XML technologies in the SmartTools Software Factory [*]

Carine Courbis
University College London
Adastral Park - Martlesham
IP5 3RE - UK
Carine.Courbis@bt.com

Didier Parigot
INRIA Sophia-Antipolis
2004, route des Dolines - BP 93
F-06902 Sophia-Antipolis Cedex - France
Didier.Parigot@sophia.inria.fr

ABSTRACT

Because of the Internet and the associated proliferation of component and distributive technologies, the way of designing and implementing complex applications has to be modified to integrate standards and code distribution. To cope with these changes, applications need to be more open, flexible and capable of evolving.

The main goal of this paper is to describe how XML technologies are used at different levels, in an application named SMARTTOOLS, a software factory for DSLs. XML technologies were, not only used for its implementation (bootstrap), but also in the generated DSL software.

The innovative part of our work was combining together three approaches: the MDA (Model-Driven Architecture) approach, the Generative Programming, and the XML technologies. The main results are *i)* to benefit from existing tools and their evolution for the XML technologies, *ii)* to have an open application (and generated tools) without proprietary techniques, and *iii)* to build software of better quality due to business models and technology separation, that is easy to adapt and modify.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

Keywords

Graphical User Interface, Visual Languages, Domain-Specific Language, XML Technologies, Model-Driven Architecture, Generative Programming, Component-Based Software Engineering.

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 are no longer stand-alone 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 interoperable data exchange format. To solve this problem, the W3C (*World Wide Web Consortium*) created the XML (*Extended Markup Language*) standard that is application and platform-independent.

- The second reason is the proliferation of new technologies that makes it difficult to choose the right and most capable of evolving one. For instance, to obtain a component-based application, a developer must choose between, at least, three component technologies: CCM (*CORBA Component Model*), EJB (*Enterprise Java Bean*), or Web Services.

- The third reason is the democratization (widespread) of computer science. That means that 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.

To cope with all these changes, the way of designing and implementing complex applications has to be replaced. The applications need to be more open, flexible, and capable of evolving. The goal of this paper, in this context, is to explained how XML technologies can play a major role, at different levels, in a new way of programming from models (in our case, DSLs - *Domain-Specific Language*) in collaboration with Generative Programming [3]. Our approach - transforming (generating) from a programming-language free and platform-independent DSL (PIM - *Platform-Independent Model*) to a specific language (PSM - *Platform-Specific Model*) - is very close to the MDA (*Model-Driven Architecture*) approach [7, 1] of the OMG (*Object Management Group*).

We claim that, to develop an open and adaptable application, the four main points to consider are the followings:

- 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 maintain the code and to facilitate its reuse;

- The *component* description language that should be as close to the application needs as possible to clearly show the provided and required services.

- If the application is interactive, the *GUI* (*Graphical User Interface*) and its views that should be device-independent.

To enforce and validate our ideas, we have developed a software factory [11, 2], named SMARTTOOLS [1] [9], based on this new way

[*]This project is supported in part by the W3C consortium 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.

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

of programming. For each DSL, this factory can produce different tools or specifications (see Figure 1):

- a DTD,
- an XML Schema,
- a default tree walker to semantically analyse documents that is easy to extend by inheritance or by plugging additional code (aspects - AOP *Aspect-Oriented Programming*),
- several parsers and associated pretty-printers (i.e. engines to transform abstract syntax trees into a readable form) to provide the DSL with a more readable concrete syntax than XML,
- and finally the glue to pack all these tools in a component ready to be exported into another application.

As the factory is mainly bootstrapped (uses itself to generate tools for its own DSLs), its generative techniques have been experimented. This application, SMARTTOOLS, demonstrates how XML schema, DTD, DOM API, BML, XSLT, XPath, CSS, WSDL, and their associated tools (mainly from the Apache Jakarta project[2]) can be integrated (see Figure 2) and orchestrated; making the application more open and easy to maintain.

