwise, from the top left corner, we can see examples of a data structure declaration, a component description, the visit method profile description, and a pretty-printer (and its associated parser) description

be able to internally manipulated the data according to the models. More precisely, an application must be able to answer to an object addressing request (formulated with XPath for an XML schema model).

The databases have also evolved from relational databases toward object databases [4] and now toward XML databases [1]. Now, this database domain manipulates mechanismes comparable to those offered by programming languages, such as a query language (XQuery [8]) comparable to the pattern-matching (XPath) and semi-structured databases comparable to tree structures (XML tree).

This idea to have a meta-language independent from any programming language and technology was already present for the construction of the Cosy compiler [3]. Indeed, the different teams involved in this European project (named Compare) needed to exchange data between their tools. Thus the first step of this project was to define this meta-language to describe the different data structures (abstract syntax trees, control-flow graphs, intermediate representations). Many concepts (extension, inheritance, importation) of this meta-language are very similar to those of the XML schema.

In SmartTools, support of our research work on generative programming, we have also defined our own abstract data model, close to our needs and independent from any programming language. This model helps defining the abstract syntax of programming or domain-specific languages, the cornerstone for all the generated tools. From this data model, SmartTools can generate, as shown by the Figure 3, the followings:

- an API to help manipulating abstract syntax trees (for instance, writing semantic analyses);
- an equivalent DTD or XML Schema to help designers creating new DSLs;
- an editor guided by the syntax to facilitate document or program edition.

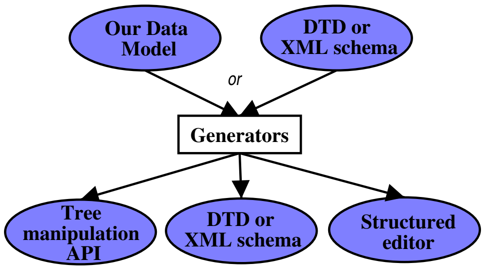

Figure 3: Generated tools from the data model

To avoid designing and implementing another propriatory tree manipulation API, we have chosen the DOM (*Document Object Model*) API standard as tree kernel. In this way, the SmartTools-specific generic code for manipulating trees is minimal thus easy to maintain, and benefits from any new service and bug fixing when this standard and its different implementations evolve. Thus our tree implementation is open, prone to evolution and can benefit from any DOM-compliant tool or service. For example, all the trees manipulated in SmartTools can be serialized in XML, transformed with XSLT, or addressed with XPath for free as these services are offered by the DOM API.

But the DOM standard does not fulfill all our needs as the manipulated trees are not strictly-typed (a DOM tree has only homogeneous nodes) thus difficult to semantically analyze. To manipulate strictly-typed trees, we generate a language-specific API (Java classes) above the DOM API; the type names of the nodes and the accessor names are provided by the model.

To establish a bridge between the programming language domain and the document domain, the data model can be transformed into an equivalent DTD or XML schema, and these document description languages accepted as data models (inputs) in our tool (see Figure 4).

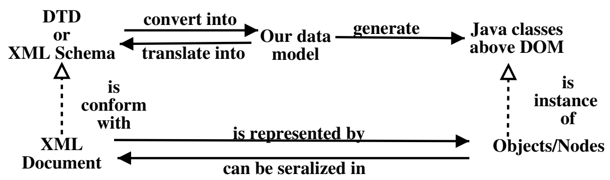

Figure 4: Bridge between languages and documents

3. SEPARATION OF CONCERN

To make more adaptable the applications, new programming paradigms appear such as AOP [23], SOP [17] or the generative programming [10]. In a certain way, the GOF book on design patterns [13] was already concerned about this problematic and offered patterns to design applications with better genericity and flexibility. One of the most famous design patterns is the visitor that separates the data structures (a hierarchy of classes) and the associated treatments. In this way, the treatments are modular (one class) making modifications and extension easier. The adaptable programming [26, 33], that provides a better flexibility towards any data structure change, has extended this design pattern. More precisely, a traversal description language (sort of extended pattern matching model) makes possible to abstract away from the underlying data structure of the application.

The visitor design pattern has been the basis of many other research works [32] and is close to the notions of multimethods [2, 28] or generic functions for functional programming. Other approaches towards a more generic programming can be mentioned such as C++ templates, generic libraries [29], generic Java types [40, 31] and the mixins programming [12]. All these approaches [30] either need source code transformations, the reflexivity, or the higher order concept to work out.

All the mechanisms to obtain more generic and modular programs are not sufficient. That explains the apparition of AOP [24]. The main objective of this new paradigm, like the visitor design pattern, is to split up the application code into entities (class, module) associated with a concern for a better application modularity.

To implement an AOP, it exists mainly three approaches: by source code transformation, by using the reflexivity (MOP - *Meta-Access Protocol*) or by modifying the language interpretor (such as JVM - *Java Virtual Machine*). That arouses many research works that try either to define new extensions (more expressive pointcut language or more powerful instructions) or to give a clear operational semantics.

In SmartTools, we have chosen to introduce an AOP strictly dedicated to our program pattern used to write semantic analyses (e.g. a type-checker or an evaluator), part of the business code of the component. Instead of using static source code transformations or the reflexivity, our program-pattern-specific AOP is implemented thanks to a step of code generation. Indeed, we have created a generator that instruments our program pattern based on the visitor design pattern. From the data model and pieces of meta-information about the semantic analysis (*visit* method signatures and the traversal), the generator can produce a default visitor that only visit the nodes including into the traversal (see Figure 5). To obtain a semantic analysis, the programmer has only to extend, by inheritance, this default visitor and override some of the *visit* methods to specify the treatment. To introduce an AOP into our semantic analyses, we have only expanded the generator in order that it produces the specific aspect-plugging code embedded into the default visitor. In this way, a analysis can be extended, not only, by inheritance, but also, dynamically, with aspects. The main advantage of a such approach is to obtain an AOP close to the needs, easy to use (clear operational semantics), and with a easy implementation that can quickly evolve with new application-specific needs.

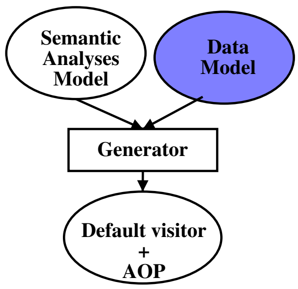

Figure 5: A default visitor with aspect plugging is generated from the abstract syntax of the language (data model) and the *visit* method configuration

With this approach, our program pattern is better splitted up into three distinct parts:

- the data model,
- the recursive traversal,
- and the semantic actions (the treatment).

This decomposition makes the transformation of the result application easier and makes possible to project the semantic actions onto another data model. Indeed, in the MDA approach context, it seems to be possible to transpose a transformation (written with semantic actions) from a PIM (*Platform Independent Models*) to a PSM (*Platform Specific Models*)towards another PIM if it exists a data model mapping these two PIMs (see Figure 6). For example, if the mapping of our component model into IDL is specified and if the generator that produces the container from our component model (see next section) is written with semantic actions, it is possible to obtain, for free, a generator that produces the container from IDL (see Figure 7). This transposition may be realized only by modifying the aspect plugging expressions. This idea comes from research works on program transformation stemming from the attribute grammars (with the structural composition concept [9, 11, 14]) and the functional programming (with the deforestation concept [39] or fold and unfold operators [20]).

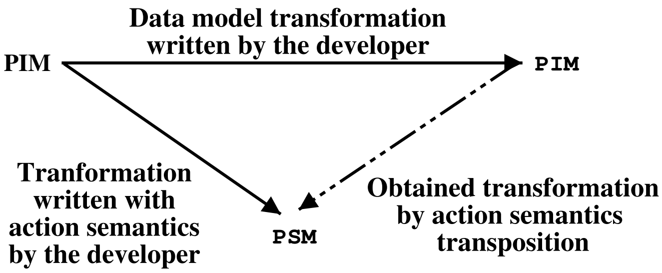

Figure 6:

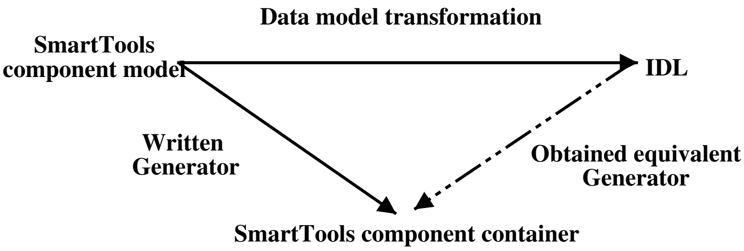

Figure 7:

4. COMPONENTS

The programming evolution has also deeply changed the modularity concept: ADA or Java packages, generic libraries in imperative programming, multi-inheritance and contract approach in Eiffel [27], or the module notion in functional programming [25]. These different approaches offer powerful mechanisms of genericity such as multi-inheritance, contract approach, or higher order. But these mechanisms are rather complex to use and do not fit in well with distributed application needs.

To cope with these new needs, many component technologies have been proposed such as COM and DCOM for Microsoft, CCM for the OMG, and EJB for Sun. More recently, the Web-Services technology has appeared with the possibility to list the component services in catalogs (UDDI - *Universal Description, Discover and Integration*).

The three main challenges in component technologies are the followings:

- to extend the classical method call to take in account the execution environment (three-tier architecture, the Internet, The message service, database access) without modifying the business code;
- to extend the classical interface notion to be able to discover the provided and required services (such as introspection in Java Bean) and to dynamically adapt the interface (such as the multi-interface notion in CORBA);
- to add meta-information on a component to manage the deployment, the security policy, etc.

These different mechanisms must be transparent towards the business code of the components. That corresponds to a kind of separation of concerns to avoid mixing the functional and the none-functional code. The OMG has proposed the MDA approach based on model transformations to get a better evolution of complex software applications towards component technologies [41]. That explains the research effort done to define a new generic component language and the link with the AOP and the model transformations.

As SmartTools generates and imports tools, it was vital to have a component architecture for its evolution and to make interconnections with environments or tools easier. Having a component architecture in our case (meta-tool) is also useful to be able to build an application with only the required components.

Our first step was to define an abstract component model i.e. independent from any component technology. The advantage of having our own component model is to clearly identify the needs of SmartTools; without this model, its needs would have been hidden under a none-application-specific component format (such as IDL). From this component model, a generator can automatically produce the none-functional code, i.e. the container that hides all the communication and interconnection mechanisms. For example, the broadcast mechanism used by a logical component to update its associated view components is totally transparent for the programmer. Additionally, it is very easy to adapt the architecture to introduce a new communication mechanism.

The second step was to define a set of transformations (projections) from our model towards well-known component technologies such as web-services, CCM, or EJB (see Figure 8) to make the exportation of the produced tools easier.

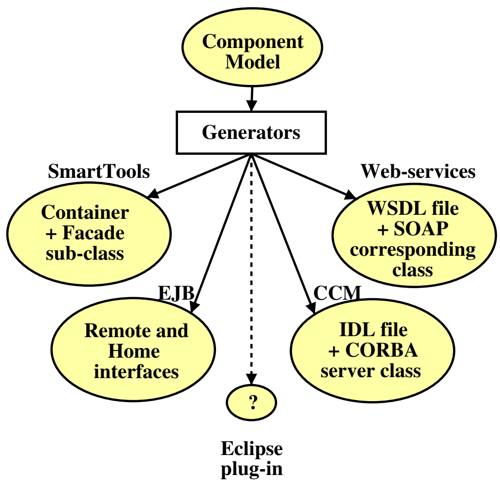

Figure 8: Component model transformations

From our projection experience, we can say that these three component technologies (Web-Services, CCM, EJB) would have not fitted in with our needs of connections and component model.

Indeed, our connection process is much more flexible and dynamic than those offered by these technologies mainly dedicated to client/server architectures or Web servers. In SmartTools, component interconnections are done dynamically when requested and use a kind of pattern-matching on the names of services provided or required by the components to bind the connectors.