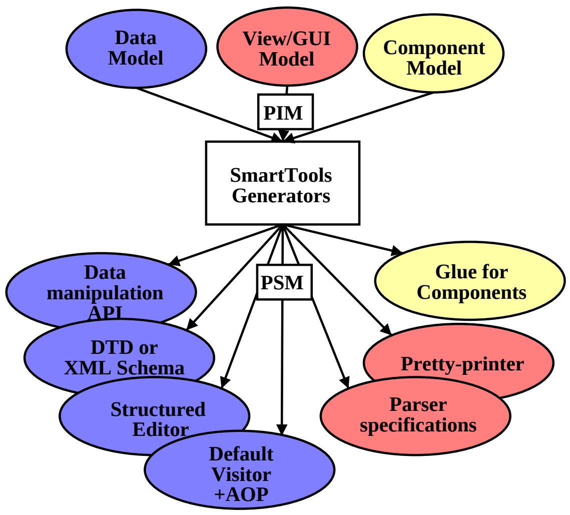


Figure 1: Overview of how Generative Programming, with a MDA approach, is used in SMARTTOOLS

Data	models	DTD, XML schema
	transformation	DOM API
	selection	XPath
View & GUI	models	own XML languages, CSS
	transformation	XSLT, XPath
	representation	BML
Component	models	own XML languages, WSDL
	deployment	own XML language

Figure 2: XML technologies in SMARTTOOLS

This paper is divided in three parts, one per model - data, view/GUI, and component - describing how XML technologies, Generative Programming, and the MDA approach can be combined together to develop an application. Finally, to conclude, we summarize all the advantages we found using XML technologies.

[2] http://jakarta.apache.org/

2. DATA MODELS

Since some years, the OMG and the W3C consortia have played major roles in the data or model integration problems with their standardization efforts. To fit with new needs, there are strong evolutions of the standard formalisms. For instance, to improve document data validity, the DTD (*Document Type Definition*), the document meta-language, has been replaced by more complex and rich data type formalisms such as XML schema or RDFS (*Resource Description Framework Schema*). With these meta-formalisms, the data (document) model is independent from any programming language. This independence has contributed to the widespread adoption of XML technologies (e.g. Web Services with SOAP - *Simple Object Access Protocol*) in any domain.

But, it is important that these formalisms are not considered only as a exchange data format. Indeed, applications built on top of these models (e.g. the CASE - *Computer Aided Software Engineering* - tools) must also 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 goal of SMARTTOOLS (support of our research work on Generative (Programming) is to help develop new tools or programming environments, especially for DSLs. The design of the tool has taken into account 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 between the programming language domain and the document domain, and to provide tools that are easy to use and built on well-known techniques.

We have first defined our own abstract data model, close to our needs and independent from any programming language. This model (DSL) helps define the abstract syntax of languages (i.e. the document structure), the cornerstone for all the generated tools. From this data model (a PIM), SMARTTOOLS can generate, as shown by Figure 3, the following:

- an API to help manipulate abstract syntax trees (for instance, writing semantic analyses);
- an equivalent DTD or XML Schema to help designers create new DSLs;
- an editor guided by the syntax to facilitate document or program edition;
- a default tree walker based on the visitor design pattern to ease the implementation of semantic analyses on documents.

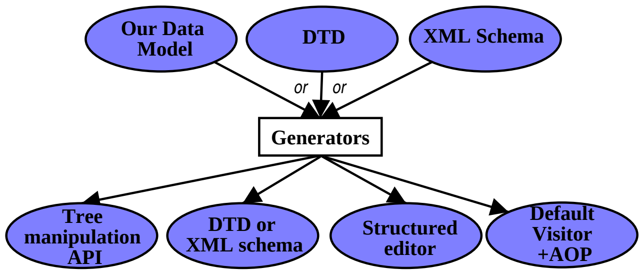


Figure 3: Generated tools from the data model

To broaden the scope of SMARTTOOLS and to benefit from existing tools, a bridge between the programming language domain and the document domain (see Figure 4) needed to be established, in both directions. In this way, the tool accepts equally, as data model (input), our proprietary data definition language, a DTD, or an XML Schema. It can also translate it into any equivalent data model (DTD, XML schema, or ours). The DSL designers are not compelled to learn our proprietary data model.

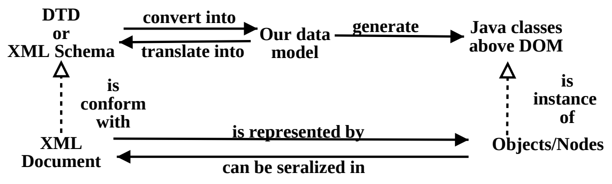


Figure 4: Bridge between languages and documents

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 or bug fixes when this standard and its different implementations evolve. Thus our tree implementation is open, capable of evolving, 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.

However, 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 data model. JAXB (*Java Architecture for XML Binding*) from Sun [10] also generates an API from a DTD or an XML Schema, and provides tools to automate the mapping between XML documents and Java objects. The generated APIs from JAXB or our tool are different like the aims of these two tools; JAXB can only be used to access, update, or validate XML documents.

To assist DSL designers writing semantic analyses (e.g. evaluators), we also generate a default top-down tree walker based on the visitor design pattern [4, 8]. This default visitor can be, either extended by inheritance (by overriding some of the visit methods), or by plugging aspects (additional code that will be weaved with the visitor code) [6]. Using this feature, any XML document conformed to its XML schema can be semantically analysed with our tool.

3. COMPONENTS

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, Web-Services technology has appeared with the possibility of listing component services in catalogs (UDDI - *Universal Description, Discover and Integration*).

The three main challenges in component technologies are the following:

- To extend the classical method call to take into account the execution environment (three-tier architecture, the Internet, the messaging service, the database access) without modifying the business code;
- To extend the classical interface notion to be able to discover the available and required services (such as introspection in Java Bean technology) 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 that avoids mixing functional and 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 [12]. This explains the research effort undertaken 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 enable interconnections with other 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 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 to the programmer. Additionally, it is very easy to adapt the architecture to introduce a new communication mechanism.

Figure 5 gives an example of component description (graph component) and Figure 6 its associated visual representation (showing the connections).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="graph" type="graph"
      extends="abstractContainer">
  <containerclass name="GraphContainer"/>
  <facadeclass name="GraphFacade"/>
  <dependance name="koala-graphics"
      jar="koala-graphics.jar"/>
  <attribute name="nodeType"
      javatype="java.lang.String"/>
  <input name="addComponent" method="addNode">
    <parameter name="nodeName"
        javatype="java.lang.String"/>
    <parameter name="nodeColor"
        javatype="java.lang.String"/>
  </input>
  <input name="addEdge" method="addEdge">
    <parameter name="srcNodeName"
        javatype="java.lang.String"/>
    <parameter name="destNodeName"
        javatype="java.lang.String"/>
  </input>
</component>
```

Figure 5: Graph component description

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

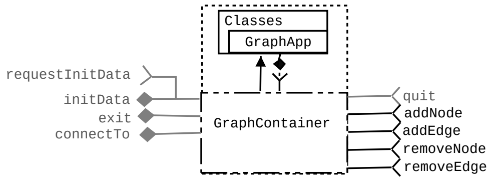


Figure 6: Graph component

Despite this approach, 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. This rich communication is possible due to our data model and the use of XML technologies (the serialized form of the documents). All components which conform with the same data model can exchange rich information between their business code. In fact, our communication protocol is very close to SOAP, except the data part. The implementation of our model was made easier due to the use of XML technologies.

Moreover, our component and deployment languages (see Figure 7 for an example) are described in XML format. Our component manager uses these two neutral formats (XML) to instantiate components and to establish connections between them. Figure 8 summarizes the operations of our component manager with the various XML files that are used.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<application repository="file:stlib/"  library="file:lib/">
  <load_component jar="view.jar"  name="glayout"/>
  <load_component jar="lml.jar" name="lml"/>
  <load_component jar="tiny.jar"
      url="file:extralib/tiny.jar" name="tiny"/>
  <connectTo id_src="ComponentManager" type_dest="glayout">
    <attribute name="docRef"
        value="file:resources/lml/boot.lml"/>
    <attribute name="xslTransform"
        value="file:resources/xsl/lml2bml.xsl"/>
    <attribute name="behaviors"
        value="file:resources/behaviors/bootbehav.xml"/>
  </connectTo>
</application>
```