In spite of this feature, it is possible to exchange complex information between two components such as sub-trees or path information (XPath) for the views and the associated logical document. The implementation of our model was rather easy.

Our experience shows that there are many advantages to create an abstract component model that fits in with the application needs, rather than using a none-specific model. With this MDA approach (based on generative programming), we were able to obtain implementations in different technologies. In this way, our tools can much more evolve and are adaptable.

The Figure 9 shows the dynamic architecture of our demonstration application from which the GUI of the Figures 2 and 12 are taken. This Figure contains all the required component types, their instances, and the interconnections between theses instances.

5. GRAPHICAL USER INTERFACES

The graphical interfaces that make applications interactive must also be adaptable to these evolutions. Two main challenges, when designing a graphical interface, should be kept in mind: the interface might be executed on different visualization devices and be accessible through a Web interface. More these device variations and the possibility to transport the interface on the network are taken into account, better the application.

Additionally, the proliferation of new models or domain-specific languages requires the ability to quickly design and implement interfaces (or pretty-printers) specific to one model or domain. In this context, the visual programming can be very useful to realize programming environments dedicated to none-complex description languages (DSLs).

The goal of our tool, is to help developing new tools or programming environments, especially for this kind of languages (DSLs). The design of the tool took in consideration the specificities of these languages: i) they have their own data description language that should be accepted as input of our tool, and ii) the designers of such languages may not have a deep knowledge in computer science. It was thus a necessity to establish a bridge with the document domain, and to provide tools easy to use and built on well-known techniques. In this way, with a few effort, a programming environment dedicated to a DSL can be quickly implemented, having one or more specific-business displays of the documents. These different displays, more user-friendly and readable than the XML format (tags embedded), are obtained through a sequence of model transformations or refinements (see Figure 10). The Figure 11 outlines the approach and shows the intensive use of standard tools or specifications such as XSLT (*Extensible Stylesheet Language Transformation*) for document transformation, CSS (*Cascading Style Sheets*) for style information, the Swing API for the graphical layout, and BML (*Bean Markup Language*) for the serialization of the graphical views. As the view model is based on a sequence of model transformations, it is possible to generate, not only the graphical view, but also the associated parser (for none complex concrete syntax). By default, there are also generic displays (none domain-specific) available to show any tree regardless of its language membership.

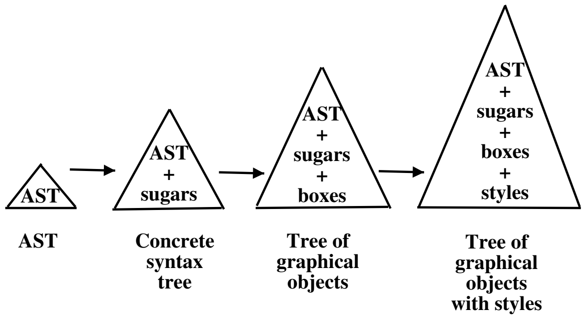

Figure 10: Specialization/refinement by successive model transformations to obtain a graphical view