Figure 7: Example of deployment description

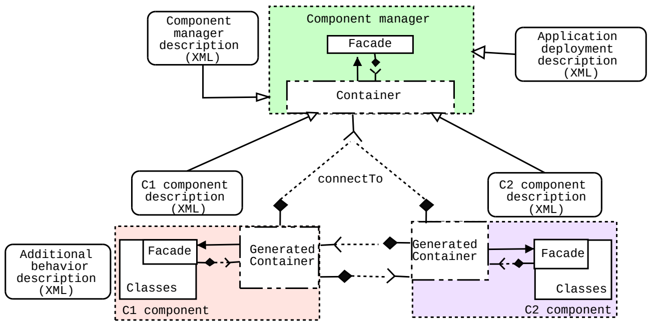

Component manager description (XML)
Component manager
Facade
Container
Application deployment description (XML)
connectTo
C1 component description (XML)
C2 component description (XML)
Additional behavior description (XML)
Facade
Classes
Generated Container
C1 component
Generated Container
Facade
Classes
C2 component

Figure 8: Functional diagram of component manager

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 9) to make the exportation of the produced tools easier.

From our projection experience, we can say that these three component technologies (Web Services, CCM, EJB) would have not

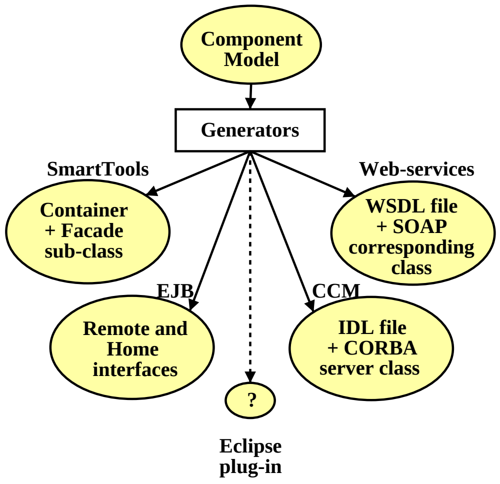

Component Model
Generators
SmartTools
Web-services
Container + Facade sub-class
WSDL file + SOAP corresponding class
EJB
CCM
Remote and Home interfaces
IDL file + CORBA server class
?
Eclipse plug-in

Figure 9: Component model transformations

fitted in with our needs of connections and component model.

Our experience shows that there are many advantages in creating 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 are adaptative and capable of evolving.

4. GRAPHICAL USER INTERFACES

The graphical user interfaces that enable interactive applications, 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 (ubiquitous computing) and be accessible through a Web interface. By taking into account different device variations and Web interfaces, the better an application user interface.

Furthermore, as the number of DSLs increases, a flexible and quick approach is needed to easily design and implement interfaces (or pretty-printers) specific to one model or domain. In this context, visual programming can be very useful in realizing dedicated programming environments as many DSL designers do not have a deep knowledge in computer science.

With our tool and 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). Our approach (outlined in Figure 11) intensively uses 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.

In this way, the implementation of the graphic part of our tool was quick as mainly based on existing tools dedicated to these standards, and benefits from any bug-fix or evolution of these tools.

By default, there are also generic displays (none domain-specific) available to show any tree regardless of its language membership.

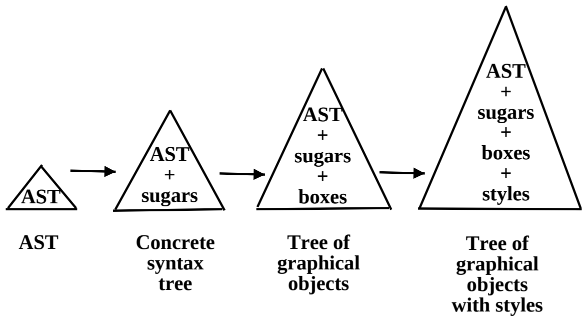

AST
AST + sugars
AST + sugars + boxes
AST + sugars + boxes + styles
AST
Concrete syntax tree
Tree of graphical objects
Tree of graphical objects with styles