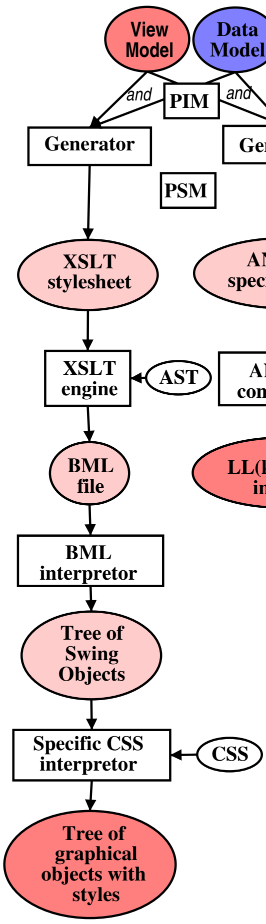

Figure 11: Implementation chain to obtain a graphical view or a parser

To be able to export our graphical views on the network,

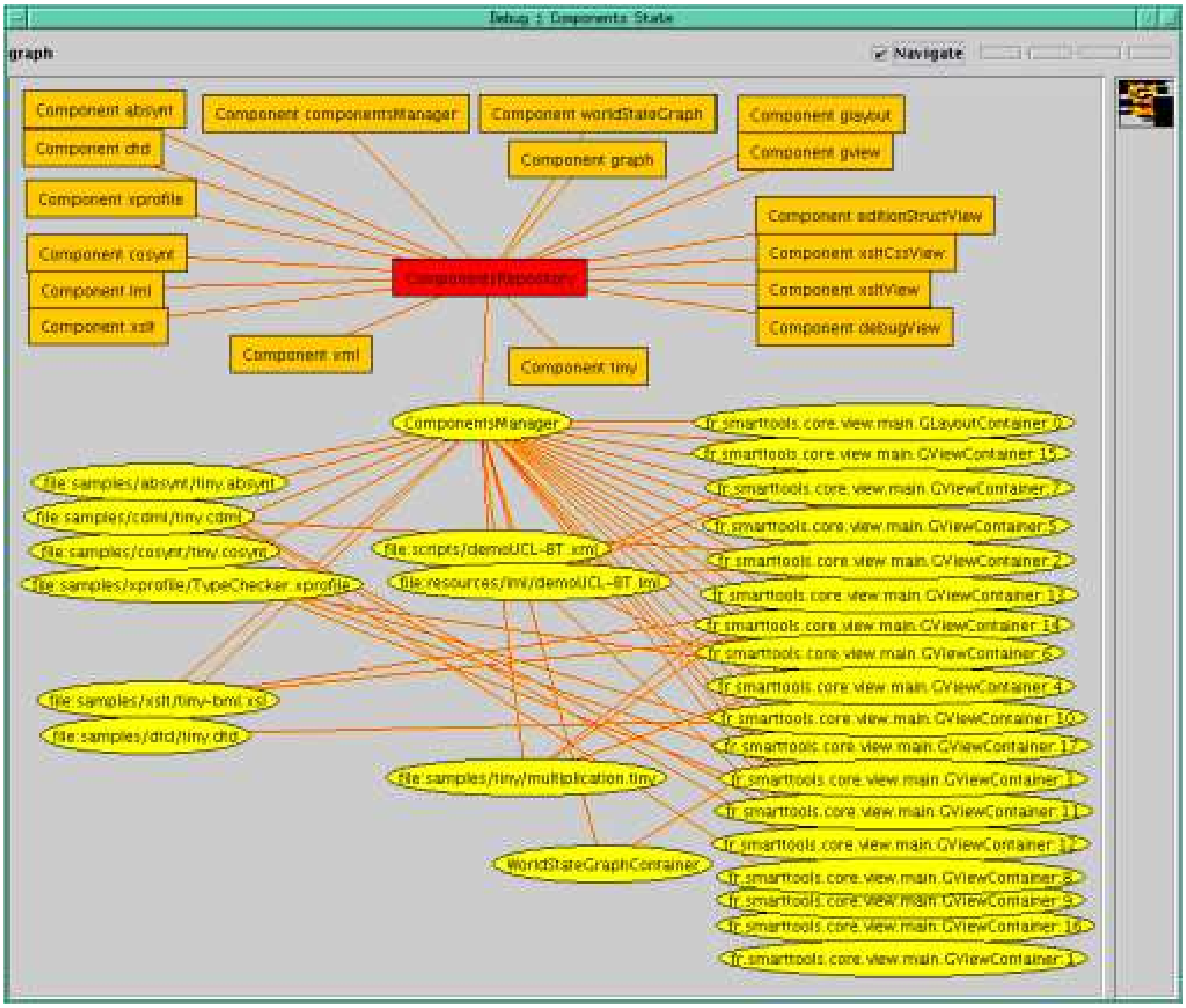

Figure 9: **The dynamic architecture of our demonstration application with the required component types, their instances, and the interconnections between these instances**

we have chosen the BML format [19] that describes all the graphical objects contained in a view to be created. The second advantage of this format is that the effective object creation (Swing objects) takes place on the view components (clients) not on the logical document component (server). The logical document component only provides the serialization file of graphical objects to the view component. . This latter only needs to incorporate a BML interpretor to create the graphical objects of the view. With this approach, it is easy to export SmartTools views into a Web browser. The logical document component and the associated view components are linked together. Any change on the document is automatically broadcasted to the views.

This model transformation, kind of "design pattern", to obtain graphical views was reused for the GUI (see Figure 2 for a screen-shot of our tool). Indeed a GUI can be considered as a tree of graphical objects (windows, tabs, panels, views, menu, etc.). All the AST manipulation methods (insert a node, remove a node, etc.) and the implementation to obtain a view can be reused. In this way, the GUI is only a particular view of this tree (see Figure 12 that shows three different views of the same GUI description document). We have defined a simple GUI-specific language useful to configure the GUI according to the applications.

In conclusion, the design of all our graphical tools uses the same "design pattern" (model transformations) that provides, on the one hand, an independence from visualization devices and, on the other hand, the ability to reconfigure the interfaces by modifying either the BML interpretor or the transformation files (the XSLT stylesheets).

6. CONCLUSION

Figure 12: This picture shows three different views of the same document (the GUI description): the generic XML display on the right, the generic hierarchical tree display on the left, and the GUI itself.

With the development of our tool (SmartTools), we have validated a new approach of software development mainly based on transformation or generation from programming-language-independent and domain-specific models. Thanks to generative programming, we can integrate new programming paradigms and technologies from the models into the target implementation programming language. Although only the Java programming language was used as implementation language for the SmartTools development, a large part of our approach can be transposed into another programming language. It exists many systems [5, 6, 18, 21, 22, 26, 35] comparable to SmartTools but they are not strongly based on this generative programming approach, used at different levels.

For the data model, the transposition should be easy (only the generators should be changed) as we claim that the data structures should not be defined with a programming language but rather with a high-level formalism independent from any programming language (such as UML or XML Schema). Thanks to its simplicity (due to its specialization) and to the code generation, our AOP approach should also be easily transposed into another type of application (e.g. in the component generator or in the container) or another implementation language.

We think that, the design patterns should be instrumented by generation tools to be usable, user-friendly, and dedicated to the application needs (less complex to use). In this perspective, all our generators can be considered as these generation tools that instrument the design patterns used in SmartTools. For example, the way we produce the graphical tools can be considered as a particular "design pattern" as it is mainly based on model transformations.

The main advantage of this design approach is to make applications easy to adapt and evolve only by updating the generators associated with each model (data structure, AOP, component programming, or GUI). These different generators provide the design methodologies that are parameterized through those models strictly restricted to the domain.

The Figure 13 outlines the approach to follow when developing an application and SmartTools instruments it. The interest of our research prototype, SmartTools, is to validate this new programming approach for different domains in a homogeneous way. Thanks to SmartTools, we can anticipate what will be the future evolutions in the programming languages and we can get convinced by the interest of separating the concerns through different models.

As Krzysztof Czarnecki and Ulrich W. Eisenecker compare the generative programming system to a factory of a particular domain [10], SmartTools can be considered as a software tool factory for any application which has a underlying data model. SmartTools uses itself this factory to be developed and this bootstrap is applied at multi-levels.

Acknowledgements

This project is supported in part by the W3C consortiun with the QUESTION-HOW IST project (see http://www.w3.org/2001/qh/ for more information). We would also like to thank Pascal Degenne and Alexandre Fau for their software development efforts in SmartTools.

7. REFERENCES

[1] S. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[2] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay.