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

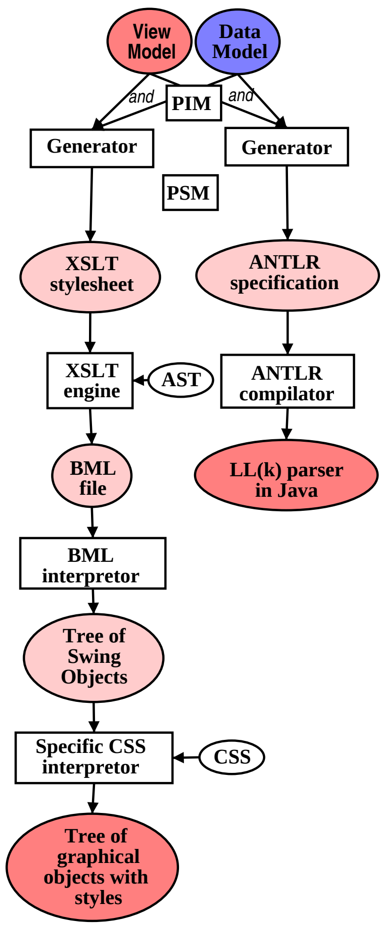


Figure 11: Implementation chain to obtain a graphical view or a parser

As our view model (DSL) 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). This feature is appealing to DSL designers who want to give a more readable and editable syntax to their DSLs.

To be able to export our graphical views on the network, we have chosen the BML format [5] that describes all the graphical objects contained in a view to be created. The second advantage of this format is the effective object creation (Swing objects) that takes place on the view components (clients) and 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. Therefore, 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. Indeed, a GUI can be considered as a tree of graphical objects (windows, tabs, panels, views, menu, etc.). Using this innovative approach, we can reuse all the tree manipulation methods (insert a node, remove a node, etc.) and implementations to obtain a view of the GUI. In this way, the GUI is only a particular view of this tree and can be serialized. For example, Figure 12 shows two views of the GUI description given Figure 13: a textual view at left, and a graphic one the GUI itself. To represent the GUI description, we have defined a simple GUI-specific language (LML) useful to configure the GUI according to the applications.