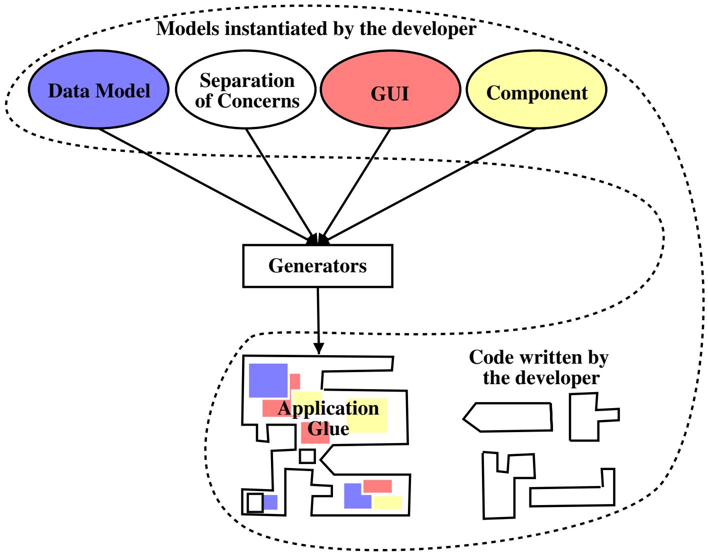

Figure 13: Model instances + generated application glue + business code = an application

Static Type Checking of Multi-Methods. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 113–128, Nov. 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[3] M. Alt, U. Abmann, and H. van Someren. Cosy compiler phase embedding with the CoSy compiler model. *Lecture Notes in Computer Science*, 786:278–??, 1994.

[4] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 223–240, Kyoto, Japan, Dec. 1989. Also appears in [O2-Book].

[5] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 143–153. IEEE Computer Society Press, 1998.

[6] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the System. *SIGSOFT Software Eng. Notes*, 13(5):14–24, November 1988. ftp://ftp-sophia.inria.fr/pub/centaur/papers/sde3.ps.

[7] J. Bézivin. From Object Composition to Model Transformation with MDA. In *TOOLS USA*, Santa-Barbara, August 2001. IEEE TOOLS-39. http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda/TOOLS.USA.pdf.

[8] W. W. W. Consortium. XQuery: the W3C query language for XML – W3C working draft. Available at http://www.w3.org/TR/xquery/, 2001.

[9] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Declarative program transformation: a deforestation case-study. In G. Nadathur, editor, *Principles and Practice of Declarative Programming PPDP'99*, volume 1702 of *Lect. Notes in Comp. Sci.*, pages 353–369, Paris, France, Oct. 1999. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Correnson99b.

[10] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications.* Addison-Wesley, June 2000. ISBN 0201309777 chapter Aspect-Oriented Decomposition and Composition http://www-ia.tu-ilmenau.de/~czarn/aop/.

[11] R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *19th ACM Symp. on Principles of Programming Languages*, pages 223–234, Albuquerque, NM, Jan. 1992. ACM press.

[12] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium, San Diego, California, 19–21 January 1998*, pages 171–183, New York, NY, USA, 1998. ACM Press.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995. ISBN 0-201-63361-2-(3).

[14] R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.

[15] O. M. Group. Meta Object Facility (MOF) specification (version 1.3). Technical report, Object Management Group, Mar. 2000.

[16] O. S. S. Group and R. Soley. Model-Driven Architecture. Technical report, OMG, November 2000. ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf.

[17] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In A. Paepcke, editor, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428. ACM Press, Oct. 1993.

[18] G. Hedin and E. Magnusson. JastAdd—a Java-based system for implementing front ends. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science, LDTA'01 First Workshop on Language Descriptions, Tools and Application, ETAPS'2001*, volume 44, Genova, Italy, April 2001. Elsevier Science Publishers. http://www.cs.lth.se/home/Gorel_Hedin/LDTA/RefJava.Submitted.pdf.

[19] IBM. Bean Markup Language. http://www.alphaworks.ibm.com/formula/bml.

[20] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lect. Notes in Comp. Sci.*, pages 68–114. Springer-Verlag, 1996. http://www.cs.chalmers.se/~patrikj/poly/afp96/notes.ps.gz.

[21] M. Jourdan, D. Parigot, C. Julié, O. Durin, and C. Le Bellec. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. In *Conf. on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990. Published as *ACM SIGPLAN Notices*, 25(6) ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/sgpln90-t.ps.gz.