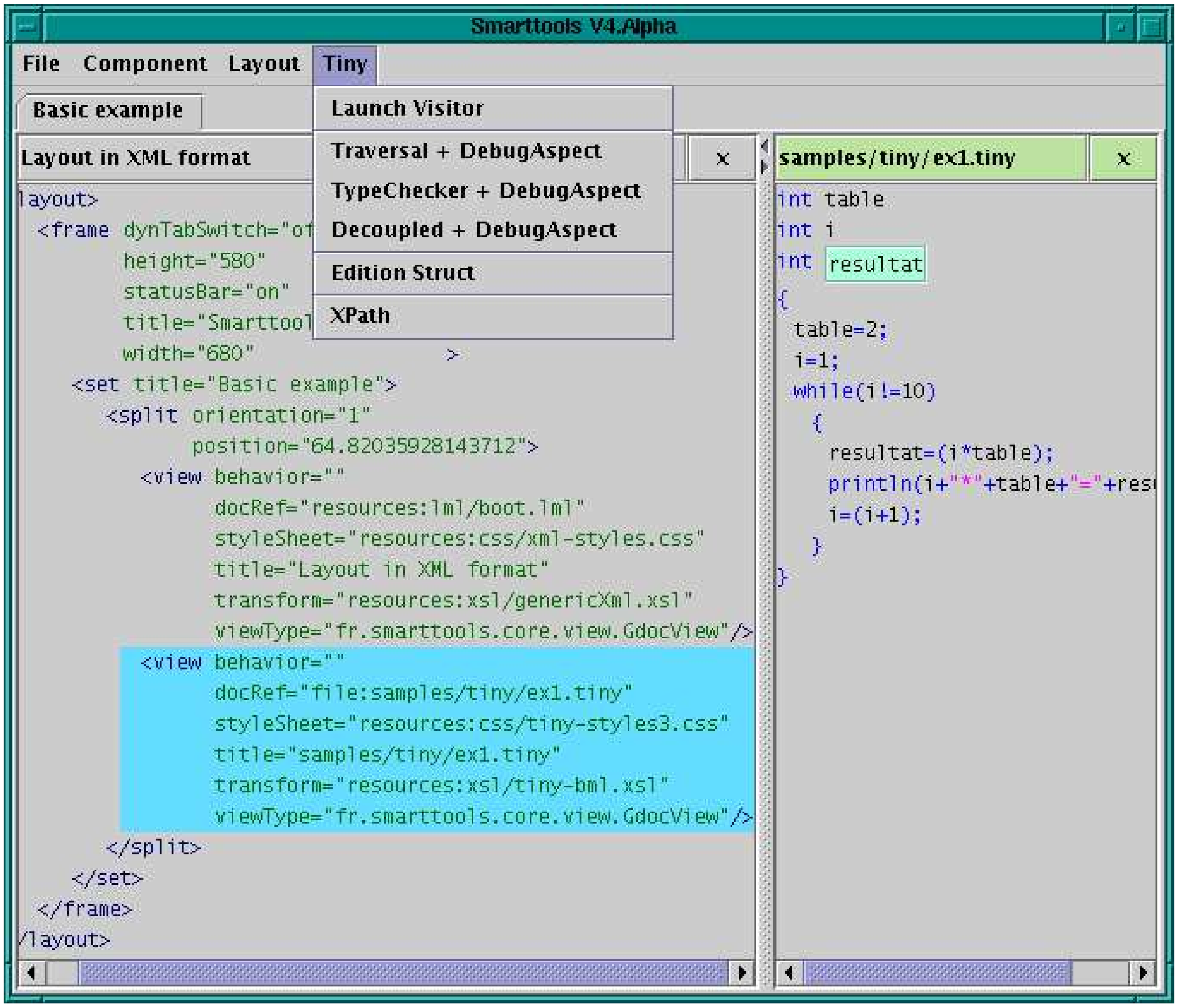
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE layout SYSTEM "file:resources/lml.dtd">
<layout>
  <frame title="Smarttools V4.Alpha" statusBar="on" width="680"
         height="580" dynTabSwitch="off">
    <set title="Basic example">
      <split orientation="1" position="68">
       <view title="Layout in XML format"
              behavior=""
              viewType="fr.smarttools.core.view.GdocView"
              docRef="resources:lml/boot.lml"
              styleSheet="resources:css/xml-styles.css"
              transform="resources:xsl/genericXml.xsl" />
       <view title="samples/tiny/ex1.tiny"
              behavior=""
              viewType="fr.smarttools.core.view.GdocView"
              docRef="file:samples/tiny/ex1.tiny"
              styleSheet="resources:css/tiny-styles3.css"
              transform="resources:xsl/tiny-bml.xsl" />
         </split>
    </set>
  </frame>
</layout>
```

Figure 13: Example of a GUI description (the corresponding screenshot shown in Figure 12)

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).

5. CONCLUSION

With the development of our tool (SMARTTOOLS), we have validated a new approach for software development mainly based on transformation or generation from programming-language-independent and domain-specific models. Thanks to Generative Programming, we can integrate, very easily, new programming paradigms and technologies from the models into the target implementation programming language by only updating the generators associated with each model (data model, component model, visualization model, GUI model); making the application easy to adapt and evolve - the main advantage of this design approach. These different generators



```
layout>
  <frame dynTabSwitch="of
         height="580"
         statusBar="on"
         title="Smarttool
         width="680"                    >
     <set title="Basic example">
        <split orientation="1"
               position="64.82035928143712">
           <view behavior=""
                 docRef="resources:lml/boot.lml"
                 styleSheet="resources:css/xml-styles.css"
                 title="Layout in XML format"
                 transform="resources:xsl/genericXml.xsl"
                 viewType="fr.smarttools.core.view.GdocView"/>
           <view behavior=""
                 docRef="file:samples/tiny/ex1.tiny"
                 styleSheet="resources:css/tiny-styles3.css"
                 title="samples/tiny/ex1.tiny"
                 transform="resources:xsl/tiny-bml.xsl"
                 viewType="fr.smarttools.core.view.GdocView"/>
        </split>
     </set>
  </frame>
/layout>
```

```
int table
int i
int resultat
{
 table=2;
 i=1;
 while(i!=10)
   {
     resultat=(i*table);
     println(i+"*"+table+"="+resu
     i=(i+1);
   }
}
```

Figure 12: This picture shows two views of the GUI description given Figure 13: a textual view at left, and a graphic one the GUI itself.

provide the design methodologies that are parameterized through those models strictly restricted to the domain.

The approach to follow when developing an application should, according to our experience, be to design DSLs specific to your concerns (e.g. data, GUI, component) and implement the generators that would produce the platform-dependant glue from the DSL documents (instantiated models) with the possibility to integrate specific business code (e.g. semantics analysis). An application is, thus, composed by a set of DSL documents, the generated application-specific glue, and the business code (see Figure 14). 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 programming languages and validate the interest of separating the concerns through different models.

With our experience, we can say that XML technologies are very helpful in the process of implementing an application and also make it more open than when non-propriatory format and tools are used. Indeed, this standard interoperable format has existing tools to validate, parse, and manipulate (e.g. transform) the content. An application that uses XML technologies benefits from their tools, and when these standards and their associated tools evolve, it also evolves and for free. We also showed that XML technologies can be extremely useful for the design and the implementation of applications. These technologies should not be restricted to the field of the document treatment and, in the future, will be more used in software development.

We are starting to investigate the relationships between UML profile, RDFS (*Resource Description Language Framework Schema*) for the Web Semantic, semi-structured databases, and our own data model. Our aims are to broaden the scope of our tool by being more open and able to translate a data model between different domains (language, document, modeling, knowledge, database), and also to benefit from their tools (as we have done with the document domain).

On the architecture, we want to pursue our efforts on Web Services in order to generate a process that orchestrates the Web Ser-

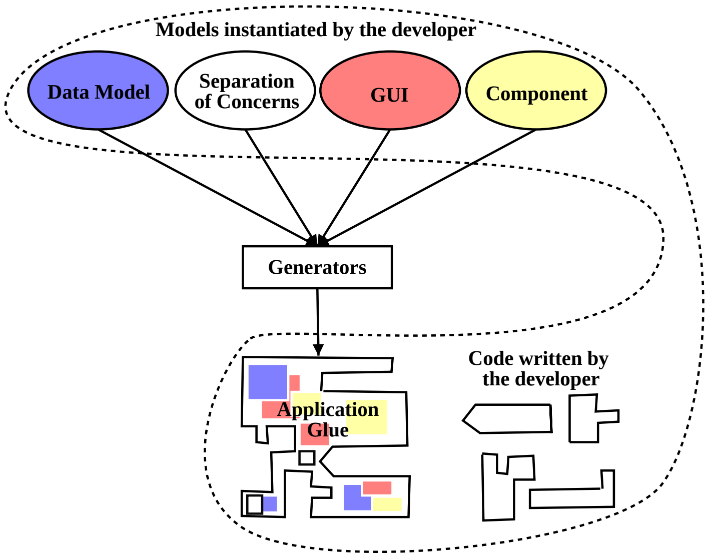


Figure 14: Model instances + generated application glue + business code = an application

vices involved in an application (from a deployment document, the component descriptions, and the additional extension service descriptions) as the aim of our tool is to help designing applications for DSLs.

6. REFERENCES

[1] 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`.

[2] S. Cook and S. Kent. The Tool Factory. In *OOPSLA'2003, workshop on Generative Techniques in the context of MDA*, Anaheim - USA, October 2003. `http://www.softmetaware.com/oopsla2003/cook.pdf`.

[3] 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/`.

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

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

[6] 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`.

[7] OMG. MDA - Model-Driven Architecture. `http://www.omg.org/mda`.

[8] 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`.

[9] 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.`

[10] Sun. *The Java Architecture for XML Binding (JAXB)*, January 2003. `http://java.sun.com/xml/jaxb/`.

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

[12] 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`.