[22] U. Kastens, P. Pfahler, and M. Jung. The Eli system. In K. Koskimies, editor, *Compiler Construction CC'98*, volume 1383 of *Lect. Notes in Comp. Sci.*, portugal, Apr. 1998. Springer-Verlag. Tool demonstration.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. *Lecture Notes in Computer Science*, 1241:220–??, 1997.

[24] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997. http://aspectj.org/documentation/papersAndSlides/ECOOP1997-AOP.pdf

[25] X. Leroy. Manifest types, modules, and separate compilation. In *POPL'94*, 1984.

[26] K. J. Lieberherr and D. Orleans. Preventive Program Maintenance in Demeter/Java. In *Proceedings of the 19th International Conference on Software Engineering*, pages 604–605. ACM Press, May 1997. http://www.ccs.neu.edu/research/demeter/biblio/demjava.html.

[27] D. Mandrioli and B. Meyer, editors. *Advances in Object-Oriented Software Enineering*. Prentice Hall, New York, 1992.

[28] T. Millstein and C. Chambers. Modular statically typed multimethods. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 279–303, Lisbon, Portugal, June 1999. Springer-Verlag. ftp://ftp.cs.washington.edu/homes/chambers/ecoop99.ps.gz.

[29] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software Practice and Experience*, 24(7):623–642, July 1994.

[30] M. Odersky. Objects + views = components? *Lecture Notes in Computer Science*, 1912:50–??, 2000. http://lamp.epfl.ch/~odersky/papers/asm2000.html.

[31] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 Jan. 1997.

[32] J. Palsberg and C. B. Jay. The Essence of the Visitor Pattern. In *COMPSAC'98, 22nd IEEE International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, Auguste 1998. http://www.cs.purdue.edu/homes/palsberg/paper/compsac98.ps.gz.

[33] J. Palsberg, B. Patt-Shamir, and K. Lieberherr. A New Approach to Compiling Adaptive Programs. In H. R. Nielson, editor, *European Symposium on Programming*, pages 280–295, Linkoping, Sweden, 1996. Springer Verlag. ftp://ftp.ccs.neu.edu/pub/people/lieber/compile-adapt-2.ps.

[34] D. Parigot, C. Courbis, P. Degenne, A. Fau, C. Pasquier, J. Fillon, C. Help, and I. Attali. Aspect and XML-oriented Semantic Framework Generator: SmartTools. In *ETAPS'2002, LDTA workshop*, Grenoble, France, April 2002. Electronic Notes in Theoretical Computer Science (ENTCS). ftp://ftp-sop.inria.fr/oasis/personnel/Carine.Courbis/smartldta02.pdf.

[35] T. Reps and T. Teitelbaum. The Synthesizer Generator. In *ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, pages 42–48. ACM press, Pittsburgh, PA, April 1984. Joint issue with Software Eng. Notes 9, 3.Published as ACM SIGPLAN Notices, volume 19, number 5.

[36] C. Simonyi. The death of programming languages, the birth of intentional programming. Technical report, Microsoft, Inc., Sept. 1995. http://citeseer.nj.nec.com/simonyi95death.html.

[37] C. Szyperski, J. Bosch, and W. Weck. Component-oriented programming. *Lecture Notes in Computer Science*, 1743:184–??, 1999.

[38] A. van Deursen and P. Klint. Little languages: Little maintenance ? *Journal of Software Maintenance*, 1998. http://www.cwi.nl/~arie/papers/domain.pdf.

[39] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In H. Ganzinger, editor, *European Symposium on Programming (ESOP '88)*, volume 300 of *Lect. Notes in Comp. Sci.*, pages 344–358, Nancy, Mar. 1988. Springer-Verlag.

[40] P. Wadler. GJ: A Generic Java. *Dr. Dobb's Journal of Software Tools*, 25(2):23–26, 28, Feb. 2000.

[41] T. Ziadi, B. Traverson, and J.-M. Jézéquel. From a UML Platform Independent Component Model to Platform Specific Component Models. In *International workshop in Software Model Engineering (WiSME02) at UML2002*, Dresden (Germany), Sept. 2002. http://www.metamodel.com/wisme-2002/papers/ziadi.pdf.